전산 SMP 7주차

2015. 11. 17

김범수
bskim45@gmail.com

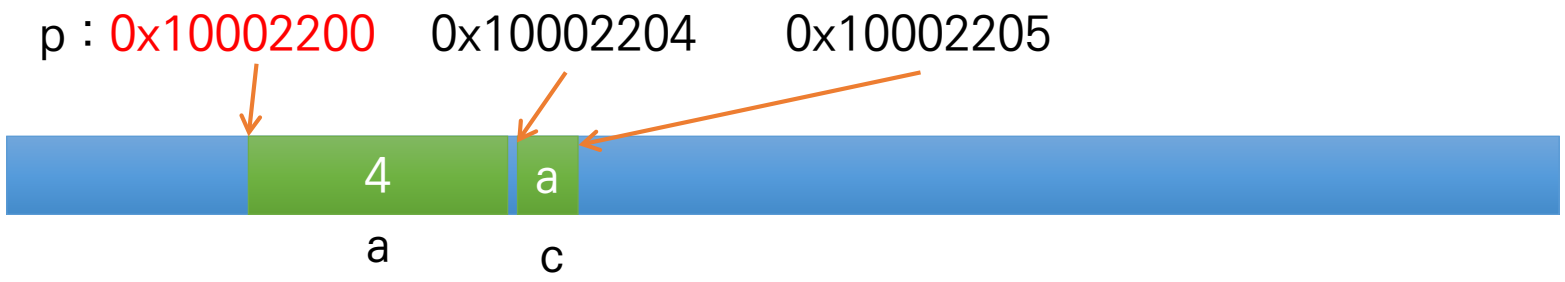
Special thanks to 박기석 (kisuk0521@gmail.com)

지난 내용 복습

Pointer and its applications

Pointer란?

- 데이터 접근에 사용되는 주소를 저장하는 타입 - 포인터도 타입이다
- int a = 4; char c = 'a'; int *p = &a;



- 메모리는 긴 일차원 공간으로 볼 수 있고, 각 byte는 자신만의 주소를 가지고 있다.

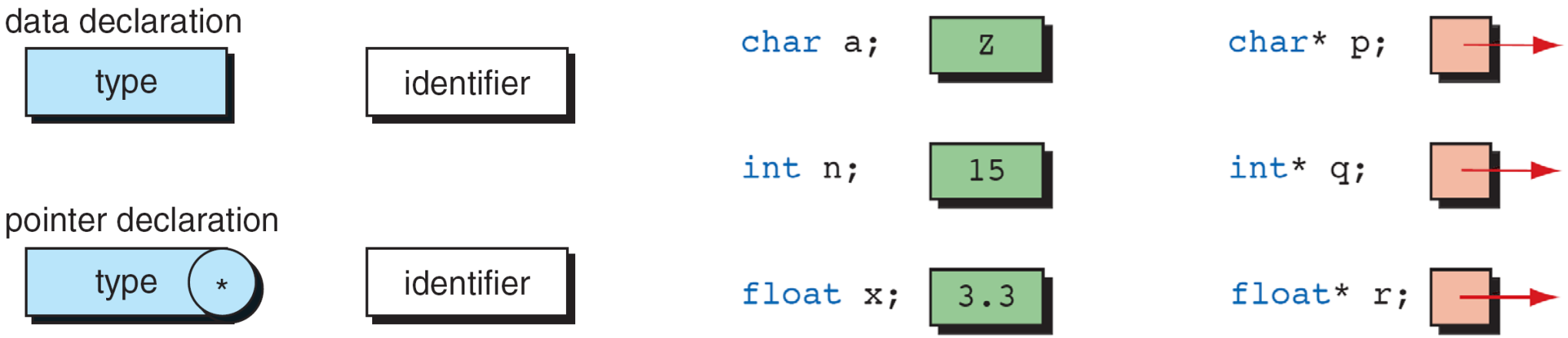
Pointer 선언, 사용

- Declaration

  int *ptr;

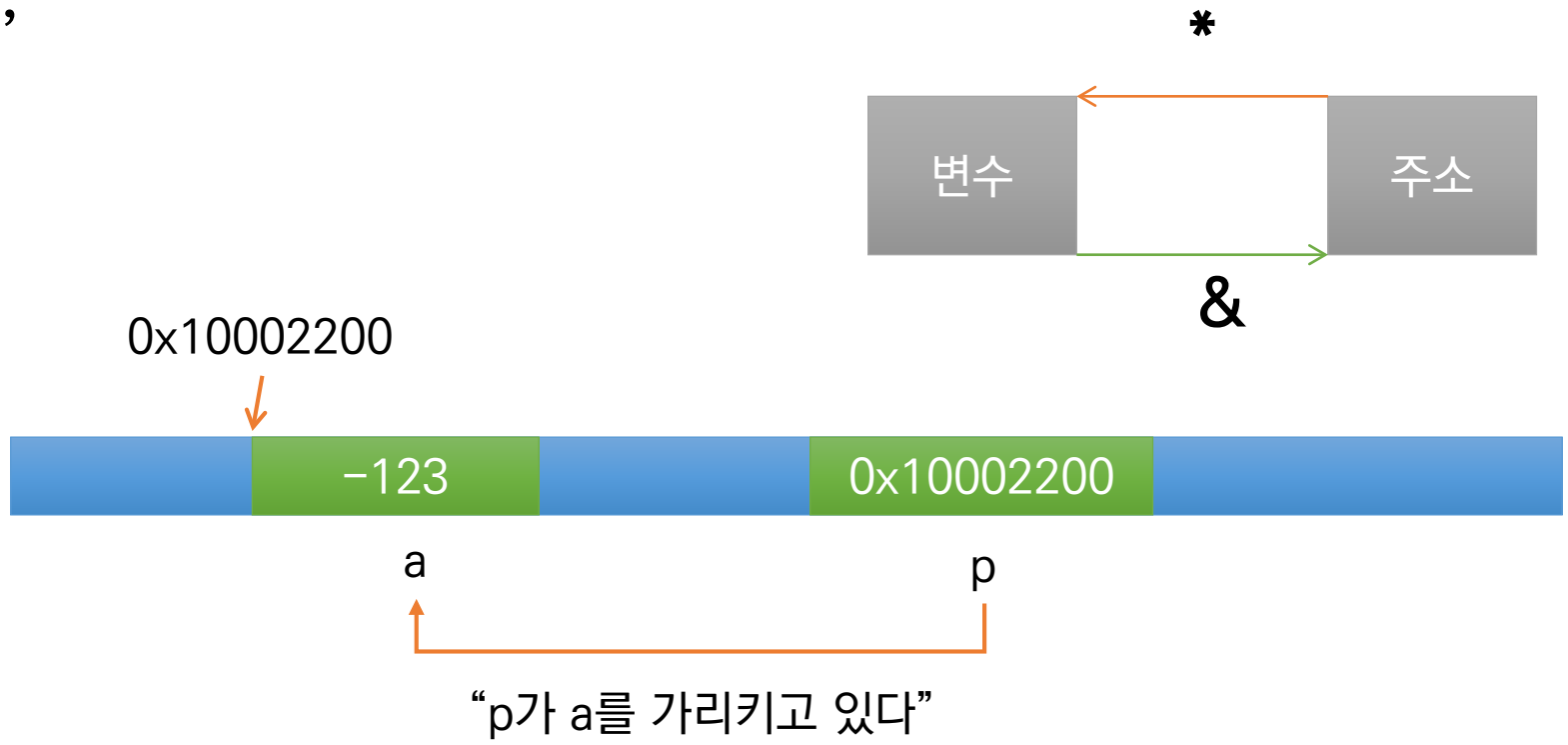
  → 정수형 데이터가 저장된 메모리의 위치(주소)를 저장할 수있는 포인터 변수

