

## Lecture 05

# 포인터 및 배열

# 리뷰



## ■ 무조건 점프

- `goto` 문이 같은 함수 내에서 레이블로 지정하는 위치로 이동하게 함
- `goto` 문을 너무 많이 사용하면 좋지 않음

```
{  
    int i=0, n=20; // 초기화  
    goto loop_cond;  
loop_body:  
    i++;  
loop_cond:  
    if (i<n)  
        goto loop_body;  
}
```

## ■ 입출력 함수

- 입출력 함수들은 `<stdio.h>` 표준 라이브러리에서 제공됨
- 문자 입출력: `putc()`, `getc()`, `putchar()`, `getchar()`, ...
- 문자열 입출력: `puts()`, `gets()`, `fputs()`, `fgets()`, ...
- 형식화된 입출력: `printf()`, `scanf()`, `fprintf()`, `fscanf()`, ...
- 파일 열림/닫힘: `fopen()`, `fclose()`
- 파일 액세스: `feof()`, `fseek()`, `ftell()`, ...
- ...

# 리뷰



- `printf()` 및 `scanf()`

- 형식화된 출력

- ```
int printf(char format[], arg1, arg2, ...)
```

- 출력형식을 정해주는 문자열

- ```
%[flags][width][.precision][modifier]<type>
```

- type: d, i (**int**), u, o, x, X (**unsigned int**), e, E, f, g, G (**double**), c (**char**), s (**string**)
    - flags, width, precision, modifier: 출력된 문자의 개수와 의미를 정함

- 형식화된 입력

- ```
int scanf(char format[], ...)
```

- `printf`와 비슷하지만 매개변수는 포인터임

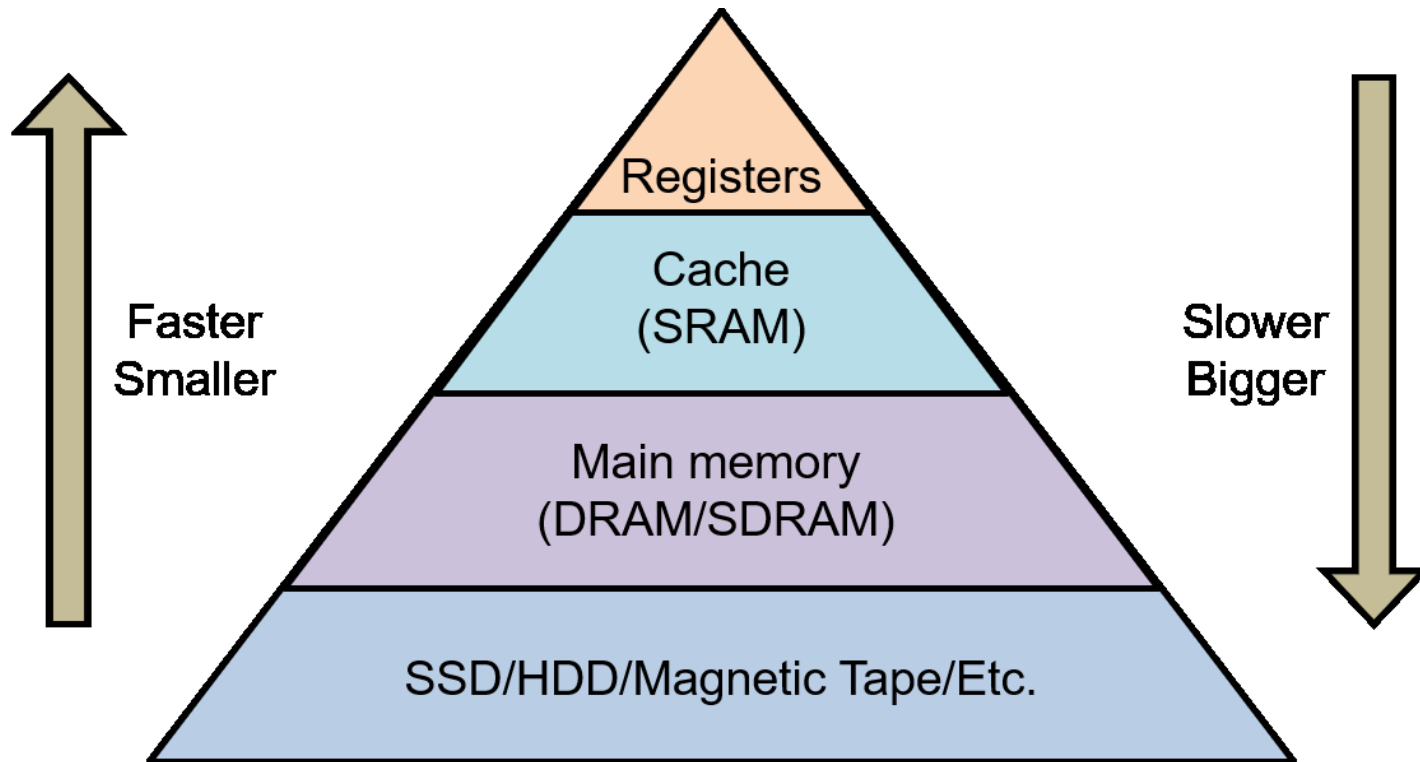
# 리뷰

## ■ 스트링 또는 문자열

- 스트링은 C 언어에서 문자 배열(`char []`)로 선언됨
- 스트링의 끝에는 `NULL` 문자(`'\0'`)가 있어야 함
- 예,  
`char str[] = "I am a string";` 또는  
`char str[20] = "I am a string";`
- `strcpy()`는 스트링 복사 함수임
- 스트링 관련 함수가 많음

