



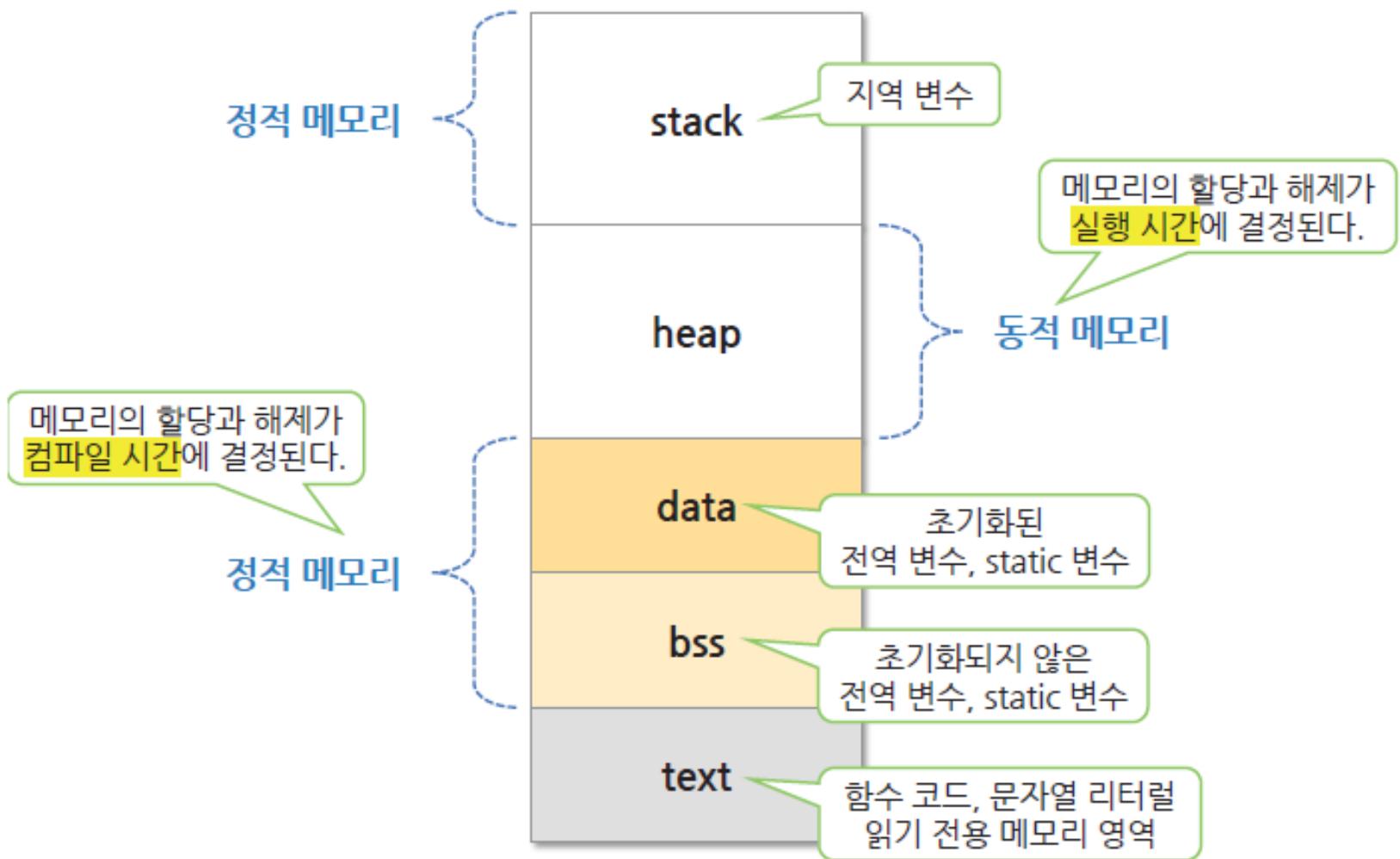
C 프로그래밍 및 실습

C Programming

이지민

# CHAP 11 동적메모리와 함수포인터

# 동적메모리의 개념(1/2)



# 동적메모리의 개념(2/2)

특징	정적 메모리	동적 메모리
메모리 할당	컴파일 시간에 이루어진다.	실행 시간에 이루어진다.
메모리 해제	자동으로 해제된다.	명시적으로 해제해야 한다.
사용 범위	지역 변수는 선언된 블록 내, 전역변수는 프로그램 전체에서 사용할 수 있다.	프로그래머가 원하는 동안만큼 사용할 수 있다.
메모리 관리	컴파일러의 책임이다.	프로그래머의 책임이다.

# 동적메모리의 필요성(1/2)

## ❖ 정수값을 입력받아서 합계를 구하는 프로그램

- **방법1:** 정수를 몇 개나 입력할지 사용자에게 물어보고, 사용자가 입력한 개수만큼 int 배열을 할당하는 경우

```
int size;  
  
printf("정수의 개수? ");  
  
scanf("%d", &size);  
  
int arr[size];
```

배열의 크기를 변수  
로 지정할 수 없다.

- **방법2:** 배열의 최대 크기를 가정해서 배열을 할당하는 경우

```
int arr[1000];
```

메모리가 낭비되므로 비효율적이고,  
버퍼 오버런이 발생할 수 있다.

## 동적메모리의 필요성(2/2)

- ❖ 동적 메모리를 사용하면 방법1과 방법2의 문제점을 모두 해결할 수 있다.
- ❖ 동적 메모리는 꼭 필요한 만큼 크기를 지정해서 메모리를 할당할 수 있다.
- ❖ 동적 메모리는 메모리의 할당과 해제 시점을 프로그래머가 마음대로 정할 수 있다.
- ❖ 동적 메모리는 메모리 사용에 있어서 프로그래머에게 최대 한의 자유를 보장하는 기능이다

# 동적메모리의 할당(1/3)

❖ 동적 메모리를 할당하려면 `malloc` 함수를 사용한다.

```
void* malloc(size_t size);
```

- `size` 바이트만큼 동적 메모리를 할당하고 할당된 메모리의 주소 (`void*`형)를 리턴한다.
- `void*`형의 포인터 : ‘어떤 형인지 알 수 없는 메모리를 가리키는 주소’라는 뜻
- 동적 메모리를 할당할 수 없으면 `NULL`을 리턴한다.

❖ `malloc` 함수가 할당하는 메모리를 어떻게 사용할지 프로그래머가 마음대로 정할 수 있다.

- `malloc` 함수가 리턴한 주소를 특정 포인터형으로 형 변환해서 포인터 변수에 저장하고 사용해야 한다.