- 포인터의 타입 크기
  - 4 byte! (32 bit) ex) int : 4byte, char : 1 byte



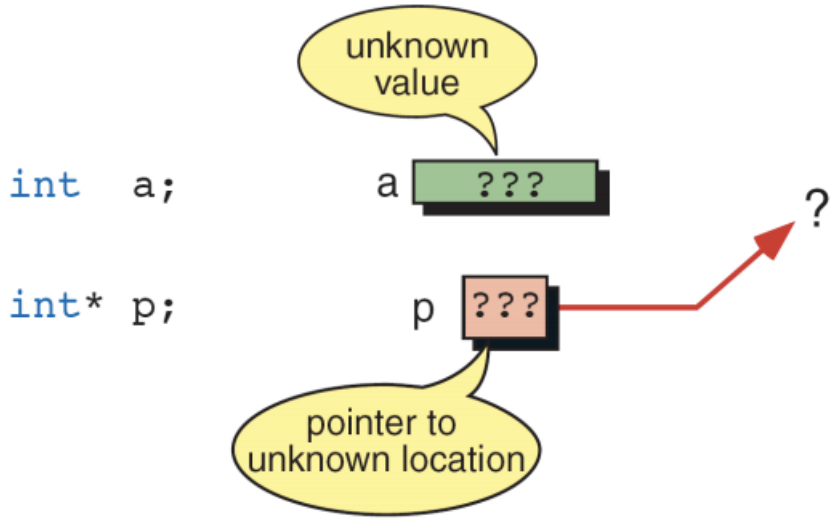
Pointer 선언, 사용

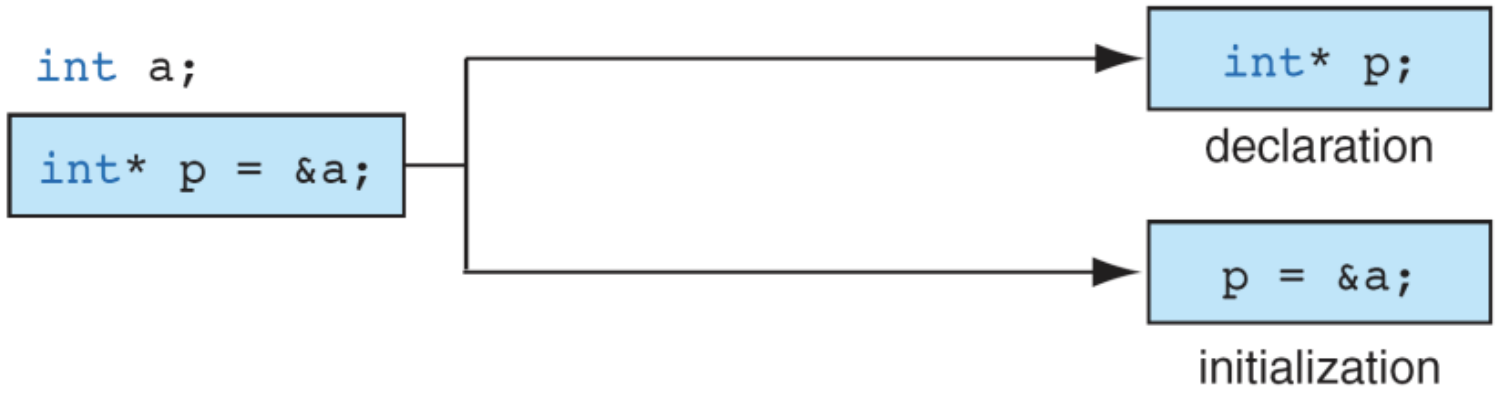
```
int a = -123;
int *p = &a;
```



포인터 변수도 결국 ‘변수’이기 때문에 자신이 가리키는 주소를 메모리 어딘가에 저장하고 있다.

초기화 되지 않은 변수를 사용하면 위험하다





함수-Pointer 사용 특징

- 포인터 변수도 일반 변수처럼 함수 인자, 반환형으로 사용 가능!
- 함수가 여러 개의 리턴값을 가져야 할 때 - 여러 인자들을 포인터로 받아서 수정할 수 있다!
- 실제 그 메모리 위치를 찾아가 값을 바꿀 수 있다.

- 주의!!
  함수 안에서 만들어진 local variable을 가리키는 포인터는 반환할 수 없다.

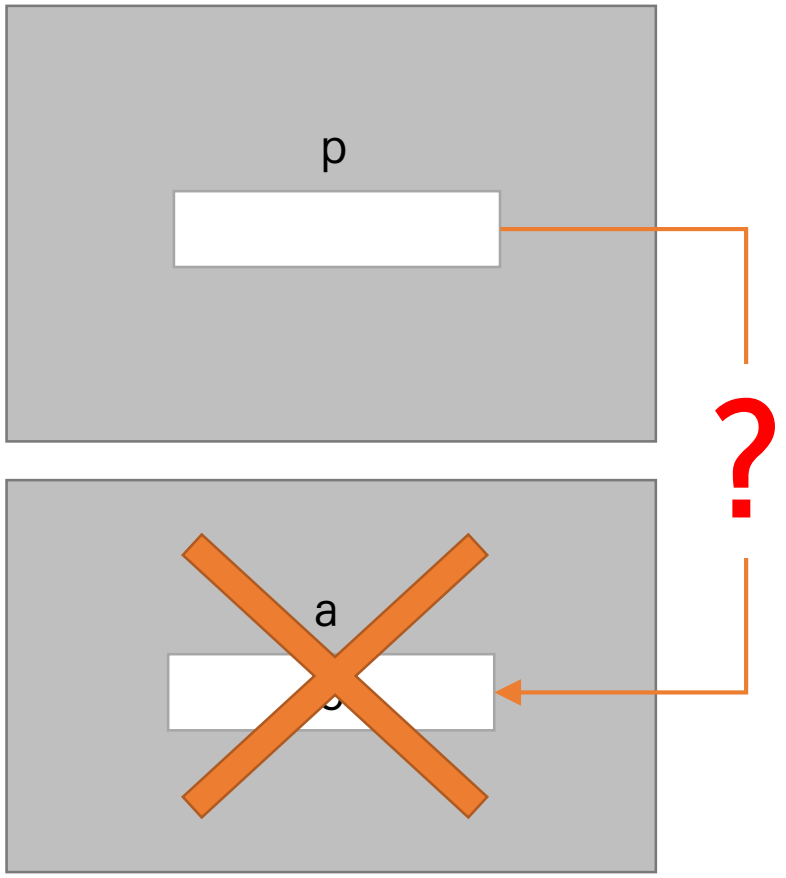
Local Variable의 주소를 리턴하는 경우

- Local variable는 그 함수(블럭)이 끝나면 사라진다.

→ 알 수 없는 곳으로의 포인터를 반환하는 것과 같다.

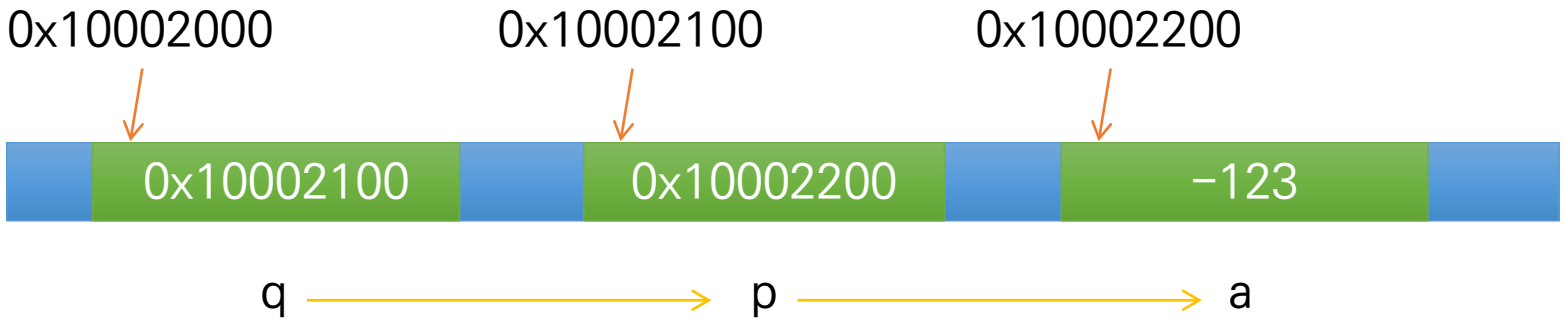
```
int main (void) {

    int* p;

    p = fun();
    printf("%d", *p);
}
```

```
int* fun (void) {

    int a = 5;

    return &a;
}
```



Pointer to Pointer (다중포인터)

- 포인터를 가리키는 포인터!
- 포인터도 주소를 저장하는 하나의 변수! 이기 때문에 메모리 특정 위치에 저장된다.




```
int a = -123;
int *p = &a;
int **q = &p;
```