# 물리 메모리 및 가상 메모리

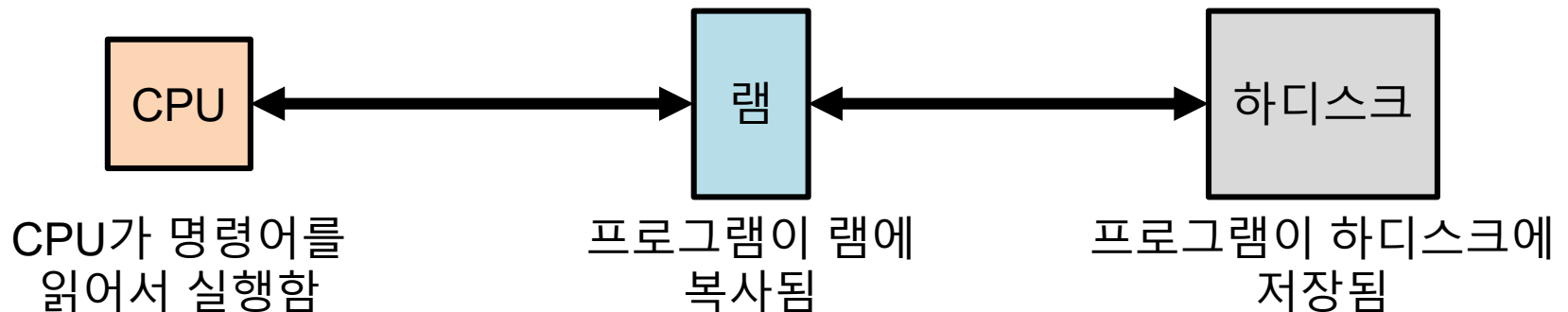
- 메모리 계층



# 물리 메모리 및 가상 메모리

## ■ 메모리 계층

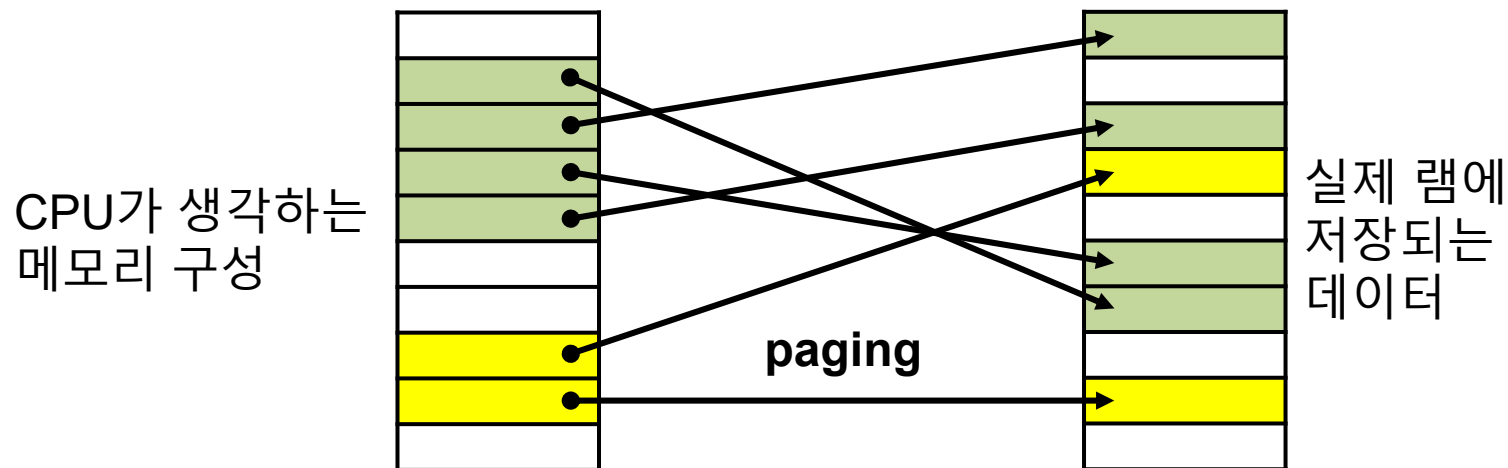
- 레지스터, 캐시, 램: 컴퓨터 프로그램을 저장하기에 적합하지 않음
  - 레지스터, 캐시: 저장 공간 부족
  - 램: 휘발성
- 하드디스크: 컴퓨터 프로그램을 저장하기에 적합하지만 접근 속도가 제일 느림
- 컴퓨터 프로그램 실행



# 물리 메모리 및 가상 메모리

## ■ 메모리 관리 메커니즘

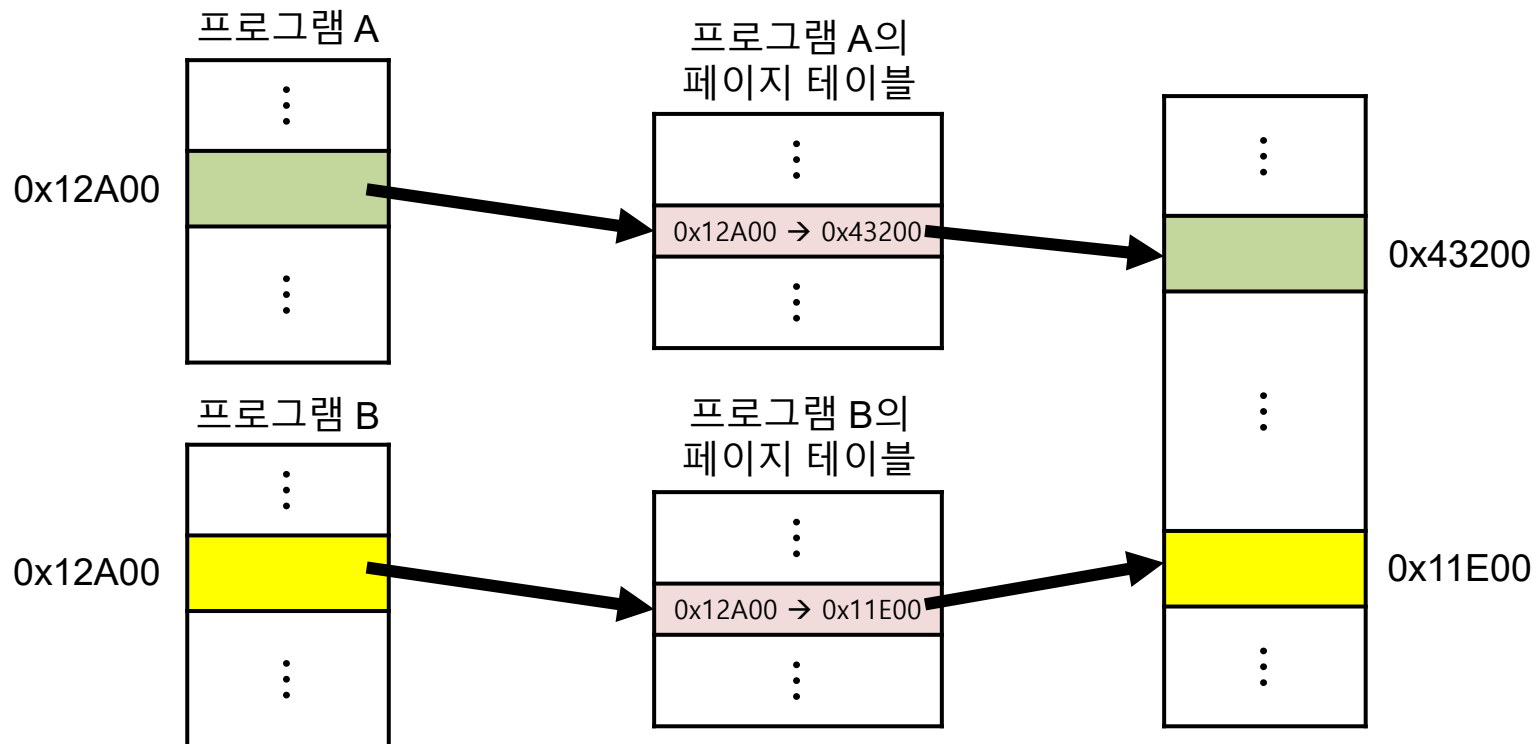
- CPU가 보는 주소값과 실제 메모리의 주소값은 차이가 있음
- 가상 메모리: CPU가 참조하는 메모리 공간
- 물리 메모리: 실제 메모리
- CPU가 참조하는 메모리 주소값은 특별한 1:1 변환 과정(**paging**)에 의해 실제 메모리의 주소값을 변환하게 됨





# 물리 메모리 및 가상 메모리

- 메모리 관리 메커니즘
  - 같은 가상 주소지만 서로 다른 물리 메모리를 참조함



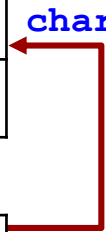
# 물리 메모리 및 가상 메모리

- 메모리 크기
  - 물리 메모리
    - 예, 16 MB(캐시) + 8 GB(램) + 512 GB(SSD) + ...
  - 가상 메모리
    - 가상 메모리 관리를 위해 사용하는 가상 주소 비트에 따름
    - 예,
      - 32비트 시스템에서는 가상 메모리  $< 2^{32} = 4 \text{ GB}$
      - 64비트 시스템에서는 가상 메모리  $< 2^{64} = 16 \text{ EB (exabytes)}$
    - 스택(stack) 및 힙(heap)
      - 스택**은 함수 호출 시 생성되는 지역 변수와 매개 변수가 저장되는 가상 메모리 영역이며, 함수 호출 완료 시 사라짐
      - 힙**은 필요에 의해 동적으로 메모리를 할당할 때 사용됨

# 포인터 및 주소

- 포인터는 어떤 변수의 메모리 번지를 기억하는 것임
  - c가 **char** 형이고 메모리에 0x11 번지에 저장됨
  - c를 가리키는 포인터가 p라고 하며, p의 값은 c가 기록되어 있는 메모리 번지가 됨
  - 즉, p = 0x11
- 메모리 번지를 알아내려면 & 연산자를 사용함
  - 메모리에 있는 대상(변수와 배열)에만 사용할 수 있음
  - 문장이나 상수 등에 사용할 수 없음
- 포인터가 가리키고 있는 변수의 값을 알아내려면 \* 연산자를 사용함

| 주소   | 저장된 값 |                      |
|------|-------|----------------------|
| 0x10 | 10    |                      |
| 0x11 | 'a'   | <b>char</b> c = 'a'; |
| 0x12 | 20    |                      |
| ...  | ...   |                      |
| 0x2a | 0x11  | <b>int</b> *p = &c;  |
| 0x2b | 133   |                      |