# 동적메모리의 할당(2/3)

동적 메모리의 주소를 저장할 포인터를 준비한다.

```
int size;  
int *arr = NULL;  
  
scanf("%d", &size);  
arr = (int*) malloc(sizeof(int)*size);
```

동적 메모리의  
바이트 크기

동적 메모리를 어떤  
용도로 사용할지에  
따라 형 변환한다.

void\*형

malloc 함수는 할당할 메모리의  
데이터형을 지정하지 않는다.

arr  
int 주소

어떤 포인터로 접근하  
는지에 따라 메모리의  
용도가 결정된다.

arr를 이용해서 int 배열  
인 것처럼 사용한다.

sizeof(int)×size바이트

동적  
메모리

# 동적메모리의 할당(3/3)

## ❖ 동적 메모리는 반드시 포인터로 접근해야 한다.

- 동적 메모리를 사용하려면 포인터를 준비하고, 이 포인터에 malloc 함수의 리턴값을 저장해야 한다.

```
arr = (int*)malloc(sizeof(int) * size);
if (arr == NULL) {
    printf("동적 메모리 할당 실패\n");
    return -1;
}
```

동적 메모리 할당이 실패했는지 확인해야 한다.

메모리 할당 실패 시 프로그램 비정상 종료

## ❖ 동적 메모리를 가리키는 포인터는 배열처럼 사용할 수 있다.

```
for (i = 0; i < size; i++)
    scanf("%d", &arr[i]);
```

# 동적메모리의 해제

❖ 동적 메모리를 해제할 때는 `free` 함수를 사용한다.

```
void free(void* memblock);
```

- *memblock*이 가리키는 동적 메모리를 해제한다.

❖ 동적 메모리를 해제한 다음에는 동적 메모리를 가리키던 포인터를 널 포인터로 만드는 것이 안전하다.

```
free(arr);
```

```
arr = NULL;
```

❖ 동적 메모리는 사용이 끝나면 반드시 명시적으로 해제해야 한다.

# 동적메모리의 이용(1/3)

```
01 #define _CRT_SECURE_NO_WARNINGS  
02 #include <stdio.h>  
03 #include <stdlib.h>  
04  
05 int main(void)  
06 {  
07     int size;  
08     int* arr = NULL;  
09     int i, sum;  
10  
11     printf("정수의 개수? ");  
12     scanf("%d", &size);  
13     arr = (int*) malloc(sizeof(int) * size);
```

malloc, free 함수  
사용 시 필요하다.

동적 메모리의 주소를  
저장할 포인터를 준비한다.

동적 메모리를 할당하고 그  
주소를 arr에 저장한다.

# 동적메모리의 이용(2/3)

```
14 if (arr == NULL) {  
15     printf("동적 메모리 할당 실패\n");  
16     return -1;  
17 }  
18  
19 printf("%d개의 정수를 입력하세요: ", size);  
20 for (i = 0; i < size; i++)  
21     scanf("%d", &arr[i]);  
22 for (i = 0, sum = 0; i < size; i++)  
23     sum += arr[i];  
24 printf("입력된 정수의 합계: %d\n", sum);
```

동적 메모리 할당이 실패했는지 확인한다.

동적 메모리를 가리키는 포인터를 배열처럼 사용할 수 있다.

# 동적메모리의 이용(3/3)

```
26     free(arr);  
27     arr = NULL;  
28 }
```

사용이 끝나면 동적 메모리를 해제하고, 포인터를 널 포인터로 만든다.

## 실행 결과

정수의 개수? 5

5개의 정수를 입력하세요: 123 42 35 61 3

입력된 정수의 합계: 264

# 동적메모리의 사용순서(1/2)

- ① 동적 메모리의 주소를 저장할 포인터를 준비한다.
  - 동적 메모리의 용도에 따라 데이터형을 정하고, 널 포인터로 초기화 한다.

```
int* arr = NULL;
```

- ① 동적 메모리를 할당할 때는 malloc 함수를 사용한다.
  - 할당할 메모리의 바이트 크기를 지정하고 리턴값을 형 변환해서 포인터에 저장한다.

```
arr = (int*)malloc(sizeof(int) * size);
```

- ① 동적 메모리를 사용할 때는 배열의 원소를 가리키는 포인터처럼 사용한다.

```
for (i = 0; i < size; i++)  
    scanf("%d", &arr[i]);
```

# 동적메모리의 사용순서(2/2)

- ④ 동적 메모리는 사용이 끝나면 free 함수로 해제한다.
  - 해제된 메모리에 접근하지 않도록 포인터를 널 포인터로 만든다.

```
free(arr);
```

```
arr = NULL;
```

# 동적메모리의 사용 시 주의사항(1/3)

## ❖ 해제된 동적 메모리를 사용해서는 안된다.

- 해제된 메모리에 접근하면 실행 에러가 발생한다.

```
int* p = (int*)malloc(sizeof(int) * 100);  
: // p가 가리키는 동적 메모리를 사용한다.  
free(p); // p가 가리키는 동적 메모리를 해제한다.  
🚫 *p = 123; // 해제된 메모리에 접근한다.(실행 에러)
```

- 허상 포인터(dangling pointer)

- 잘못된 주소가 들어있는 포인터
- 실행 에러의 원인이 된다.

- 동적 메모리 해제 후에는 포인터를 널 포인터로 만든다.

```
free(p); // 사용이 끝난 동적 메모리를 해제한다. (p가 허상 포인터가 된다.)  
p = NULL; // p를 사용하지 못하도록 널 포인터로 만든다.  
🚫 *p = 123; // 메모리 0번지에 접근하므로 프로그램이 죽는다.
```

# 동적메모리의 사용 시 주의사항(2/3)

## ❖ 해제된 동적 메모리를 다시 해제해서는 안된다.

- 해제된 동적 메모리를 다시 해제하려고 하면 실행 에러가 발생한다.

```
int* p = (int*)malloc(sizeof(int) * 100);
:           // p가 가리키는 동적 메모리를 사용한다.
free(p);    // 사용이 끝난 동적 메모리를 해제한다. (p는 허상 포인터가 된다.)
:
```

🚫 **free(p);** // 실수로 해제된 메모리를 다시 해제하면 실행 에러가 발생한다

- **free** 함수 호출 후 널 포인터로 만든다.

```
free(p);    // 사용이 끝난 동적 메모리를 해제한다.
p = NULL;   // p를 널 포인터로 만든다.
free(p);    // free 함수는 매개변수가 널 포인터면 아무것도 하지 않고 리턴한다.
```

# 동적메모리의 사용 시 주의사항(3/3)

- ❖ free 함수는 동적 메모리를 해제할 때만 사용해야 한다.

int x;	
int* p = &x;	// p는 지역 변수를 가리킨다.
🚫 free(p);	// 지역 변수를 가리키는 p로 free 함수를 호출하면 실행 에러가 발생한다.