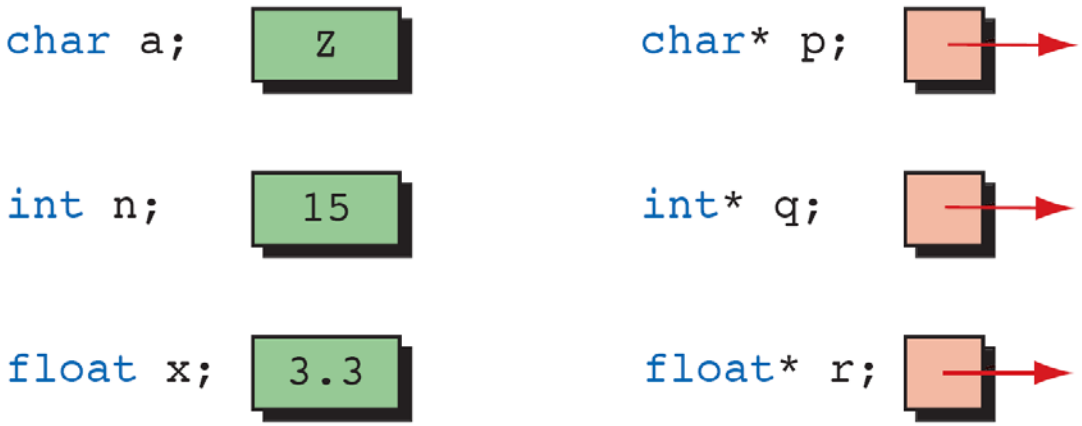
Pointer를 선언할 때 타입이 필요한 이유

int *p; char *q;

- 어차피 크기는 4byte(32 bit)로 다 똑같은 거 아닌가?
- 컴퓨터가 포인터로 메모리에 접근해 데이터를 읽어올 때!
  - 그 포인터 타입에 따라 가져오는 byte 수가 달라진다.

→“int * 포인터면 여기서 부터 4byte 까지 읽어서 정수로 쳐”

→“char * 포인터면 여기서 부터 1byte 만 읽어서 문자로 쳐”

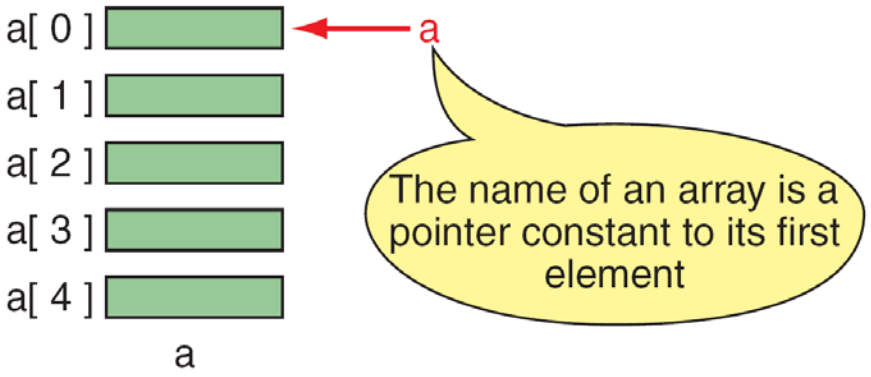


void *

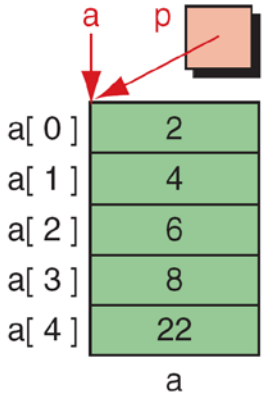
- 어떤 데이터 타입과도 연관되지 않는 일반적 형태의 포인터
  - 어떤 포인터 값도 void pointer에 넣어줄 수 있다.
  - 어떤 포인터 값에도 void pointer 값을 넣어줄 수 있다.
- 단, void pointer는 그 상태 그대로는 dereferencing 할 수 없다!
  Why? 타입이 없음 → 내가 보려는 데이터의 타입이 뭐지? → compile error
- casting을 통해서 dereferencing 할 수 있다.

```
int a = -123;
void *p;
p = &a;
printf("%d", *(int *)p);
```

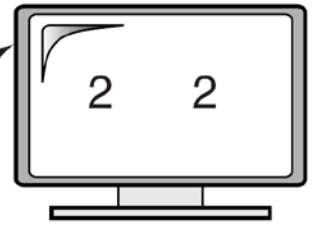
배열 포인터 (Pointer to Arrays)

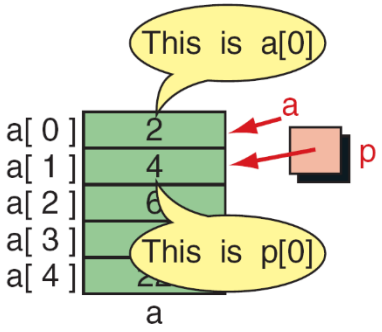


a ⟷ &a[0]

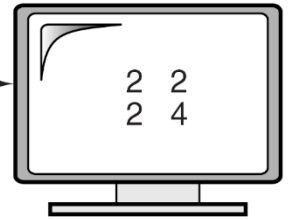



```
#include <stdio.h>
int main (void)
{
int  a[5] = {2, 4, 6, 8, 22};
int* p    =  a;
  …
  printf("%d %d\n", a[0], *p);
  …
  return 0;
} // main
```



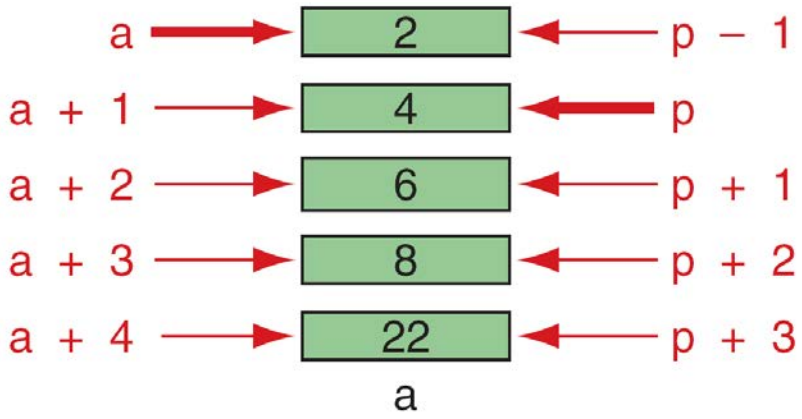


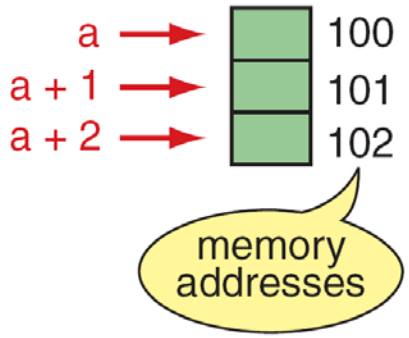

```
#include <stdio.h>
int main (void)
{
  int  a[5] = {2, 4, 6, 8, 22};
  int* p;
  …
  p = &a[1];
  printf("%d %d", a[0], p[-1]);
  printf("\n");
  printf("%d %d", a[1], p[0]);
  …
} // main
```



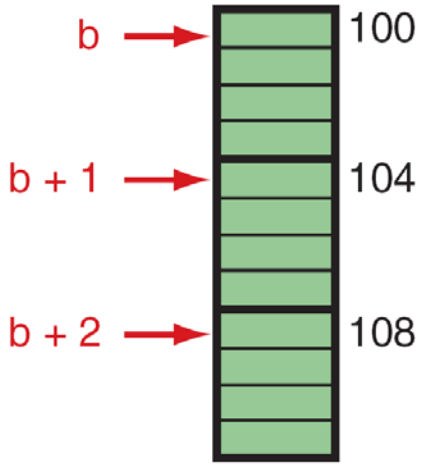
포인터 연산

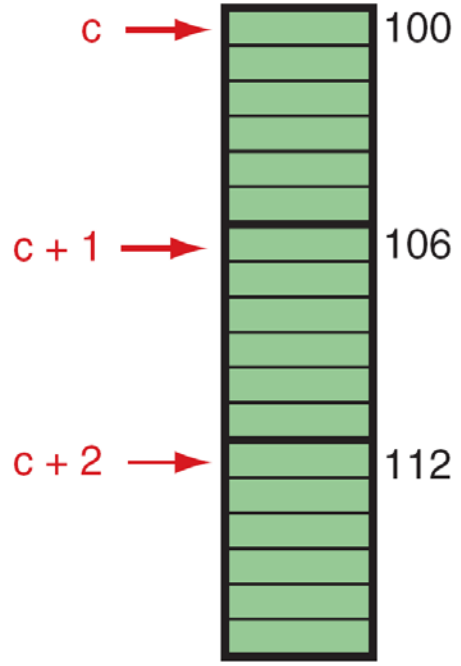
- 포인터 p → p ± n → p + n * (sizeof (one element))






```
char  a[3];
int   b[3];
float c[3];
```