# 포인터 및 주소

- 포인터와 메모리 주소

- 운영체제가 들어가 있는 시스템

포인터가 기억하는 메모리 번지가 **가상 메모리**의 주소값임

- 간단한 운영체제가 들어가 있거나 운영체제가 없는 시스템

포인터가 기억하는 메모리 번지가 **물리 메모리**의 주소값임

- 예,

```
int x=1, y=20, z[10];
```

```
int *p; // 정수형 변수의 포인터 선언
```

```
p = &x; // p가 x에 가리킴
```

```
y = *p; // y = 1
```

```
*p = 9; // x = 9
```

```
p = &z[0]; // p가 z[0]에 가리킴
```

# & 연산자

## ■ & 연산자

- 어떤 변수의 메모리 번지를 알아내기 위해 사용됨

```
int n=4;
double f=1.222;
int *pn = &n;
double *pf = &f;
```

## ■ 메모리 번지를 알아낼 수 없는 대상

- 레지스터 변수

```
register int x=10; int *px = &x; // 에러
```

- 상수/프리프로세서(#define)

```
int *p = &1220; // 에러
```

- 수식

```
int *p = &(x*2+10); // 에러
```

## ■ t 데이터형 변수 x의 포인터는 t \* 형 변수임

# \* 연산자

## ■ \* 연산자

- 포인터에 \* 연산을 하면 그 포인터가 가리키는 변수의 값을 알려줌

```
int n=4;
double f=1.222;
int *pn = &n;
double *pf = &f;
printf("n = %d/n", *pn); // n = 4 출력됨
*pf = *pf + *pn; // f = 5.222
```

- x의 포인터가 px라면 \*px는 x 대신으로 사용될 수 있음
- \*와 & 연산자는 산술 연산자보다 우선순위가 높음

```
*pn += 1; // n = 5
*pn = *pn * 2; // n = 10
++*pn; // n = 11, 이것은 (*pn)++ 같음
```

# 포인터형 변환

- 어떤 형의 변수에 대한 포인터를 다른 포인터형으로 변환할 수 있음

```
int n = 4;  
int *pn = &n;  
float *pf = (float *)pn;
```

- pf가 원래 정수형 변수 n을 가리키지만 \*pf의 형이 float임
- 메모리에 저장된 데이터가 변환되지 않지만 그 데이터를 해석하는 방식이 달라짐

```
printf("pn points to %d\n", *pn); // 4  
printf("pf points to %f\n", *pf); // 0.00000
```

- void \* 라는 것이 어떤 형의 포인터도 될 수 있음

# 포인터와 함수의 매개변수

- 값에 의한 호출(call by value)
  - 함수에 인자를 변수에 대입된 값을 던져 주는 것을 의미함

```
void incx(int x) {  
    x++;  
    printf("Inside incx\n");  
    printf("x addr : %p\n", &x);  
    printf("x value : %d\n\n", x);  
}
```

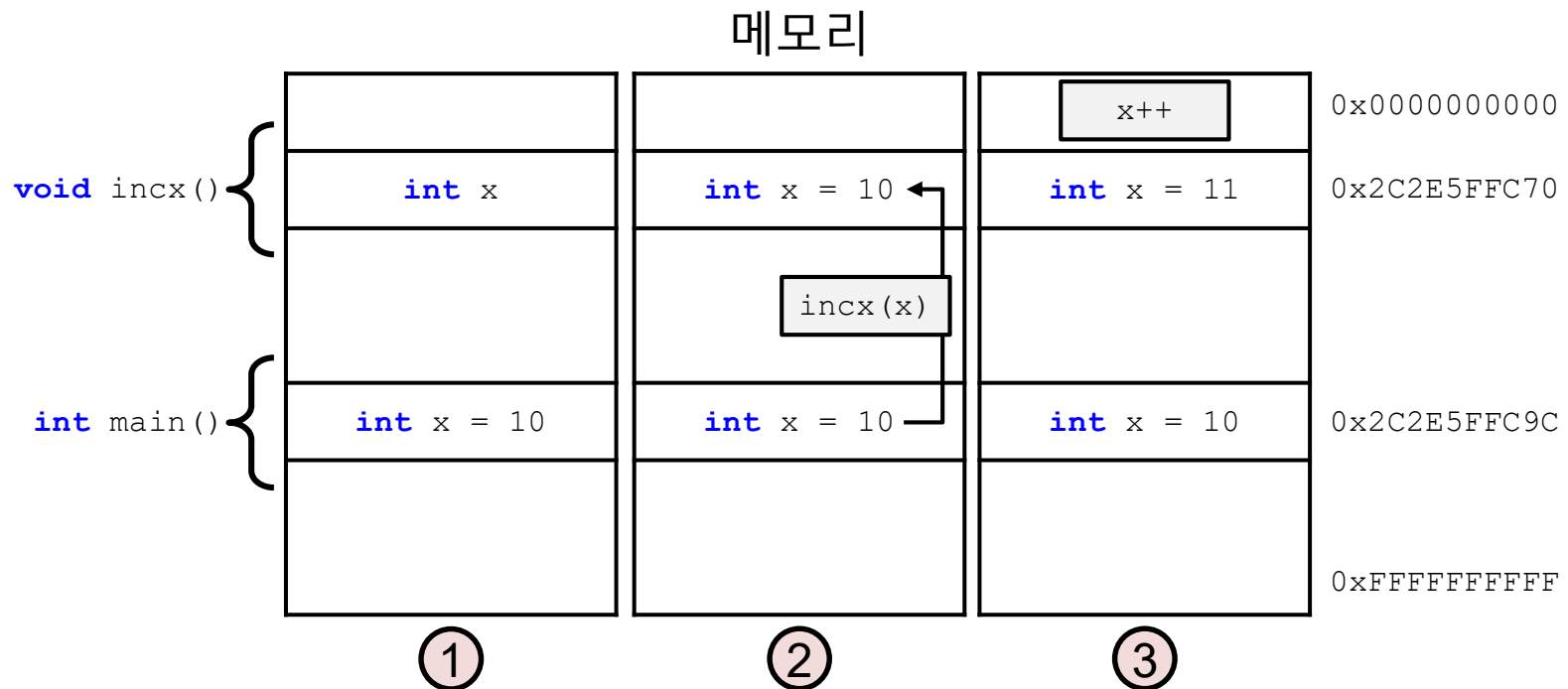
```
int main() {  
    int x = 10;  
    printf("Before incx\n");  
    printf("x addr : %p\n", &x);  
    printf("x value : %d\n\n", x);  
    incx(x);  
    printf("After incx\n");  
    printf("x addr : %p\n", &x);  
    printf("x value : %d\n\n", x);  
    return 0;  
}
```

```
Before incx  
x addr  : 0000002C2E5FFC9C  
x value : 10  
  
Inside incx  
x addr  : 0000002C2E5FFC70  
x value : 11  
  
After incx  
x addr  : 0000002C2E5FFC9C  
x value : 10
```