- ❖ 동적 메모리의 주소를 잃어버리지 않도록 주의해야 한다.

```
int* test_memory(int cnt)
{
    int i;
    int* p = (int*)malloc(sizeof(int) * cnt);
    for (i = 0; i < cnt; i++)
        p[i] = 0;
}
```

함수가 리턴될 때 지역 변수 p가 소멸된다.  
(동적 메모리의 주소를 잃어버리게 된다.)

# 동적메모리에 문자열 할당하기(1/2)

- ❖ 문자열을 처리할 때 동적 메모리를 사용하면 꼭 필요한 만큼 메모리를 할당하고 사용할 수 있다.
- ❖ 두 문자열을 결합하여 새로운 문자열을 만들어서 리턴하는 `join_string` 함수의 정의
  - 동적 메모리에 문자열을 생성하기 때문에  $s_1$ 과  $s_2$ 의 길이에 관계없이 언제든지  $s_1$ 과  $s_2$ 를 연결한 문자열을 새로 생성할 수 있다.
  - 문자열을 생성할 수 없으면 NULL 리턴
  - `join_string`이 리턴하는 문자열의 주소를 `char*`형 변수에 저장하고 사용한다.
  - 사용이 끝나면 `free` 함수로 해제해야 한다.

# 동적메모리에 문자열 할당하기 | (2/2)

동적 메모리의 주소를  
저장할 포인터를 준비한다.

```
char* ptr = NULL;  
ptr = join_string("water", "melon");
```

동적 메모리의 주소를  
리턴한다.

입력 매개변수이므로 변경되지 않는다.

동적 메모리의 주소  
이므로 사용이 끝난  
후 해제해야 한다.

ptr  
char 주소

문자열의 길이는 5

문자열의 길이는 5

동적  
메모리

sizeof(char) \* (5 + 5 + 1)바이트

# join\_string함수의 정의 및 사용(1/2)

```
06 #define MAX_STR 128
```

```
09 char* join_string(const char* s1, const char* s2)
```

```
10 {
```

```
11     int len = 0;
```

```
12     char* p = NULL;
```

동적 메모리의 주소를  
저장할 포인터 변수

문자열을 결합할 수  
없으면 NULL 리턴

```
13
```

```
14     if (s1 == NULL || s2 == NULL)
```

```
15         return NULL;
```

결합할 문자열의 길이

```
16
```

```
17     len = strlen(s1) + strlen(s2) + 1;
```

동적 메모리를 할당하고  
동적 메모리에 결합된  
문자열을 저장한다.

```
18
```

```
19     p = (char*)malloc(sizeof(char)*len);
```

```
20     strcpy(p, s1);
```

```
21     strcat(p, s2);
```

```
22     return p;
```

동적 메모리의 주소를 리턴한다.  
p가 가리키는 동적 메모리는 함수를  
호출한 곳에서 해제해야 한다.

# join\_string함수의 정의 및 사용(2/2)

```
23 int main(void)
24 {
25     char s1[MAX_STR] = "";
26     char s2[MAX_STR] = "";
27     char* s3 = NULL;
28
29     printf("첫 번째 문자열? ");
30     gets_s(s1, sizeof(s1));
31     printf("두 번째 문자열? ");
32     gets_s(s2, sizeof(s2));
33     s3 = join_string(s1, s2);
34     if (s3 != NULL)
35         printf("연결된 문자열: %s\n", s3);
36     free(s3);
37     s3 = NULL;
38 }
```

## 실행 결과

```
첫 번째 문자열? water
두 번째 문자열? melon
연결된 문자열: watermelon
```

동적 메모리의 주소를  
리턴값으로 받아온다.

s3가 가리키는 동적 메모리는 사용 후 해제해야 한다.

# 동적메모리에 구조체 할당하기

## ❖ 구조체 하나를 할당하는 경우

- 먼저 구조체 포인터를 준비한다.
- 동적 메모리를 할당하고 그 주소를 구조체 포인터에 저장한다.
- 포인터로 동적 메모리에 할당된 구조체에 접근할 때는 간접 멤버 접근 연산자(->)를 사용한다.

```
struct content* p = NULL;  
p = (struct content*) malloc(sizeof(struct content));  
strcpy(p->title, "Avengers");  
:  
free(p);  
p = NULL;
```

# 동적메모리에 구조체 여러 개 할당하기 | (1/4)

- ❖ 구조체 배열을 할당하면 메모리가 낭비된다.
  - 구조체가 몇 개 사용될지 알 수 없는데 미리 할당하기 때문
- ❖ 포인터 배열만 준비해두고 구조체는 필요할 때마다 하나씩 동적 메모리에 할당하는 것이 더 효율적이다.

```
struct content arr[100];
```

content 구조체를 100개  
메모리에 할당한다.

```
struct content* arr[100] = { NULL };
```

포인터(주소)만 100개  
메모리에 할당한다.

# 동적메모리에 구조체 여러 개 할당하기 | (2/4)

- ❖ 필요한 시점에 구조체 하나를 동적 메모리에 할당하고, 그 주소만 포인터 배열의 원소로 저장하고 사용한다.

```
arr[cnt] = (struct content*)malloc(sizeof(struct content));
```

- 포인터 배열과 동적 메모리를 사용하면 구조체를 꼭 필요한 개수만큼 할당할 수 있다.
- ❖ 동적 메모리에 할당된 구조체를 사용하려면, 포인터 배열에 보관해둔 주소를 이용한다.
    - 포인터 배열의 원소가 구조체 포인터이므로 -> 연산자를 이용한다.

arr[i]가 구조체 포인터이므로  
멤버 접근 시 ->를 이용한다.

```
printf("%s %d %.1f\n", arr[i]->title, arr[i]->price, arr[i]->rate);
```

# 동적메모리에 구조체 여러 개 할당하기 | (3/4)

- ❖ 동적 메모리의 사용이 끝나면, 포인터 배열의 원소를 이용해서 동적 메모리 각각을 해제해야 한다.
- ❖ 동적 메모리 해제는 동적 메모리 할당과 1:1로 대응된다.

```
for (i = 0; i < cnt; i++) {  
    free(arr[i]);  
    arr[i] = NULL;  
}
```

동적 메모리를 cnt번 할  
당했으면 cnt번 해제해  
야 한다.