Arithmetic Operations on Pointers

- p + 5, 5 + p, p - 5
- p1 - p2
- p++, --p
- p1 >= p2
- p1 != p2

Long Form	Short Form
if (ptr == NULL)	if (!ptr)
if (ptr != NULL)	if (ptr)

Pointer Arithmetic

- 두 포인터 p, q에 대하여 가능한 연산
  - p, q 모두 같은 데이터 형을 다루는 포인터여야 한다.
  - p – q, q – p : p의 위치와 q의 위치의 차이를 단위수로 계산
  - (int)p – (int)q, (int)q – (int)p : p와 q의 주소 값 차이

*** 주의: p + q는 사용할 수 없다. (무엇보다 의미가 없다)**

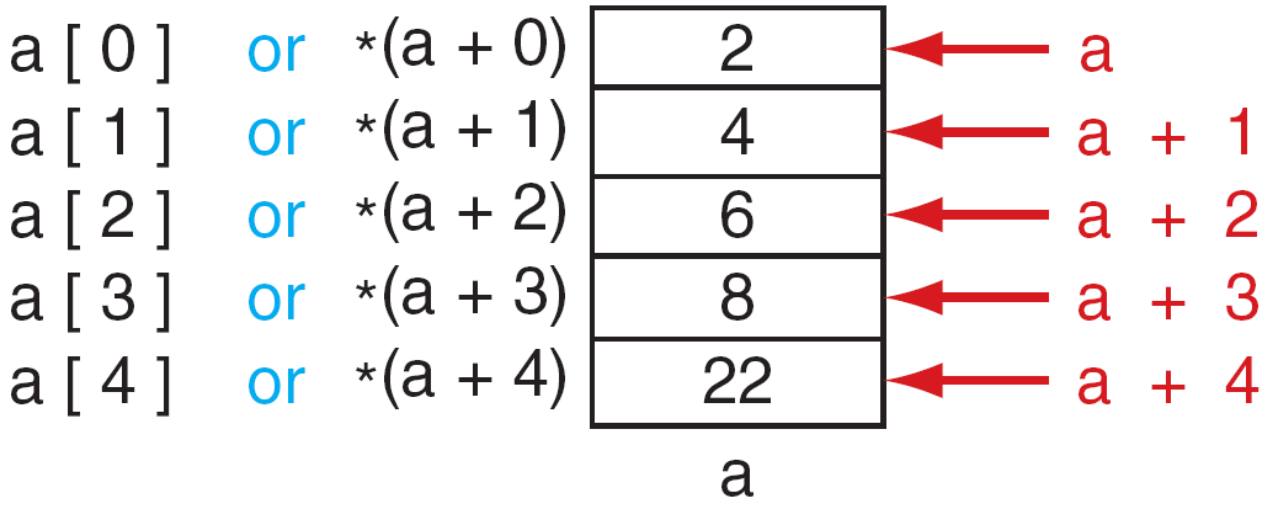
왜?

(0x0000~0xffff)+(0x0000~0xffff) = ??

주소값 overflow가 일어날 수 있음

and 꼭 overflow가 아니더라도
사용할 수 있는 메모리 주소는 한정되어 있기 때문

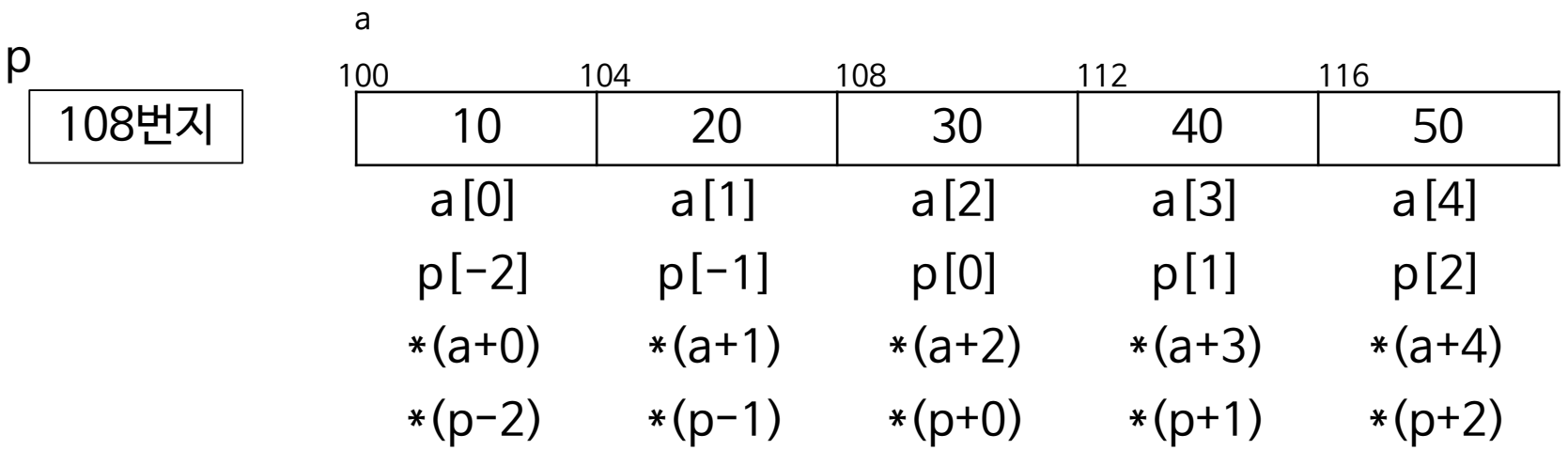
Dereferencing



*(a + i) ⟺ a[i]

포인터와 배열

```
int a[5]={10, 20, 30, 40, 50};
int *p=&a[2];

printf("%d %d %d %d₩n",a[2], *(a+2), p[0], *(p+0)); // 30출력
printf("%d %d %d %d₩n",a[1], *(a+1), p[-1], *(p-1)); // 20출력
```

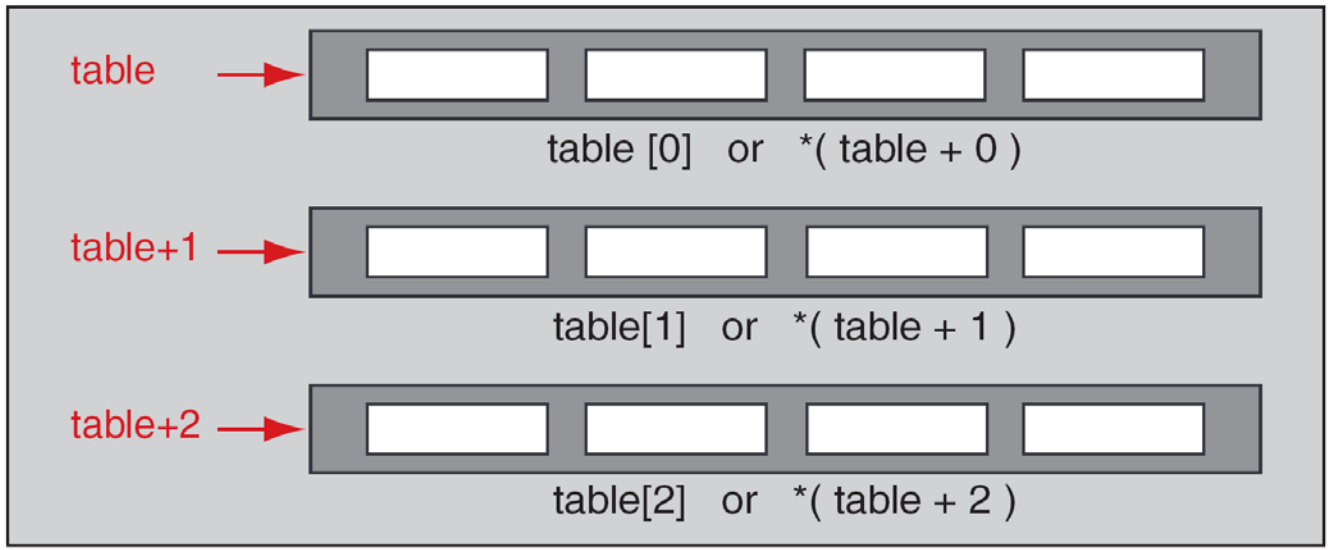


2차원 Array

table[2] == *(table+2)

table[2] [1]
== (*(table+2)) [1]
==*(table[2]+1)
==*(*(table+2)+1)