# 포인터와 함수의 매개변수

- 값에 의한 호출(call by value)
  - 함수에 인자를 변수에 대입된 값을 던져 주는 것을 의미함



# 포인터와 함수의 매개변수

- 참조에 의한 호출(call by reference)
  - 주소값을 참조하고 이를 호출하는 의미함

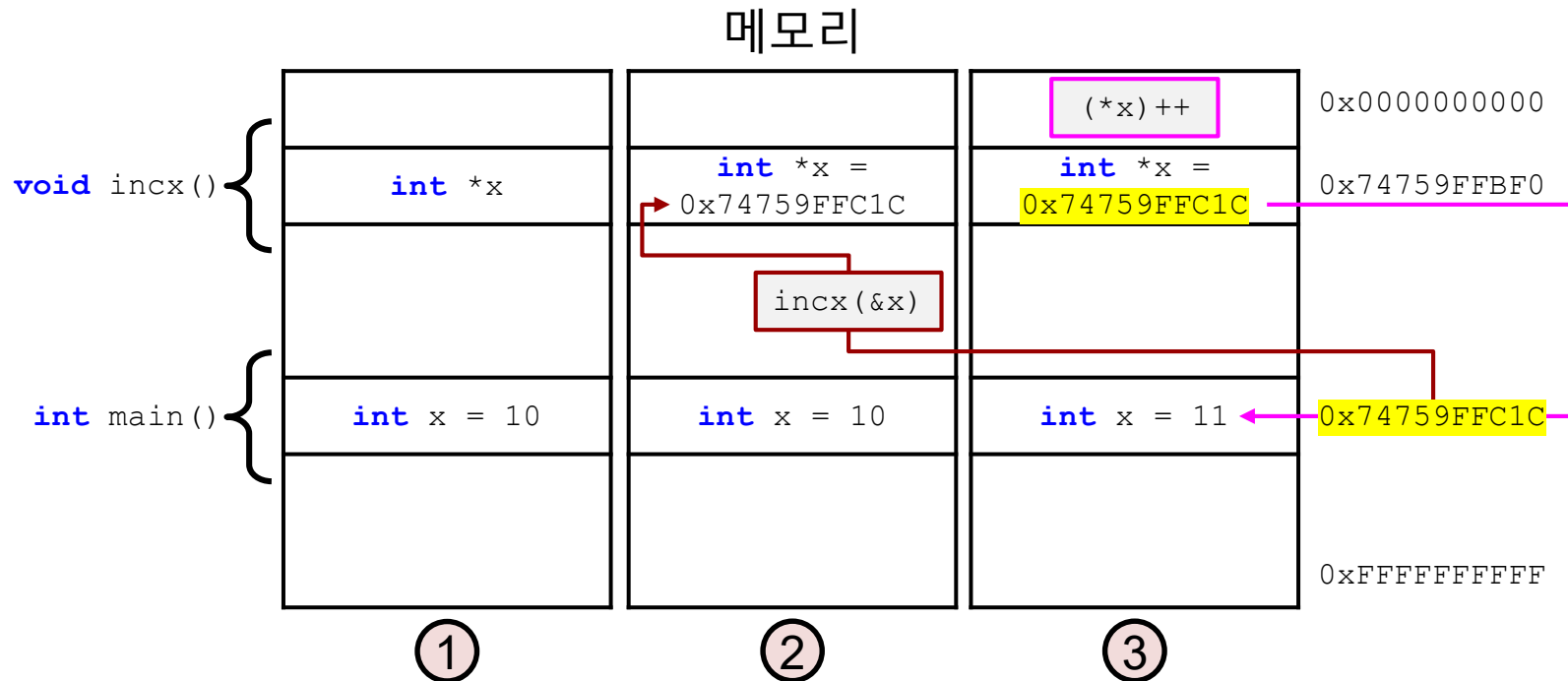
```
void incx(int *x) {  
    (*x)++;  
    printf("Inside incx\n");  
    printf("x addr : %p\n", &x);  
    printf("x value : %p\n", x);  
    printf("x pointer : %d\n\n", *x);  
}
```

```
int main() {  
    int x = 10;  
    printf("Before incx\n");  
    printf("x addr : %p\n", &x);  
    printf("x value : %d\n\n", x);  
    incx(&x);  
    printf("After incx\n");  
    printf("x addr : %p\n", &x);  
    printf("x value : %d\n\n", x);  
    return 0;  
}
```

```
Before incx  
x addr   : 00000074759FFC1C  
x value  : 10  
  
Inside incx  
x addr   : 00000074759FFBF0  
x value  : 00000074759FFC1C  
x pointer : 11  
  
After incx  
x addr   : 00000074759FFC1C  
x value  : 11
```

# 포인터와 함수의 매개변수

- 참조에 의한 호출(call by reference)
  - 주소값을 참조하고 이를 호출하는 의미함



# 포인터와 함수의 매개변수

- 두 변수의 값을 교환하는 함수

- swap(a, b)

```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

temp 지역변수를 사용하지 않으려면  
swap 함수를 어떻게 수정하면 되는가?

- 호출할 때

```
int main() {  
    ...  
    int a=10, b=20;  
    swap(&a, &b);  
    ...  
}
```

# 포인터와 함수의 매개변수

- 수열의 합과 수열의 곱을 계산하기
  - **return** 문이 하나의 값만 내보낼 수 있음
  - 참조에 의한 호출을 통해 여러 개의 값을 내보낼 수 있음

```
void sum_prod(int arr[], int size, int *sum, int *prod) {  
    int k;  
    *sum = 0;  
    *prod = 1;  
    for (k=0; k<size; k++) {  
        *sum += arr[k];  
        *prod *= arr[k];  
    }  
}
```

# 포인터와 함수의 매개변수

- 다음의 코드에서 어떤 문제점이 있는가?

```
char * get_msg() {  
    char msg[] = "I am string";  
    return msg;  
}
```

```
int main() {  
    char *str = get_msg();  
    puts(str);  
    return 0;  
}
```

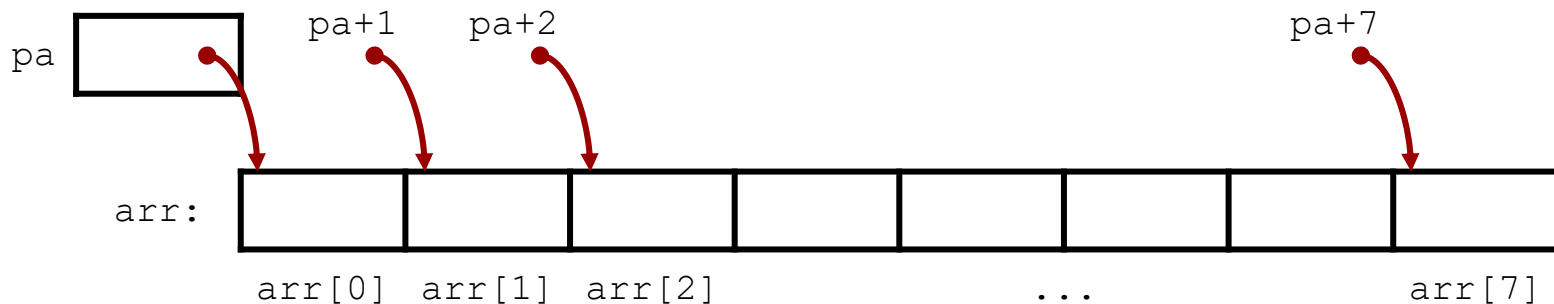
msg는 지역변수이기 때문에 함수에서 벗어나면 유효하지 않음

str는 쓰레기 값을 갖고 있음

# 포인터와 배열

- 배열 선언: `int arr[8];`
- 배열의 첫 요소의 포인터 선언: `int *pa; pa = &arr[0];`

|                      |                   |                     |
|----------------------|-------------------|---------------------|
| <code>*pa</code>     | $\leftrightarrow$ | <code>arr[0]</code> |
| <code>*(pa+1)</code> | $\leftrightarrow$ | <code>arr[1]</code> |
|                      |                   | <code>...</code>    |
| <code>*(pa+i)</code> | $\leftrightarrow$ | <code>arr[i]</code> |



# 포인터와 배열

- 배열형의 변수 값은 0번째 요소의 값이 됨

`pa = &arr[0]; ↔ pa = arr;`

- 유의 사항

- 포인터 `pa`는 일종의 변수이므로 `pa = arr`나 `pa++` 등을 사용 가능
- 배열 이름 `arr`는 변수가 아니므로 `arr = pa`나 `arr++` 등을 사용 **불가능**

- 배열을 사용하는 2개의 방식

- 첨자(인덱스) : `arr[i]`
- 오프셋(offset): `*(pa+i)` 또는 `*(arr+i)`

- 배열을 함수에 넘겨줄 경우 첫 번째 요소의 위치만 넘겨줌

- 첫 번째 요소의 위치가 알게 되면 나머지를 액세스할 수 있음
- 메모리에 대한 절감



# 포인터와 배열

- 문자열의 길이를 계산하기(v1)