# 동적메모리에 구조체 여러 개 할당하기 | (4/4)

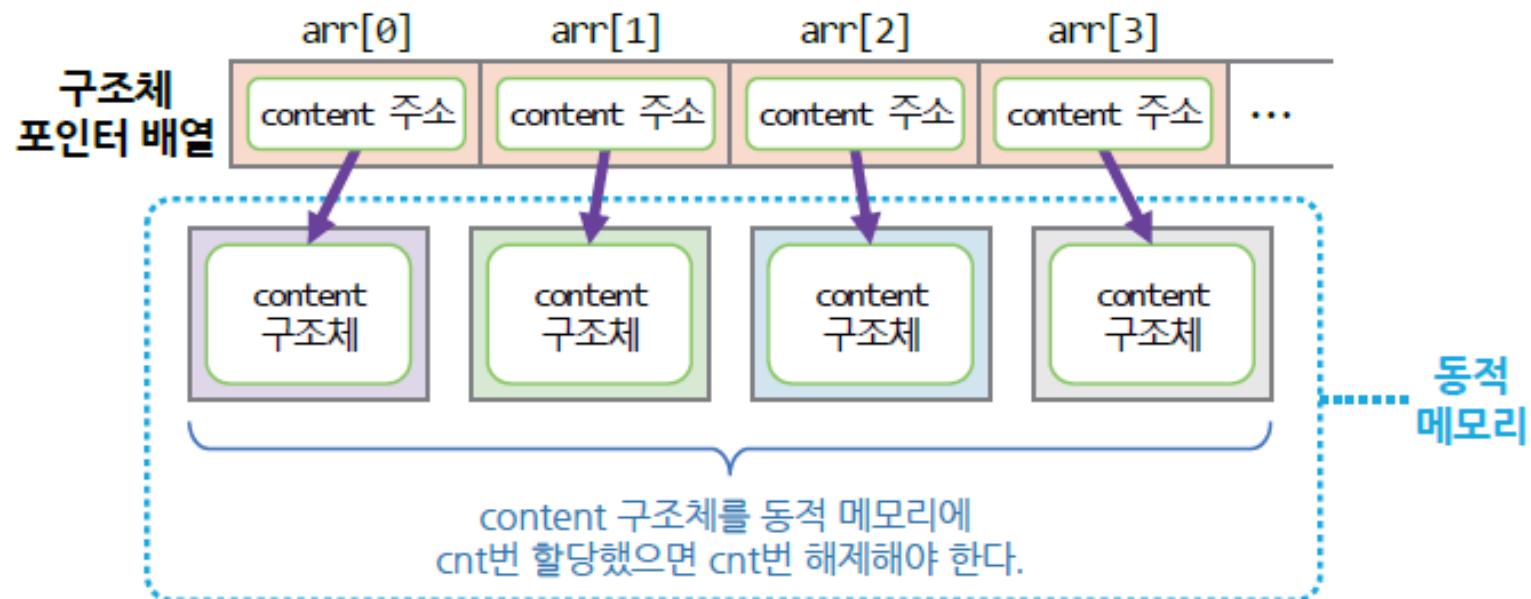
배열의 원소가 content 구조체 포인터이다.

포인터만 100개 할당한다.

```
struct content* arr[100] = { NULL };
int cnt = 0;
:
arr[cnt] = (struct content*)malloc(sizeof(struct content));
```

실제로 할당된 content 구조체의 개수

동적 메모리에 할당된 content 구조체의 주소를 arr[cnt]에 저장한다.



# 구조체 포인터 배열과 동적메모리(1/4)

```
06 #define MAX 100           ----- 최대 콘텐츠의 개수
07 #define STR_SIZE 40
08
09 struct content {
10     char    title[STR_SIZE];
11     int     price;
12     double   rate;
13 };
14
15 int main(void)
16 {
17     struct content* arr[MAX] = { NULL };      ----- 구조체 포인터 배열
18     int cnt = 0;                                ----- 실제로 할당된 content
19     int i;                                     ----- 구조체의 개수
20 }
```

# 구조체 포인터 배열과 동적메모리(2/4)

```
21     while (cnt < MAX) {  
22         char title[STR_SIZE] = "";  
23         printf("콘텐츠를 등록합니다.(. 입력 시 종료)\n제목? ");  
24         gets_s(title, sizeof(title));  
25         if (strcmp(title, ".") == 0)           content 구조체 하나를 동적  
26             break;                           메모리에 할당하고, 그 주소를  
27  
28         arr[cnt] = (struct content*)malloc(sizeof(struct content));  
29         strcpy(arr[cnt]->title, title);  
30         printf("가격? ");  
31         scanf("%d", &arr[cnt]->price);  
32         arr[cnt]->rate = 5.0;  
33         cnt++;                            할당된 구조체의 개수를  
34         while (getchar() != '\n') {}        증가시킨다.  
35     }
```

# 구조체 포인터 배열과 동적메모리(3/4)

```
36     printf("제목           가격   평점\n");
37     for (i = 0; i < cnt; i++) {
38         printf("%-20s %5d %4.1f\n", arr[i]->title, arr[i]->price, arr[i]->rate);
39     }
40
41     for (i = 0; i < cnt; i++) {
42         free(arr[i]);
43         arr[i] = NULL;
44     }
45 }
```

content 구조체 목록을 모두 출력한다.

동적 메모리를 cnt번 할당했으면 cnt번 해제해야 한다.

# 구조체 포인터 배열과 동적메모리(4/4)

## 실행 결과

콘텐츠를 등록합니다.(. 입력 시 종료)

제목? Avengers

가격? 11000

콘텐츠를 등록합니다.(. 입력 시 종료)

제목? Aquaman

가격? 5500

콘텐츠를 등록합니다.(. 입력 시 종료)

제목? Shazam!

가격? 7700

콘텐츠를 등록합니다.(. 입력 시 종료)

제목? .

제목	가격	평점
----	----	----

Avengers	11000	5.0
----------	-------	-----

Aquaman	5500	5.0
---------	------	-----

Shazam!	7700	5.0
---------	------	-----

# 동적메모리 관리함수

❖ 동적 메모리 관리 함수를 사용하려면 `<stdlib.h>`가 필요하다.

함수 원형	기능
<code>void* malloc(size_t size);</code>	동적 메모리를 <i>size</i> 바이트만큼 할당한다.
<code>void free(void* ptr);</code>	<i>ptr</i> 이 가리키는 동적 메모리를 해제한다.
<code>void* calloc(size_t num, size_t size);</code>	동적 메모리 배열을 <i>num</i> × <i>size</i> 바이트만큼 할당하고 0으로 초기화한다.
<code>void *realloc(void *ptr, size_t new_size);</code>	<i>ptr</i> 이 가리키는 동적 메모리의 크기를 <i>new_size</i> 로 변경해서 재할당한다.