- 헷갈리니 다차원 배열에서는 Index를 사용하자



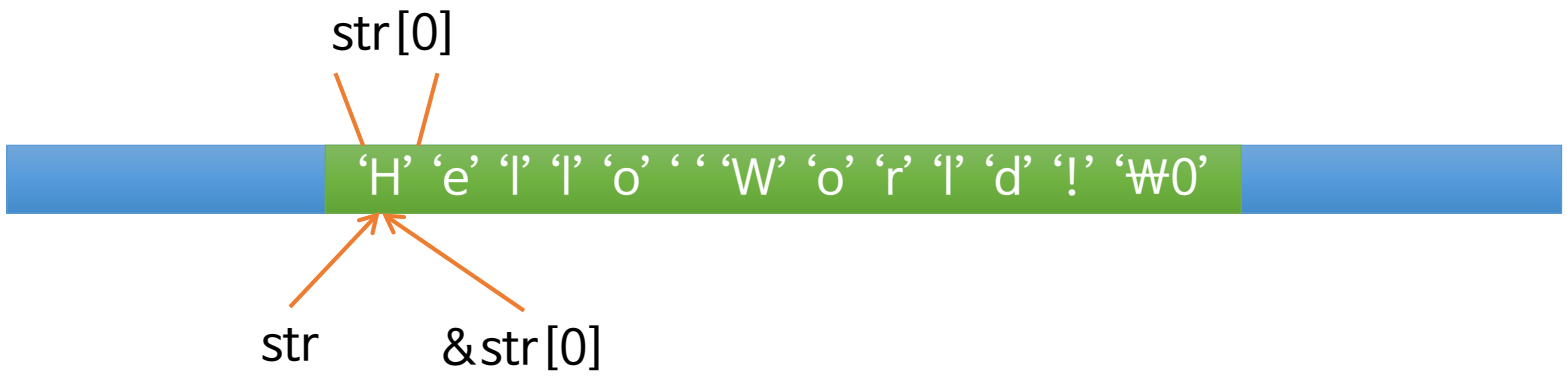
int table[3][4] ;

```
for (i = 0; i < 3; i++)
  {
   for (j = 0; j < 4; j++)
      printf("%6d", *(*(table + i) + j));
   printf( "\n" );
  } // for i
```

Print Table

문자열과 포인터

- 다시한번! 여러개의 문자+‘₩0’ 이 들어있는 배열
- 배열의 이름은 첫번째 element의 주소와 같다.

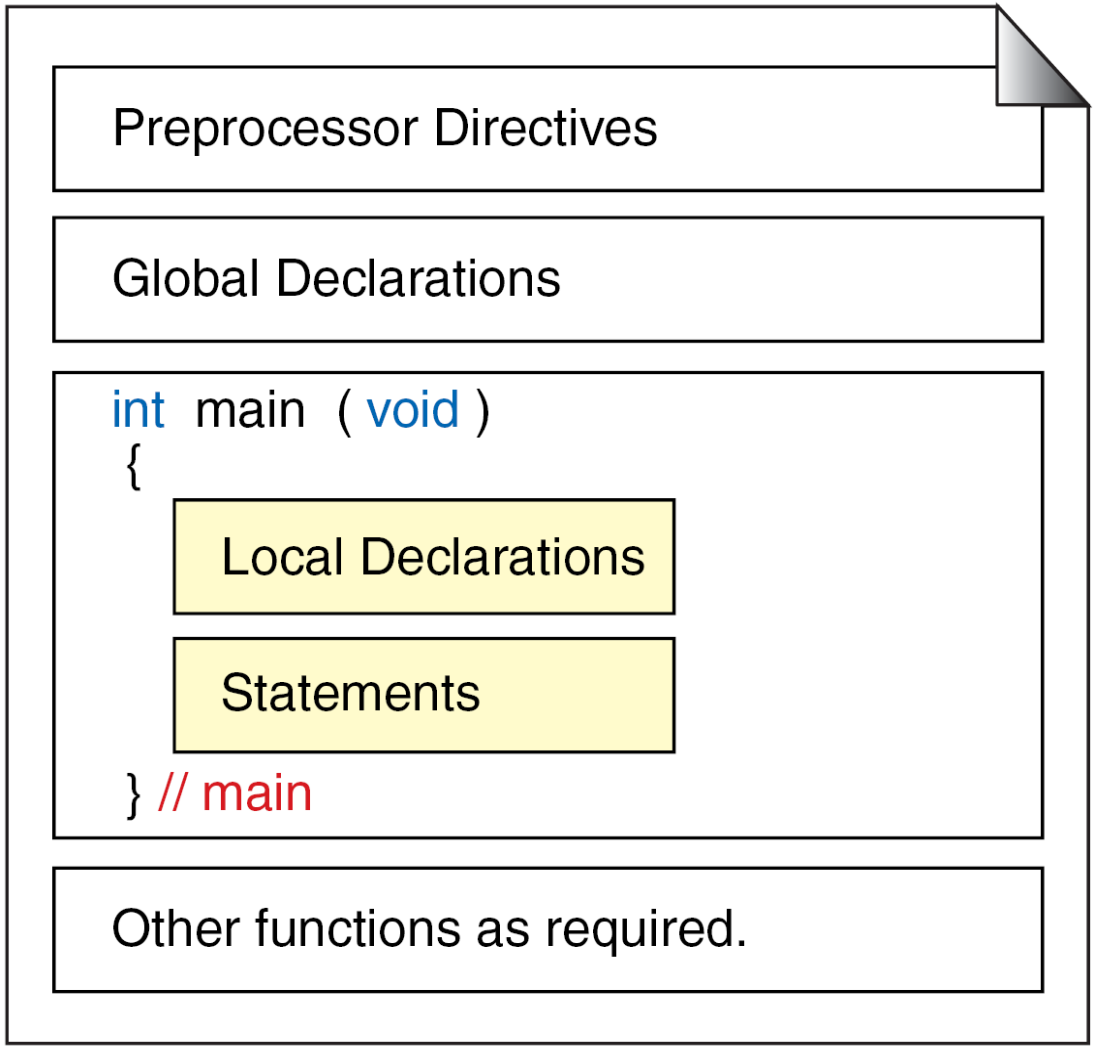


- 그래서 scanf로 %s 받을 때 & 안 붙인다!

배열 함수에 넘겨주기

- 1차원 배열
  Int ary1 [10];
  void function(int *ary);
  void function(int ary[]);
- 2차원 배열
  int ary2 [5] [6];
  void function(int (*ary) [6]);
  void function(int ary[] [6]);
- 3차원 배열
  (int ary[] [3] [4])

Scopes


Preprocessor Directives
Global Declarations
int main (void)
{
Local Declarations
Statements
} // main
Other functions as required.