```
int strlen(char s[]) {  
    int n;  
    for (n=0; *s != '\0'; s++)  
        n++;  
    return n;  
}
```

- 함수 정의 부분에서: `int strlen(char s[])` ↔ `int strlen(char *s)`
- 배열의 한 부분만을 함수에 전달할 수 있음
- 음수인 인덱스(`arr[-2]` 등)를 사용할 수 있지만 쓰레기 값이 생김

# 포인터와 배열

## ■ 번지 연산

- `p`가 배열의 어떤 요소에 대한 포인터임
- `p++`는 `p`가 다음 요소를 가리키게 함
- `p+=i`는 현재의 요소로부터 `i`번째의 요소를 가리키게 함
- `p`에 어떤 정수를 더하거나 뺄 수 있음

## ■ 문자열의 길이를 계산하기(v2)

```
int strlen(char *s) {  
    char *p = s;  
    for (; *p != '\0'; p++)  
        ;  
    return p-s;  
}
```

# 포인터와 배열

## ■ `sizeof()` 연산자 기반 배열의 길이를 계산하기

- 변수, 데이터형, 구조체, 배열 등의 메모리 크기를 바이트 단위로 알아내줌

```
int arr[10];  
int k = sizeof(arr); // k=40  
int h = sizeof(arr[0]); // h=4  
int len = k/h; // len=10
```

## ■ `sizeof()` 사용 시 **유의사항**

- 배열을 함수에 전달할 경우 배열의 첫 요소의 번지만 전달되므로 함수 내에서 `sizeof()` 연산자를 사용하면 전체의 배열 크기를 알아낼 수 없음

```
void func(int arr[]) {  
    int k = sizeof(arr); // 32-bit OS: k=4, 64-bit OS: k=8  
}
```

- 프리프로세서로 정의하여 사용하는 것을 권장함

```
#define array_len(arr) (sizeof(arr)==0) ? 0 : sizeof(arr)/sizeof((arr)[0])
```

# 문자 포인터와 함수

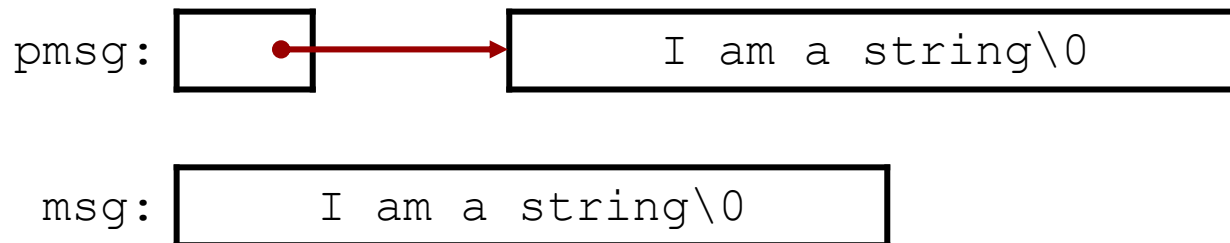
- 문자열은 문자들로 이루어진 배열이며, 항상 NULL 문자 ('\\0')로 끝나게 되어 있음

- 문자열의 2가지 선언 방법

```
char msg[] = "I am a string";
```

```
char *pmsg = "I am a string";
```

- msg는 배열이며, 항상 일정한 장소에 저장됨
- pmsg는 포인터이므로 가리키는 위치를 바꿀 수 있음



# 문자 포인터와 함수

- `string.h` 표준 라이브러리에서 많은 문자열 관련 함수는 제공됨

`strcpy(s, t)` : 문자열 `t`를 문자열 `s`로 복사함

`strcmp(s, t)` : 사전순으로 문자열 `s`와 `t`를 비교함

`strlen(s)` : 문자열 `s`의 길이를 알아냄

`strcat(s, t)` : 문자열 `s`의 끝에 `t`를 붙임

`strchr(s, c)` : 문자 `c`가 문자열 `s`에 있는지 검토함

...

# 문자 포인터와 함수

## ■ 문자열 복사 함수를 살펴보기

```
void strcpy(char *s, char *t) {  
    int k=0;  
    while ((s[k]=t[k]) != '\0')  
        k++;  
}
```

```
void strcpy(char *s, char *t) {  
    while ((*s++=*t++) != '\0')  
        ;  
}
```

```
void strcpy(char *s, char *t) {  
    while ((*s=*t) != '\0') {  
        s++;  
        t++;  
    }  
}
```

```
void strcpy(char *s, char *t) {  
    while (*s++=*t++)  
        ;  
}
```

# 다차원 배열

- C에서 2차원 배열을 다음과 같이 선언할 수 있음

```
int arr[2][4] = {  
    {10, 20, 30, 40},  
    {11, 22, 33, 44},  
};
```

i 번째 행, j 번째 열의 값을 읽어 들이면 `arr[i][j]` 을 사용함

- 2차원 배열을 함수로 전달하려면 열의 수를 반드시 써야함

```
func(int arr[2][4])
```

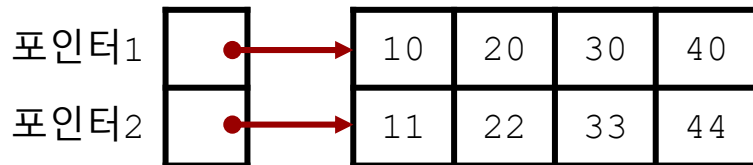
- 각 행이 열의 수로 구성된 배열이므로 포인터 방식을 사용할 수 있음

```
func(int arr[][4]) ↔ func(int (*arr)[4])
```

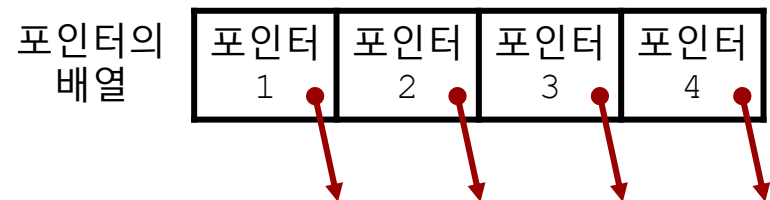
# 다차원 배열

- 포인터 방식을 사용할 때 유의사항
  - 꺾쇠 괄호([])가 \* 연산자보다 우선순위가 높음
    - `int (*arr)[4];`  
4개 정수들의 배열을 가리키는 포인터는 선언됨
    - `int *arr[4];`  
4개 정수의 포인터로 이루어진 배열이 선언됨

`int (*arr)[4];`



`int *arr[4];`





# 다차원 배열

## ■ 다차원 배열과 포인터

- `int arr[2][4];`

행 2개와 열 4개로 이루어진 2차원 배열이 선언됨

**행의 길이가 동일해 야 함**

→ 총 8개의 정수가 한곳에 모여 저장됨

- `int *arr[2];`

2개 정수의 포인터로 이루어진 배열이 선언됨

2차원 배열로 생각될 수 있으며, **행의 길이가 달라도 됨**

각 포인터가 다른 정수 값이나 정수 배열에 가리킬 수 있음

→ 각 포인터가 4개 정수의 배열에 가리키고 있다면, 위와 차이가 있음

- ① 각각 4개 배열이 한곳에 모여 저장되지만, 총 8개가 한곳에 모여 저장된다는 보장이 없음
- ② 총 8개의 정수와 2개의 포인터도 저장되므로 위의 2차원 배열보다 더 많은 메모리가 소요됨

# 다차원 배열

- 어떤 달의 며칠이 1년의 몇 번째 날인가를 계산하고 그 역으로 계산하기

```
static char daytab[2][13] = {
    {0,31,28,31,30,31,30,31,31,30,31,30,31},
    {0,31,29,31,30,31,30,31,31,30,31,30,31},
};

int day_of_year(int year, int month, int day) { // 월별 날짜 → 연별 날짜
    int i, leap;
    leap = (year%4 == 0) && (year%100 != 0) || (year%400 == 0);
    for (i=1; i<month; i++)
        day += daytab[leap][i];
    return day;
}
```

# 다차원 배열

- 어떤 달의 며칠이 1년의 몇 번째 날인가를 계산하고 그 역으로 계산하기

// 년, 연별 날짜 → 달, 월별 날짜