# Quiz

1. 다음 중 동적 메모리를 사용해야 하는 이유를 모두 고르시오.

- ① 메모리를 원하는 크기만큼 할당할 수 있기 때문에
- ② 메모리를 원하는 시점에 할당할 수 있기 때문에
- ③ 메모리를 원하는 시점에 해제할 수 있기 때문에
- ④ 메모리가 자동으로 해제되기 때문에

2. 동적 메모리를 할당하는 malloc 함수의 매개변수로는 어떤 값을 전달하는가?

- ① 할당할 메모리의 데이터형
- ② 할당할 메모리의 주소
- ③ 할당할 메모리의 초기값
- ④ 할당할 메모리의 바이트 크기

3. malloc 함수로 할당한 동적 메모리를 접근하려면 어떻게 해야 하는가?

- ① void\*형의 포인터를 이용한다.
- ② 사용하고자 하는 용도에 맞는 포인터를 이용한다.
- ③ 사용하고자 하는 데이터형의 변수로 복사해서 이용한다.

4. 구조체를 한꺼번에 배열로 할당하는 대신 필요할 때마다 하나씩 동적 메모리에 할당하고 그 주소만 저장하려면 무엇이 필요한가?

- ① 구조체 배열
- ② 구조체 포인터
- ③ 구조체 포인터 배열
- ④ void 포인터

# 함수 포인터

- ❖ 함수 포인터는 함수의 주소를 저장하는 포인터이다.
- ❖ 함수도 컴파일 및 링크 후에 메모리의 특정 번지에 할당된다.
- ❖ 함수 코드는 텍스트 세그먼트라는 읽기 전용 메모리 영역에 할당된다.

# 함수 포인터의 선언(1/2)

❖ 함수 포인터가 가리킬 함수의 원형이 필요하다.

- 함수 포인터가 가리킬 함수의 원형과 함수 포인터의 데이터형이 같아야 하기 때문

형식

리턴형 (\*포인터명)(매개변수목록);

사용예

```
int(*pf)(int) = get_factorial;  
void(*pprint)(const struct point*) = NULL;
```

함수의 원형

int get\_factorial(int num);

리턴형은  
그대로 써준다.

함수 포인터의 선언

int (\*pf)(int);

매개변수 목록도  
그대로 써준다.

매개변수 이름은  
생략할 수 있다.

\*와 포인터 변수명은  
( )로 묶어준다.

## 함수 포인터의 선언(2/2)

- \*와 포인터 변수명을 반드시 ()로 묶어주어야 한다.
  - ()를 생략하면 포인터형을 리턴하는 함수 선언문이 된다.

```
int *pf(int); // int*형을 리턴하는 pf 함수의 선언문(pf는 함수 이름)
```

- 함수 포인터가 어떤 함수를 가리킬지 아직 알 수 없으면 널 포인터로 초기화한다.

```
int(*pf)(int) = NULL;
```

- 함수의 주소를 구하려면 함수의 이름 앞에 &를 적어주거나, 함수 이름만 사용할 수 있다.

```
int(*pf)(int) = &get_factorial;
```

```
pf = get_factorial;
```

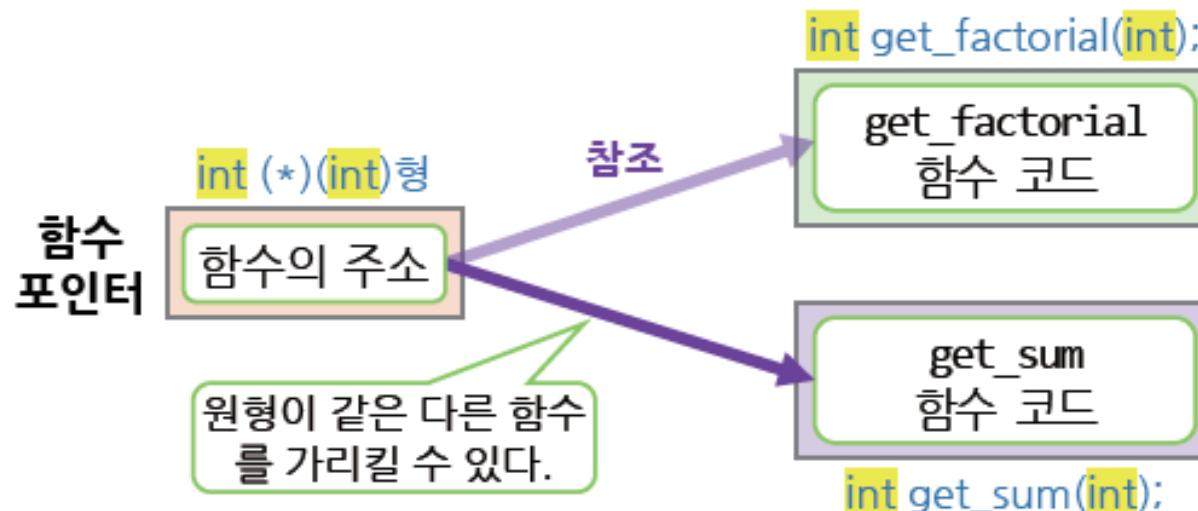
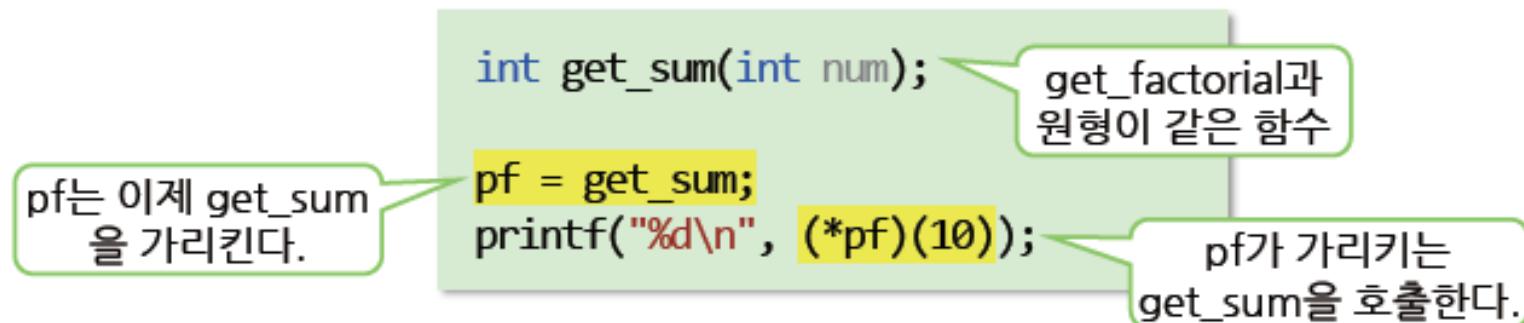
# 함수 포인터의 사용(1/3)