Storage Class (변수의 존재기간과 접근범위)

	지역변수	정적변수	전역변수	register변수
지정자	auto	static	extern	register
저장장소	스택	정적 데이터 영역	정적 데이터 영역	레지스터
선언위치	함수내부	함수내부/외부	함수내부/외부	함수내부
유효범위	함수내부	함수내부/외부	프로그램 전체	함수내부
생존기간	함수종료시	프로그램 종료시	프로그램 종료시	함수종료시
초기값	초기화 안됨	0으로 초기화	0으로 초기화	초기화 안됨

언제 어떤걸 쓰나요

- 일반적으로 전역변수의 사용을 자제하고, 지역변수인 자동변수를 이용

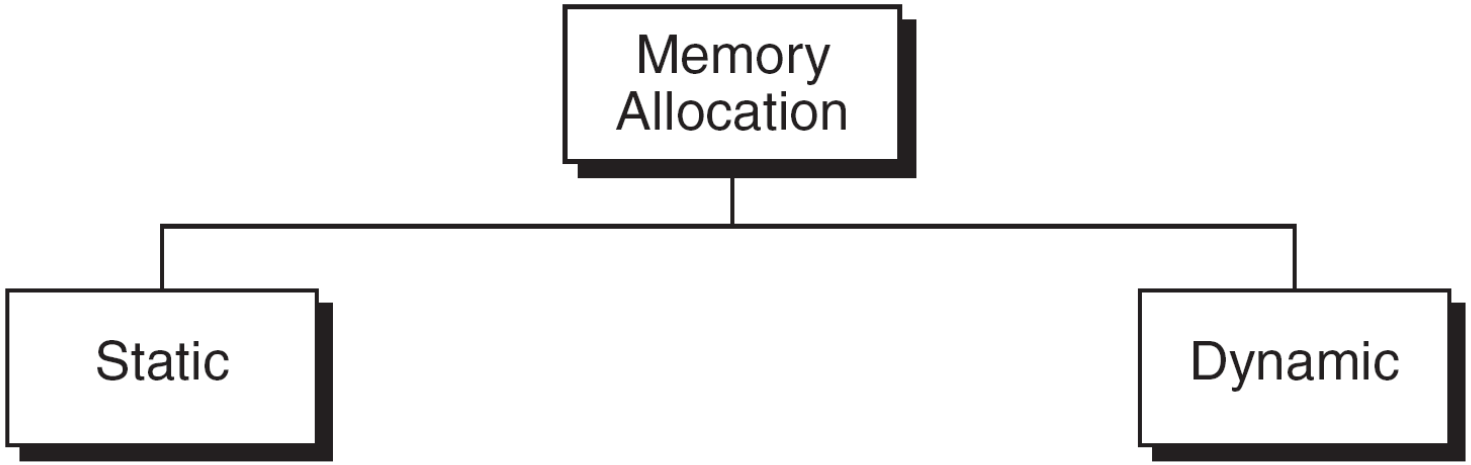
- 실행속도를 빠르게 하고 싶을 때: 레지스터 변수
- 함수나 블록 내부에서 계속 값을 저장하고 싶을 때: 정적 지역변수
- 해당 파일 내부에서만 변수를 공유하고 싶을 때: 정적 전역변수
- 프로그램의 모든 영역에서 값을 공유하고 싶을 때: 전역 변수

- 전역변수는 모든 함수에서 공유할 수 있는 저장공간을 이용할 수 있는 장점이 있지만 어디선가 잘못 바꾸면 프로그램 전체에 영향을 미친다→ 위험해
- 함수의 인자(argument)로 선언된 변수는 지역변수와 같이 함수 내부에서만 유효

Dynamic Memory Allocation

동적 할당

Two types of memory allocation



왜 쓰나요?

- 배열의 크기를 입력 받아 그 크기만큼의 배열을 만들고 싶을 때

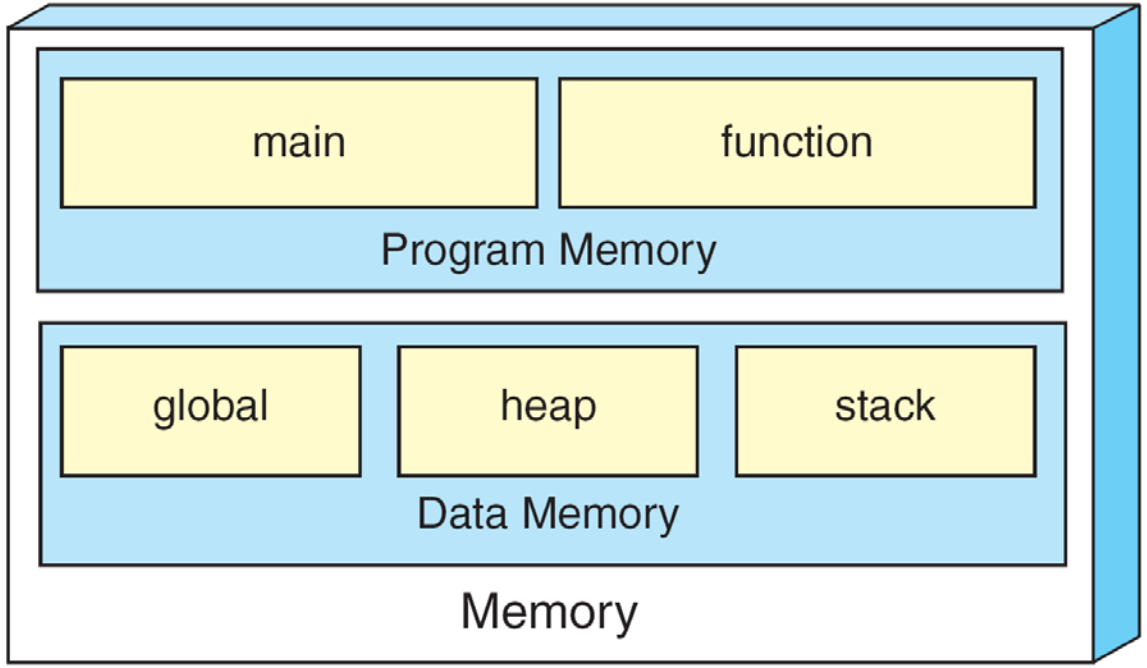
```
int main(void) {
    int size;

    scanf("%d", &size);

    int array[size];
}
```

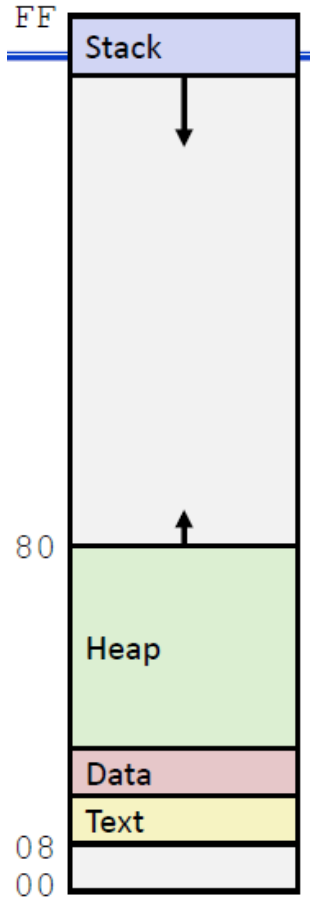
Compile Error! 선언은 항상 맨 위에 와야 한다.
선언 후에 배열이 할당되어야 하는데… 이럴 때는 어떻게?

- 프로그램 실행 중에 메모리를 할당해 데이터를 저장할 공간을 생성하는 방법

A Conceptual View of Memory

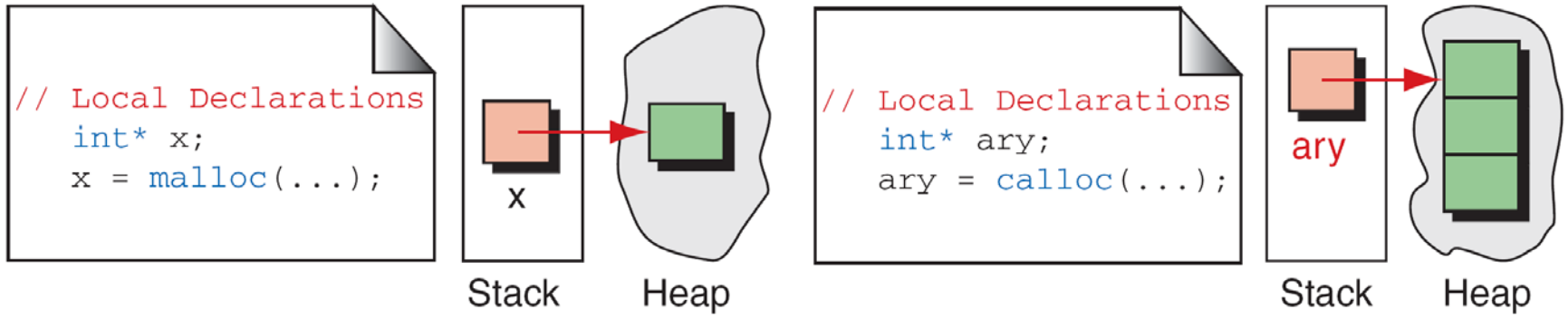
- We can refer to memory allocated in the heap only through a pointer.