```
void month_day(int year, int yearday, int *pmonth, int *pday) {  
    int i, leap;  
    leap = (year%4 == 0) && (year%100 != 0) || (year%400 == 0);  
    for (i=1; yearday > daytab[leap][i]; i++)  
        yearday -= daytab[leap][i];  
    *pmonth = i;  
    *pday = yearday;  
}
```

# 명령 라인 매개변수

- C의 `main` 함수가 실행될 때 2개의 매개변수가 전달됨
  - **`argc`** : 프로그램을 실행하기 위한 매개변수의 개수
  - **`argv`** : 매개변수들의 모임인 문자열을 가리키는 포인터
- `argv[0]` 은 프로그램의 이름이므로 `argc`는 최소 1이 됨
- 예, 명령 라인 매개변수를 그대로 출력하는 프로그램

```
int main(int argc, char *argv[]) {  
    int k;  
    for (k=1; k<argc; k++)  
        printf("%s ", argv[k]);  
    printf("\n");  
    return 0;  
}
```

# 탐색(searching) 및 정렬(sorting)

## ■ 어떤 값이 배열에서 있는지 탐색하기

```
int linear_search(int arr[], int len, int val) {  
    int k;  
    for (k=0; k<len; k++) {  
        if (arr[k] == val)  
            return k;  
    }  
    return -1;  
}
```

배열 인덱스  
사용

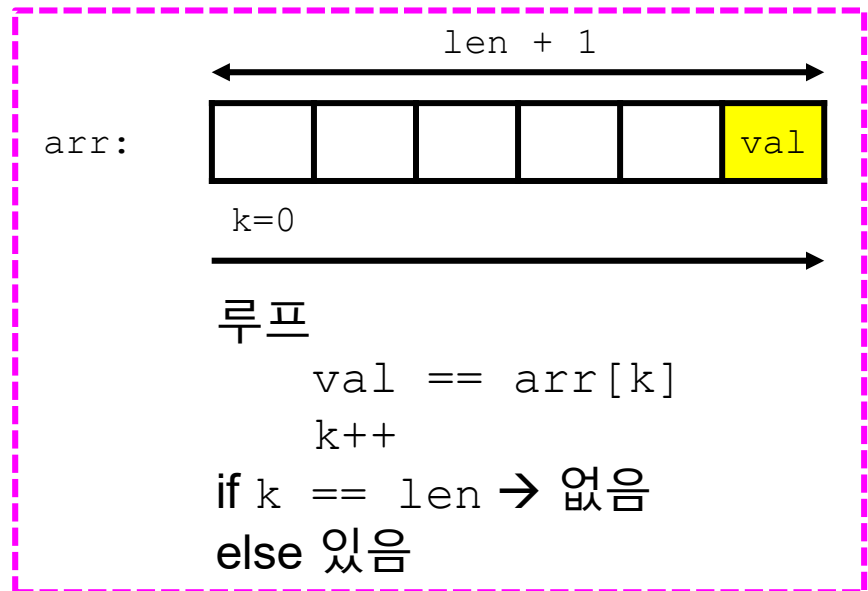
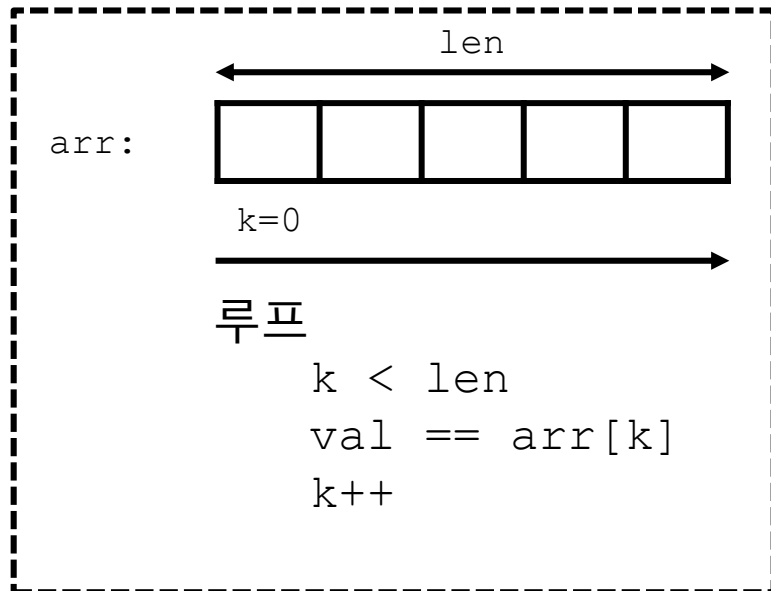
```
int * linear_search(int arr[], int len, int val) {  
    int *p;  
    for (p=arr; p<(arr+len); p++) {  
        if (*p == val)  
            return p;  
    }  
    return NULL;  
}
```

포인터 사용

# 탐색(searching) 및 정렬(sorting)

- 어떤 값이 배열에서 있는지 탐색하기
  - 루프를 돌리면서 2개의 비교를 함
    - `val == arr[k]` : 구하고 싶은 `val` 값이 배열의 요소와 같은 지 비교
    - `k < len` : 배열에 벗어나갈 지 비교

## 개선 버전



# 탐색(searching) 및 정렬(sorting)

- 어떤 값이 배열에서 있는지 탐색하기(개선 버전)

```
// 이제 arr 배열의 끝에서 val 값이 있음
int linear_search(int arr[], int len, int val) {
    int k=-1;
    while (arr[++k] != val)
        ;
    if (k==len)
        return -1;
    else
        return k;
}
```

- 배열이 정렬되어 있으면 탐색이 더 빠름

# 삽입 정렬(insertion sort)



## ■ 삽입 정렬

- ① 두 번째 요소부터 시작함
- ② 그 앞(왼쪽)의 요소들과 비교함
- ③ 삽입할 위치를 지정함
- ④ 비교했던 요소들을 뒤로 옮김
- ⑤ 지정한 위치에 삽입함

```
void isort(int arr[], int len) {  
    int k;  
    for (k=1; k<len; k++)  
        if (arr[k]<arr[k-1])  
            shiftel(arr, k);  
}
```