- ❖ 함수 포인터가 가리키는 함수를 호출하려면 역참조 연산자를 이용한다.
  - \*와 함수 포인터를 ()로 묶어주어야 한다.
- ❖ 역참조 연산자 없이 함수 포인터로 직접 함수를 호출할 수도 있다.
  - 함수 포인터를 함수 이름인 것처럼 사용할 수 있다.

```
printf("%d\n", (*pf)(5));  
printf("%d\n", pf(5));
```

# 함수 포인터의 사용(2/3)

- ❖ 함수 포인터도 변수이므로 값을 변경할 수 있다.
  - 함수 포인터가 가리키는 함수는 원형이 같아야 한다.



# 함수 포인터의 사용(3/3)

- ❖ 함수 포인터에 원형이 다른 함수의 주소를 저장하면 컴파일 경고가 발생한다.
  - 경고를 무시하고 함수를 호출하면 실행 에러가 발생한다.

```
int get_max(int x, int y);
```

원형이 다르다는 컴파일  
경고가 발생한다.

🚫 pf = get\_max;

🚫 printf("%d\n", (\*pf)(10));

원형이 다른 함수 호출 시  
실행 에러가 발생한다.

- ❖ 함수 포인터도 사용 전에 널 포인터인지 검사하고 사용하는  
것이 안전하다.

```
if (pf)
```

```
printf("%d\n", (*pf)(10));
```

사용 전에 NULL 포인터인  
지 검사한다.

# 함수 포인터의 선언 및 사용(1/2)

```
03 int get_factorial(int num)
04 {
05     int i;
06     int result = 1;
07
08     for (i = 1; i <= num; i++)
09         result *= i;
10
11     return result;
12 }
13
14 int get_sum(int num)
15 {
16     return num * (num + 1) / 2;
17 }
```

두 함수의 원형이 같다.

# 함수 포인터의 선언 및 사용(2/2)

```
18 int main(void)
19 {
20     int(*pf)(int) = get_factorial;
21
22     if (pf)
23         printf("%d ! = %d\n", 10, (*pf)(10));
24
25     pf = get_sum;
26     if (pf)
27         printf("%d까지 합계 = %d\n", 10, pf(10));
28 }
```

pf는 이제 get\_sum  
함수를 가리킨다.

---- pf는 get\_factorial 함수를 가리킨다.

pf가 가리키는 함  
수를 호출한다.

pf가 가리키는 함  
수를 호출한다.

## 실행 결과

```
10 ! = 3628800
10까지 합계 = 55
```

# 함수 포인터형의 정의

❖ 함수 포인터형을 정의하고, 함수 포인터형의 변수를 선언할 수도 있다.

형식

`typedef 리턴형 (*포인터형명)(매개변수목록);`

사용예

```
typedef int(*funcptr_t)(int);
typedef void(*pprint_t)(const struct point*);
```

함수 포인터형의 변수로  
가리킬 함수의 원형

`int get_factorial(int num);`

함수 포인터형의 정의

`typedef int (*funcptr_t)(int);`

리턴형은 그  
대로 써준다.

매개변수 목록도  
그대로 써준다.

매개변수 이름은  
생략할 수 있다.

\*와 함수 포인터형명은  
( )로 묶어준다.

funcptr\_t는 데이터형 이름이다.

# 함수 포인터형의 사용

- ❖ 함수 포인터형의 변수는 함수 포인터가 된다.

\*가 필요 없다.

```
funcptr_t pf = NULL;
```

- ❖ 함수 포인터형을 정의해두면 함수 포인터 변수를 선언하는 것이 더 간단하다.
- ❖ 원형이 같은 함수에 대한 포인터를 여러 번 선언해야 할 때는 함수 포인터형을 정의하고 사용하는 것이 좋다.

```
pf = get_factorial;
```

// pf는 get\_factorial 함수를 가리킨다.

```
printf("%d\n", (*pf)(10));
```

// pf가 가리키는 함수를 호출한다.

# 함수 포인터 배열(1/3)

## ❖ 배열의 원소가 함수 포인터인 배열

- 함수 포인터형을 정의해서 이용하는 경우

```
typedef void(*pfunc_t)(const char*);  
pfunc_t arr[3] = { NULL };
```