Accessing Dynamic Memory

(a) Static Memory Allocation



(b) Dynamic Memory Allocation

Dynamic Memory Allocation

- `<stdlib.h>`

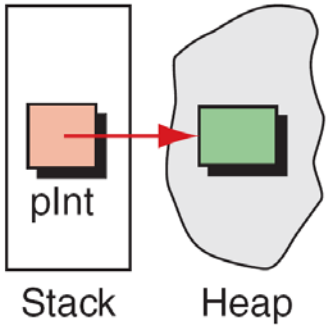
- `void *malloc (size_t size);`
- `void *calloc (size_t count, size_t size);`
- `void *realloc (void* ptr, size_t newSize);`
- `void free (void* ptr);`

malloc

- 지정하는 크기(byte) 만큼 메모리(힙)를 할당
- 기본적으로 void *로 리턴 → 원하는 타입으로 casting 해줘야 한다
- 할당에 실패한 경우 NULL 리턴

- void *malloc (size_t size);

```
if (!(pInt = malloc(sizeof(int))))
    // No memory available
    exit (100) ;
// Memory available
…
```



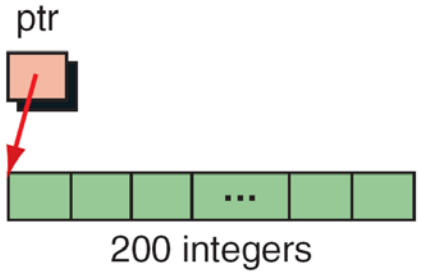
- int* p = (int *)malloc(sizeof(int) * 4);
  - int형의 크기가 4byte이기 때문에 16byte의 공간이 동적으로 할당
- char *str = (char *)malloc(sizeof(char) * 10);

calloc

- `void *calloc (size_t count, size_t size);`
- count * size 만큼의 메모리를 할당하고 해당 힙 영역을 0으로 초기화하여 반환

- `int *p = (int *)calloc(4, sizeof(int));`
- `char *str = (char *)calloc(10, sizeof(char));`

`malloc(sizeof(int)*4)` ↔ `calloc(4, sizeof(int))`



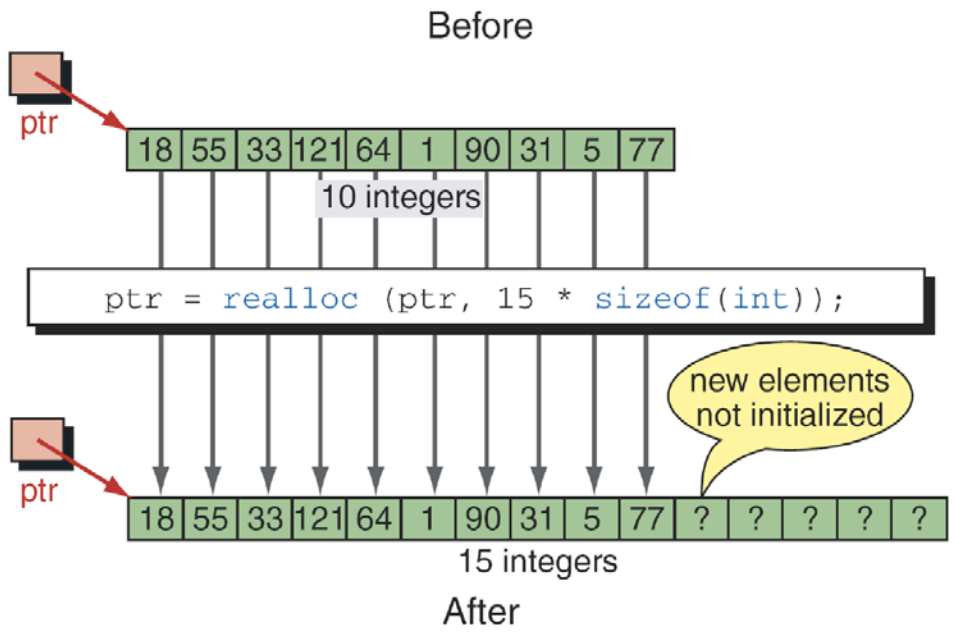

```
if (!(ptr = (int*)calloc (200, sizeof(int))))
    // No memory available
    exit (100) ;

// Memory available
…
```

realloc

- 할당받은 메모리 공간을 새로운 크기로 재할당

- `void *realloc(void *ptr, size_t newSize);`

```
int *p = (int *)calloc(4, sizeof(int));
p = (int *)realloc(p, sizof(int) * 6);

char *str = (char *)calloc(10, sizeof(char));
str = (char *)realloc(str, 20*sizeof(char));
```

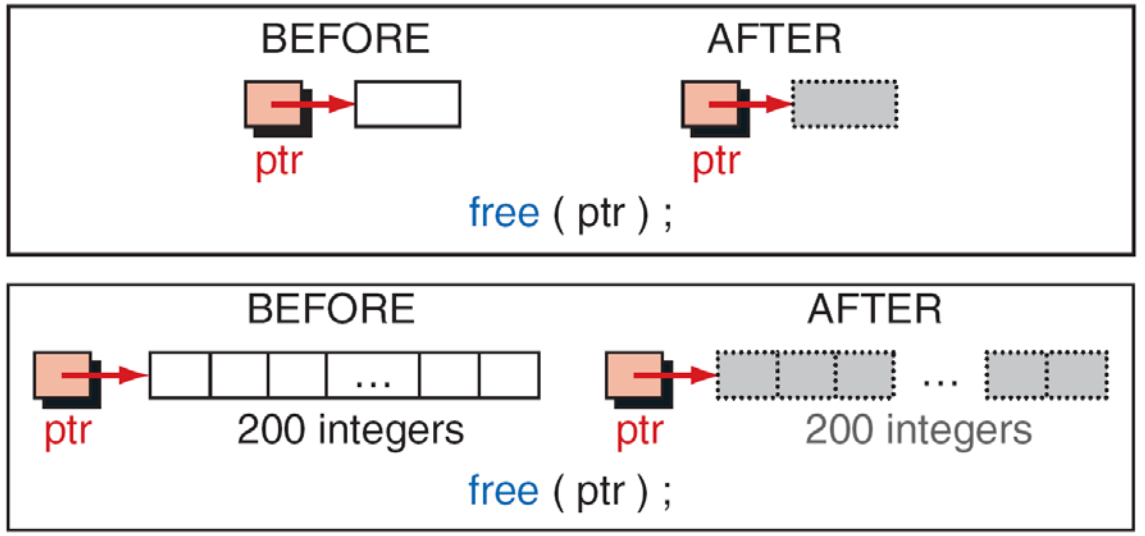


free

- 동적 할당한 메모리 반환
- 사용하지 않을 때 & 프로그램의 마지막에 반드시 반환해야 한다!

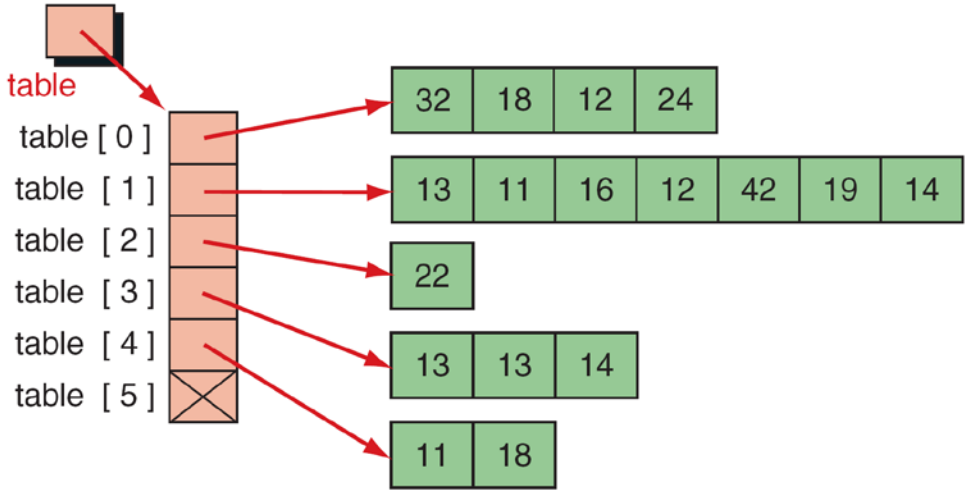
- `void free (void * ptr);`

```
free(ptr);
free(str);
```



동적할당으로 2차원 배열 만들기 – 포인터 배열

```
int **table;
table = (int **)calloc (3, sizeof(int *));
table[0] = (int *)calloc(4, sizeof(int));
…
```