```
void shiftel(int arr[], int k) {  
    int kval = arr[k];  
    for (; k && arr[k-1]<kval; k--)  
        arr[k] = arr[k-1];  
    arr[k] = kval;  
}
```



# 퀵 정렬(quick sort)

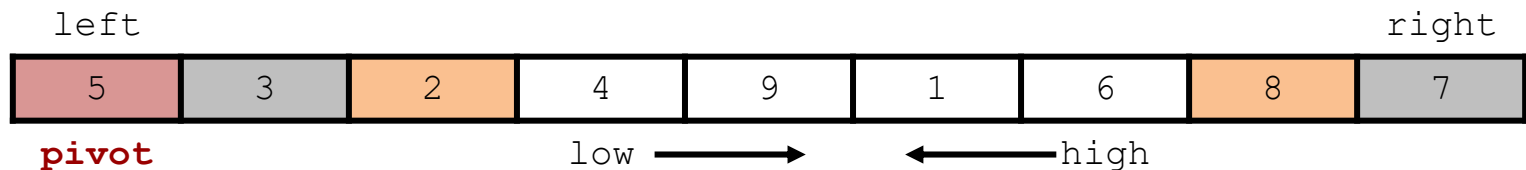
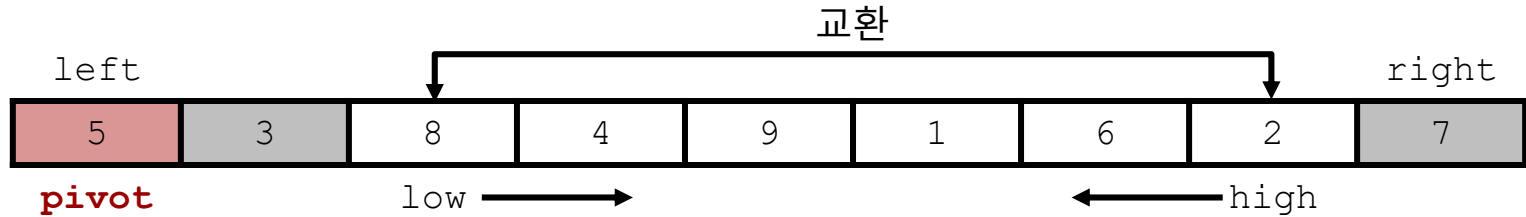
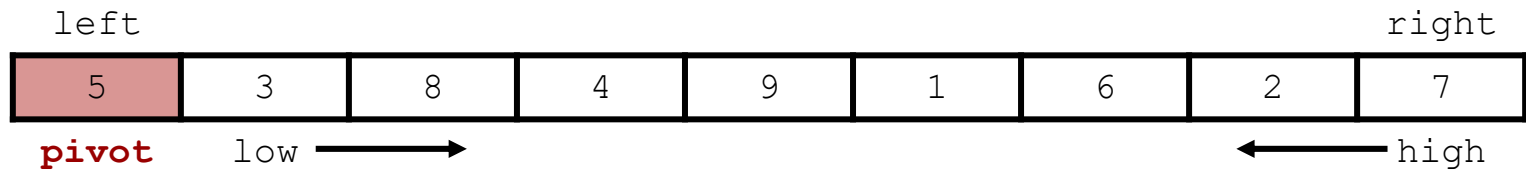
- 퀵 정렬

- ① 한 요소를 **피벗(pivot)**으로 선택함
- ② 피벗을 기준으로 피벗보다 작은 요소들은 모두 피벗의 왼쪽으로, 피벗보다 큰 요소들은 모두 피벗의 오른쪽으로 옮겨짐
- ③ 피벗을 제외한 왼쪽 배열과 오른쪽 배열을 다시 정렬함
- ④ 배열을 더 이상 분할할 수 없을 때까지 반복함

- `stdlib.h` 표준 라이브러리에서 `qsort()` 함수가 제공됨

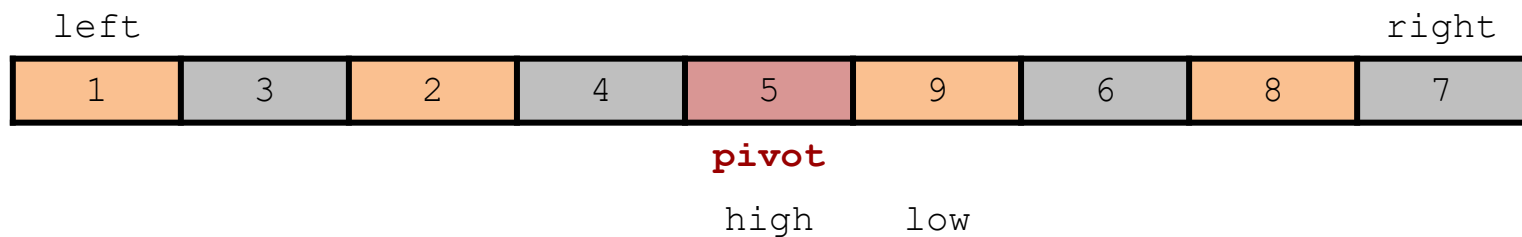
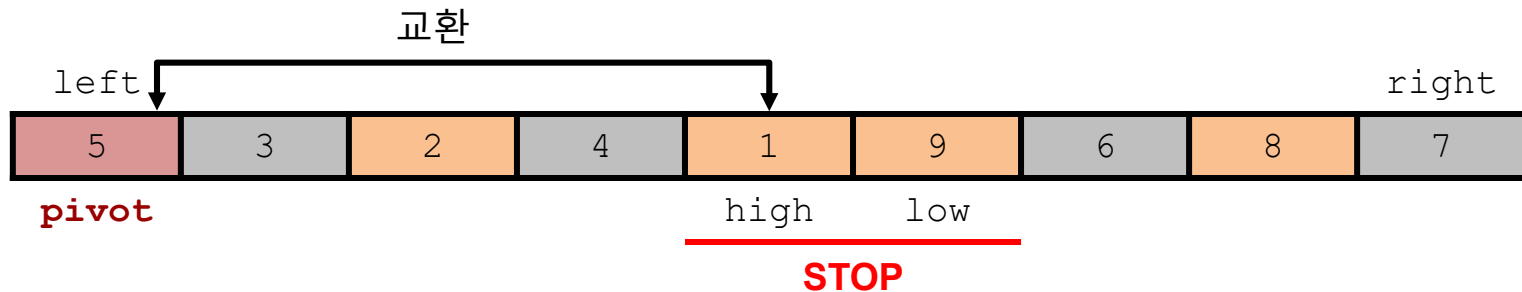
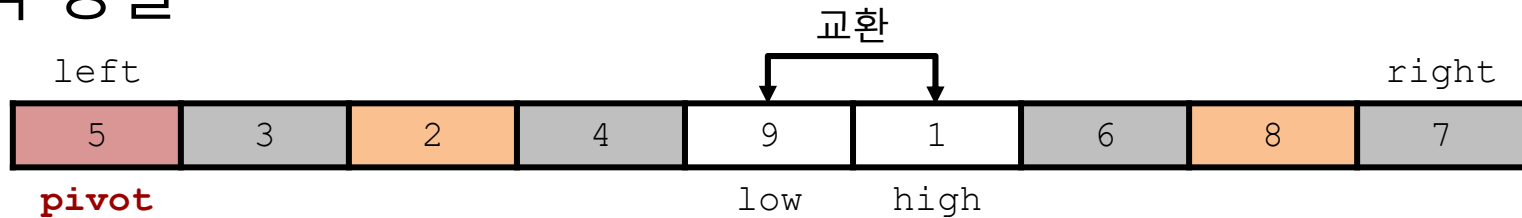
# 퀵 정렬(quick sort)

## ■ 퀵 정렬



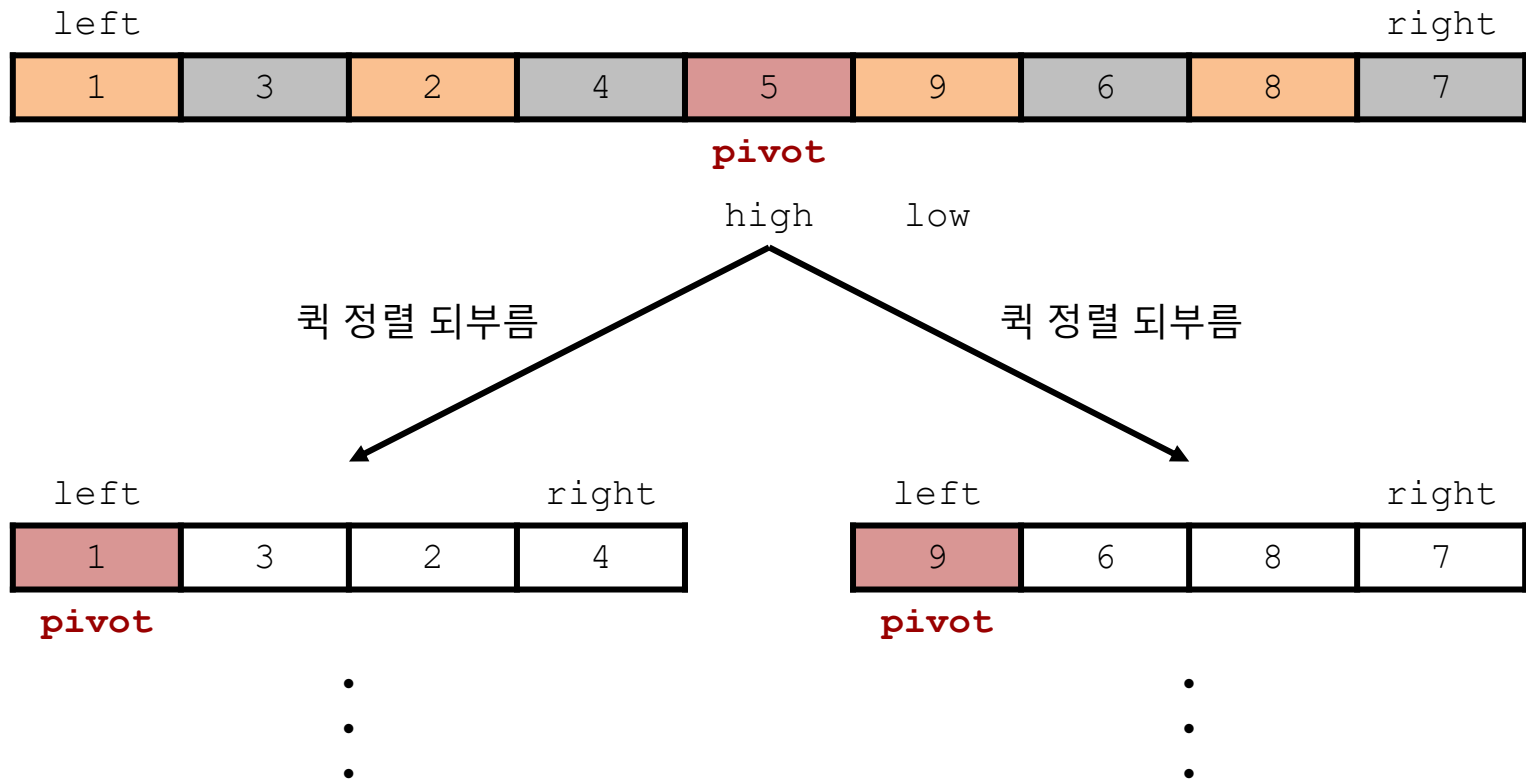
# 퀵 정렬(quick sort)

## ■ 퀵 정렬



# 퀵 정렬(quick sort)

## ■ 퀵 정렬



# 퀵 정렬(quick sort)

## ■ 퀵 정렬