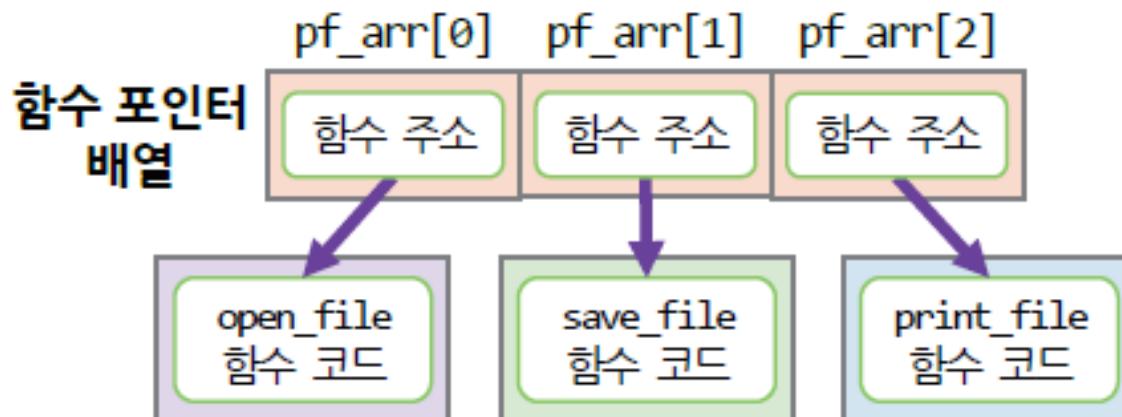
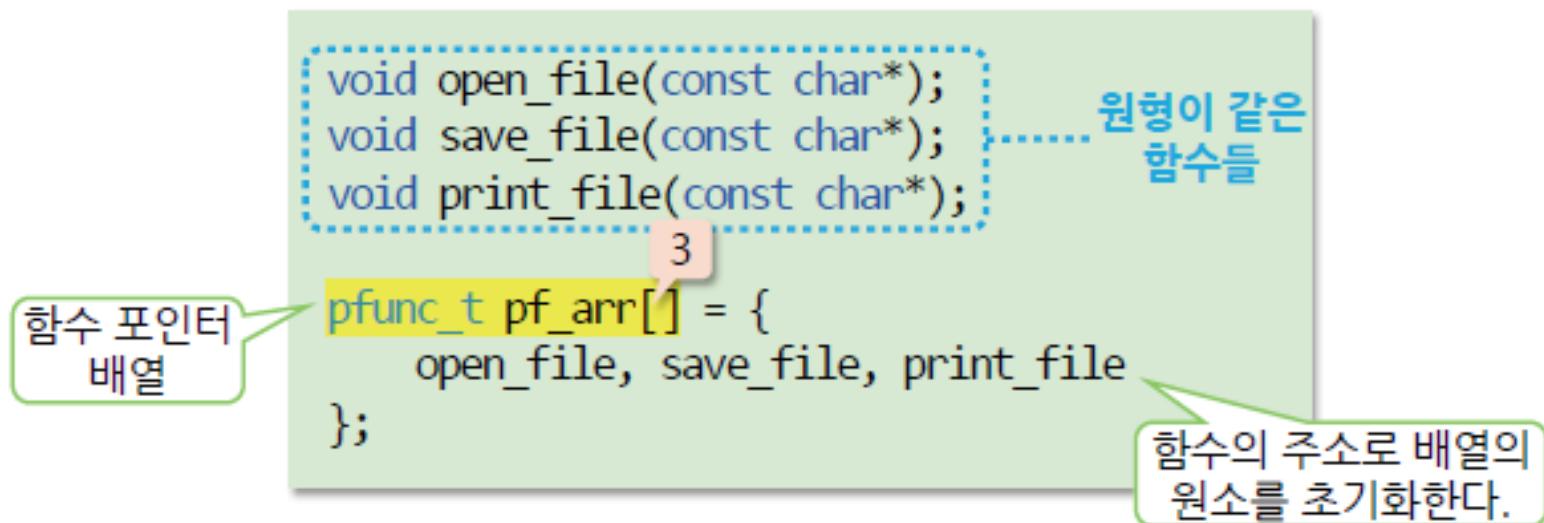
함수 포인터형

- 함수 포인터형을 이용하지 않고 직접 선언하는 경우

```
void (*arr[3])(const char*) = { NULL };
```

# 함수 포인터 배열(2/3)

## ❖ 함수 포인터 배열의 메모리 구조



# 함수 포인터 배열(3/3)

- ❖ 함수 포인터 배열을 이용하면 원형이 같은 함수들을 모아서 관리할 수 있다.

## switch문을 이용하는 경우

```
switch (selected) {  
case 0:  
    open_file("test.doc"); break;  
case 1:  
    save_file("test.doc"); break;  
case 2:  
    print_file("test.doc"); break;  
}
```

선택된 메뉴 번호에  
따라 호출할 함수를  
직접 지정해야 한다.

## 함수 포인터 배열을 이용하는 경우

```
pfunc_t pf_arr[] = {  
    open_file, save_file, print_file  
};
```

함수 포인터 배열에 원형이  
같은 함수의 주소를 모아둔다.

```
pf_arr[selected]("test.doc");
```

선택된 메뉴 번호를  
인덱스로 사용해서  
함수를 호출한다.

# 함수 포인터 배열을 이용한 메뉴처리(1/4)

04

```
typedef void(*pfunc_t)(const char*);
```

함수 포인터형 정의

06

```
void open_file(const char* filename) {
```

07

```
    printf("%s 파일을 엽니다.\n", filename);
```

08

```
}
```

09

10

```
void save_file(const char* filename) {
```

11

```
    printf("%s 파일을 저장합니다.\n", filename);
```

12

```
}
```

13

14

```
void print_file(const char* filename) {
```

15

```
    printf("%s 파일을 인쇄합니다.\n", filename);
```

16

```
}
```

함수들의 원형이  
모두 같다.

# 함수 포인터 배열을 이용한 메뉴처리(2/4)

```
18 int main(void)
19 {
20
21     pfunc_t pf_arr[] = { open_file, save_file, print_file };
22
23     const char* menu_str[] = {
24         "열기", "저장", "인쇄", "종료"
25     };
26
27     int size = sizeof(pf_arr) / sizeof(pf_arr[0]);
28
29     while (1) {
30         int i;
31         int selected = 0;
32
33         for (i = 0; i < size + 1; i++)
34             printf("%d. %s ", i, menu_str[i]);
```

메뉴 번호와 인덱스가 같도록  
함수 포인터 배열을 초기화

메뉴 출력에 사용할  
문자열 포인터 배열

메뉴를 출력한다.

# 함수 포인터 배열을 이용한 메뉴처리(3/4)

```
32     printf("선택? ");
33     scanf("%d", &selected);
34     if (selected == size)
35         break;
36     if (selected >= 0 && selected < size)
37         pf_arr[selected]("test.doc");
38     else
39         printf("잘못 선택하셨습니다.\n");
40     }
41 }
```

종료 메뉴(3번)  
선택 시 루프 탈출

선택된 메뉴 번호를 인덱스로  
이용하여 함수 포인터 배열의  
원소로 함수를 호출한다.

# 함수 포인터 배열을 이용한 메뉴처리(4/4)

## 실행 결과

```
0.열기 1.저장 2.인쇄 3.종료 선택? 0
```

test.doc 파일을 엽니다.

```
0.열기 1.저장 2.인쇄 3.종료 선택? 1
```

test.doc 파일을 저장합니다.

```
0.열기 1.저장 2.인쇄 3.종료 선택? 2
```

test.doc 파일을 인쇄합니다.

```
0.열기 1.저장 2.인쇄 3.종료 선택? 3
```

# qsort 함수(1/2)

❖ 쿼크 정렬(quick sort) 알고리즘으로 배열을 오름차순 또는 내림차순으로 정렬하는 함수

```
void qsort(void* ptr, size_t count, size_t size, int(*compare)(const void*, const void*));
```

- *ptr* : 정렬할 배열의 시작 주소
- *count* : 정렬할 배열에 들어있는 원소의 개수
- *size* : 배열 원소 하나의 바이트 크기
- *compare* : 배열의 원소와 키 값을 비교할 때 호출할 함수의 주소

❖ *qsort* 함수를 호출하려면 먼저 배열의 원소와 키 값을 비교할 때 사용될 비교 함수를 정의하고, 그 주소를 *compare*로 전달해야 한다.