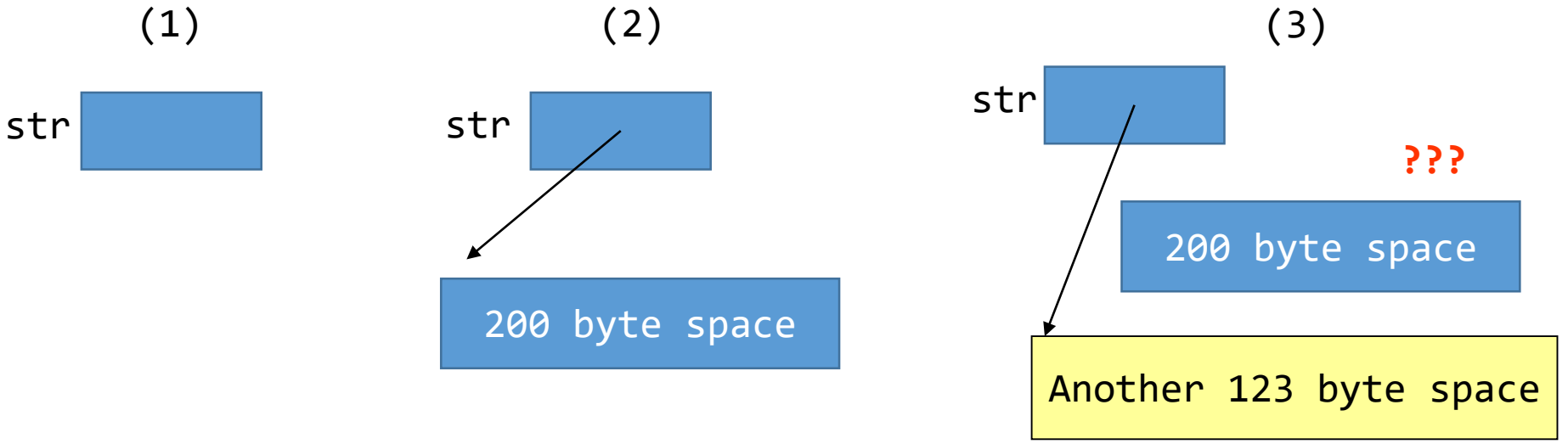

```
table = (int**)calloc (rowNum + 1, sizeof(int*));

table[0] = (int*)calloc (4, sizeof(int));
table[1] = (int*)calloc (7, sizeof(int));
table[2] = (int*)calloc (1, sizeof(int));
table[3] = (int*)calloc (3, sizeof(int));
table[4] = (int*)calloc (2, sizeof(int));
table[5] = NULL;
```

포인터의 부적절한 사용으로 인한 주요 부작용들

- **Unreachable Memory** (이 공간을 가리키는 포인터가 존재하지 않는다)
- **Dangling Pointer** (어딘가 가리키고 있기는 하지만, 그 위치에 뭐가 있는지 알 수 없다.)
- **Buffer Overflow** (할당되어 있는 양 이상으로 메모리를 다루게 되는 경우)
- **Segmentation Fault** (잘못된 주소 접근)

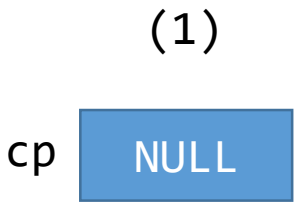
허공에 뜬 메모리 공간 (unreachable memory)

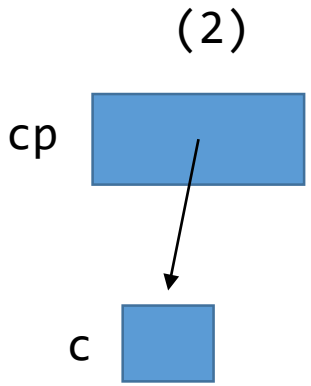
```
{
  char *str;                                    (1)
  str = (char *) malloc(200 * sizeof(char));    (2)
… (no free() !!)
  str = (char *) malloc(123 * sizeof(char));    (3)
}
```

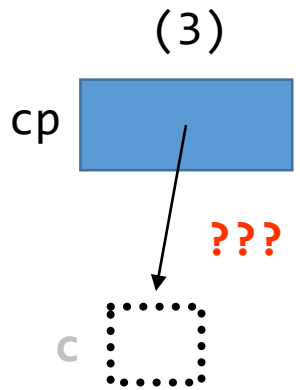


길 잃은 포인터 (dangling pointer)

```
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)
```







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

Buffer overflow

- **버퍼 (buffer)**
  - 컴퓨터 내에서 사용되는 임시 저장소 (메모리 공간)
  - 컴퓨터에서 일어나는 대부분의 작업은 효율을 위해 버퍼를 많이 사용한다.

- **Buffer overflow**
  - Overflow = 넘친다
  - 저장한 내용이 할당되어 있는 버퍼의 영역 바깥으로 넘치는 것
  - 만일 넘친 영역이 다른 용도로 사용되고 있었다면, 그 원래의 값을 파괴해버린다.
  - 넘친 영역의 값이 파괴됨을 이용하여 대상 컴퓨터를 혼란시켜 침입하는 기법이 많이 사용된다.

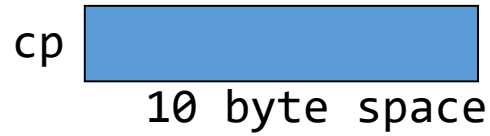
버퍼 오버플로우의 예

```
char cp[10];                    (1)
int a=0x12345678;
strcpy(cp, "Hello, ");          (2)
printf("cp: %s a=%x\n", cp, a);
strcat(cp, "World!");           (3)
printf("cp: %s a=%x\n", cp, a);
```

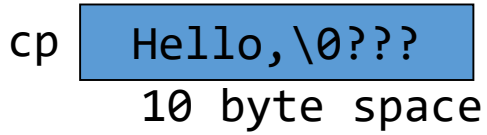
→ 의도한 대로 cp: Hello, a=0x12345678 이 출력됨

→ ??? 오류 or Buffer overflow – a의 값이 바뀜

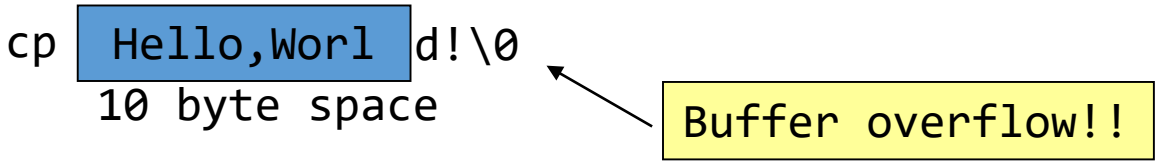
(1)



(2)



(3)



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 유효한지 꼼꼼히 살펴봐야 한다.

- e.g.

```
int *p;              // 초기화 없이 선언
*p = 123;            // 초기화되지 않은 포인터 dereferencing - 여긴 어디 나는 누구?
printf("%d\n", *p);    // ???
```

Problem

- 입력 받은 숫자 크기의 integer array를 할당하고, array에 채울 숫자를 입력 받은 후 출력하고, array의 size를 다시 입력 받아서 array를 재 할당한 후 숫자 입력과 출력을 수행하는 프로그램을 만들자
- `malloc`(혹은 `calloc`)과 `realloc` 함수를 이용한다.
- `find_min_max(int *arr, int size, int *max, int *min)`

- 입력: 최대 999999/ 최소 1
- free를 잊지 말자!

```
Input array size : 5
2 1 5 4 3
Entered numbers are
2 1 5 4 3
Maximum is 5
Minimum is 1

Input changed array size : 8
5 2 3 11 66 32 22 10
Entered numbers are
5 2 3 11 66 32 22 10
Maximum is 66
Minimum is 2
```

Tidy Up

- 동적으로 할당된 메모리는 **반드시 시스템에 반환**해야 함!

→ 그래야 다른 프로그램이 메모리 공간을 활용할 수 있다. (메모리 leak)

- **실행 전**에 미리 할당 vs **실행 중**에 동적으로 할당하고 반환

→ 공간이 **필요한 만큼**만 쓰기 때문에 효율적!

→ 메모리 공간의 크기는 정해져 있다.

- 얼마나 큰 공간이 필요한 지 알 수 없을 때

→ 적당히 크게? - 공간이 부족하거나, 공간이 낭비되거나

다음시간

- Enumeration, Structure, Union
- String (Advanced)