```
void swap(int *a, int *b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}
```

```
void quicksort(int arr[], int left, int right) {  
    if (left < right) {  
        int pi = partition(arr, left, right);  
        quicksort(arr, left, pi-1);  
        quicksort(arr, pi+1, right);  
    }  
}
```

```
int partition(int arr[], int left, int right) {  
    int pivot = arr[left];  
    int low = left+1;  
    int high = right;  
    while (low <= high) {  
        while (arr[low] <= pivot && low <= right) low++;  
        while (arr[high] > pivot && high >= left) high--;  
        if (low < high)  
            swap(&arr[low], &arr[high]);  
    }  
    swap(&arr[left], &arr[high]);  
    return high;  
}
```

# 이분탐색(binary search)

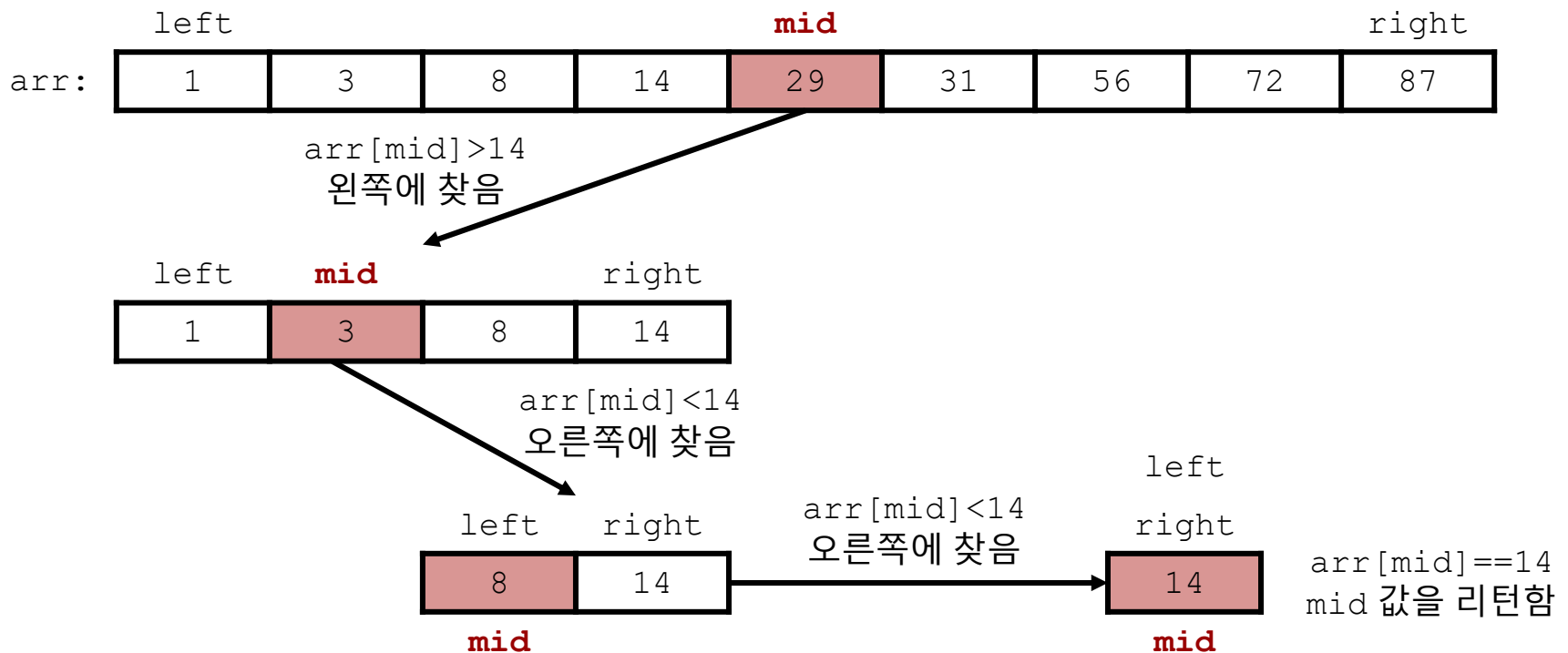


## ■ 이분탐색

- ① 입력 배열이 이미 정렬되어 야 함
- ② 중간 위치의 요소를 찾고 싶은 요소와 비교함
  - 같다면 검색이 종료됨
  - 작다면 왼쪽 요소들을 다시 검사함
  - 크다면 오른쪽 요소들을 다시 검사함
- ③ 찾게 되거나 더 이상 분할될 수 없을 때까지 반복함

# 이분탐색(binary search)

- 예, 아래의 배열에서 14를 찾기



# 이분탐색(binary search)



## ■ 이분탐색

```
int binsearch(int arr[], int len, int val) {  
    int left=0, right=len-1, mid;  
    while (left <= right) {  
        mid = (left+right)/2;  
        if (arr[mid] > val)  
            right = mid-1;  
        else if (arr[mid] < val)  
            left = mid+1;  
        else  
            return mid;  
    }  
    return -1;  
}
```

## ■ stdlib.h 표준 라이브러리에서 bsearch() 가 제공됨



# 함수의 포인터

- 함수는 변수가 아니지만 함수의 포인터는 정의할 수 있음
  - 함수의 포인터를 선언하고 지정하기

```
int add(int a, int b) {  
    return a+b;  
}  
  
int main() {  
    int (*pf)(int, int);  
    pf = &add;  
    printf("%d\n", pf(10,5)); // 15  
    return 0;  
}
```

# 함수의 포인터

- 함수는 변수가 아니지만 함수의 포인터는 정의할 수 있음
  - 함수의 포인터를 다른 함수로 전달하기

```
int add(int a, int b) {  
    return a+b;  
}  
  
int sub(int a, int b) {  
    return a-b;  
}  
  
int calc(int a, int b, int (*op)(int, int)) {  
    return op(a, b);  
}  
  
int main() {  
    printf("%d\n", calc(10,5,add)); // 15  
    printf("%d\n", calc(10,5,sub)); // 5  
    return 0;  
}
```

# 함수의 포인터

- 함수는 변수가 아니지만 함수의 포인터는 정의할 수 있음
  - 함수의 포인터로 이루어진 배열을 사용하기

```
int add(int a, int b) {
    return a+b;
}

int sub(int a, int b) {
    return a-b;
}

int mul(int a, int b) {
    return a*b;
}

int div(int a, int b) {
    if (b!=0)
        return a/b;
    else
        return -1;
}

int main() {
    int (*pfarr[])(int, int) = {add,sub,mul,div};
    int x=10, y=5;
    printf("Add: %d\n", pfarr[0](x,y));
    printf("Sub: %d\n", pfarr[1](x,y));
    printf("Mul: %d\n", pfarr[2](x,y));
    printf("Div: %d\n", pfarr[3](x,y));
    return 0;
}
```