# qsort 함수(2/2)

## ❖ 비교 함수의 원형

```
int compare(const void* e1, const void* e2);
```

- $e1, e2$ : qsort 함수에 인자로 전달된 배열 원소를 가리키는 포인터
- $e1$ 이 가리키는 원소가 더 크면 0보다 큰 값을 리턴하고,  $e2$ 가 가리키는 원소가 더 크면 0보다 작은 값을 리턴한다. 두 원소의 값이 같으면 0을 리턴한다.

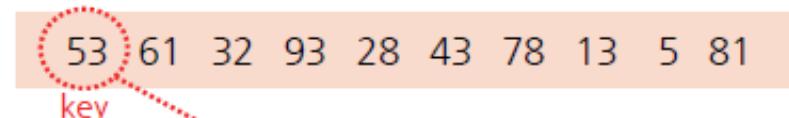
```
int compare_int(const void *e1, const void *e2)
{
    const int *p1 = (const int*)e1;
    const int *p2 = (const int*)e2;
    return (*p1 - *p2);
}
```

$e1, e2$ 는  $\text{void}^*$  형이므로 원소에 대한 포인터형으로 형변환 후 사용해야 한다.

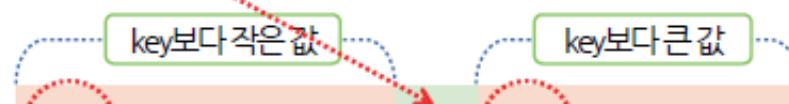
# 퀵 정렬

❖ 배열의 원소 중 하나를 키(key)로 선택하고, 키보다 작은 값과 키보다 큰 값으로 나눈다. 각각에 대해 다시 퀵 정렬을 수행한다. → 분할 정복 알고리즘

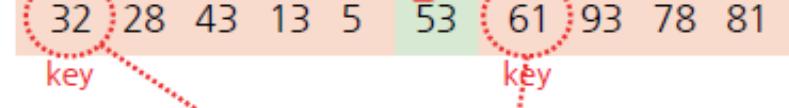
1단계 : 53~81에 대한 퀵 정렬



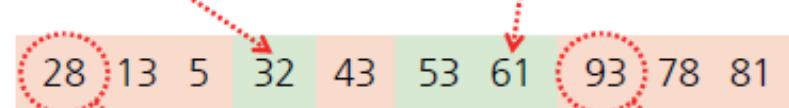
2단계 : 32~5에 대한 퀵 정렬  
61~81에 대한 퀵 정렬



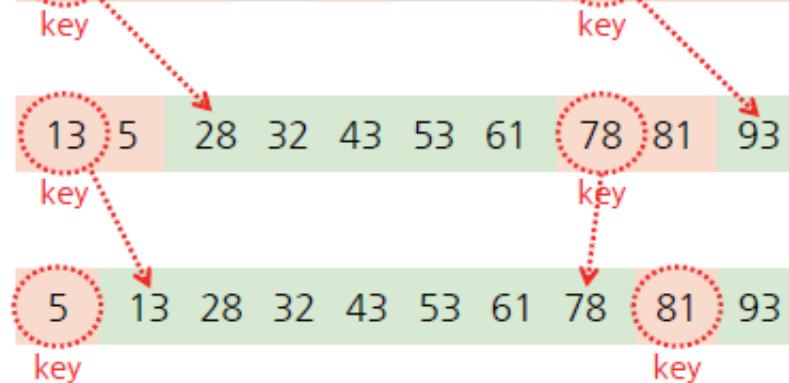
3단계 : 28~5에 대한 퀵 정렬  
43에 대한 퀵 정렬  
93~81에 대한 퀵 정렬



4단계 : 13~5에 대한 퀼 정렬  
78~81에 대한 퀼 정렬



5단계 : 5에 대한 퀼 정렬  
81에 대한 퀼 정렬



# qsort 함수를 이용한 int배열의 정렬(1/3)

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 int compare_int(const void *e1, const void *e2);
05 void print_array(const int arr[], int size);
06
07 int main(void)
08 {
09     int arr[] = { 53, 61, 32, 93, 28, 43, 78, 13, 5, 81 };
10     int size = sizeof(arr) / sizeof(arr[0]);
11
12     printf("정렬 전: ");
13     print_array(arr, size);
```

# qsort 함수를 이용한 int 배열의 정렬(2/3)

```
15     qsort(arr, size, sizeof(arr[0]), compare_int);  
16  
17     printf("정렬 후: ");  
18     print_array(arr, size);  
19 }  
20
```

qsort 함수로 int 배열을 정렬한다.

qsort 함수에 의해서 호출될 비교 함수

```
21     int compare_int(const void *e1, const void *e2)  
22 {  
23     const int *p1 = (const int*)e1;  
24     const int *p2 = (const int*)e2;  
25     return (*p1 - *p2);  
26 }
```

e1, e2는 void\* 형이므로 원소에 대한 포인터형으로 형 변환 후 사용해야 한다.

# qsort 함수를 이용한 int배열의 정렬(3/3)

```
28 void print_array(const int arr[], int size)
29 {
30     int i;
31     for (i = 0; i < size; i++)
32         printf("%d ", arr[i]);
33     printf("\n");
34 }
```

## 실행 결과

정렬 전: 53 61 32 93 28 43 78 13 5 81

정렬 후: 5 13 28 32 43 53 61 78 81 93

# 콜백 함수

❖ 프로그래머가 정의한 함수의 주소를 라이브러리 함수를 호출할 때 전달해서 특정 조건일 때 호출되도록 등록하는 기능

## 사용자 응용 프로그램

```
int compare_int(const void *e1,  
    const void *e2)  
{  
    :  
}  
void qsort(void *ptr,  
    size_t count, size_t size,  
    int(*compare)(const void*, const void*))  
{  
    :  
}
```

```
int main(void)  
{  
    int arr[] = { 53, 61, 32,  
        93, 28, 43, 78, 13, 5, 81  
    };  
    :  
}
```

```
qsort(arr, size, sizeof(arr[0]),  
    compare_int);  
:  
}
```

qsort를 호출하면서  
compare\_int 함수의  
주소를 전달한다.

## 표준 C 라이브러리

```
void qsort(void *ptr,  
    size_t count, size_t size,  
    int(*compare)(const void*, const void*))  
{  
    :  
}
```

```
if( compare(ptr[i], ptr[idx]) > 0 )  
    :  
}
```

```
int rand(void)  
{  
    :  
}
```

라이브러리 내부 코드  
compare 함수 포인터가  
가리키는 compare\_int  
함수를 호출한다.

# Quiz

1. 원형이 `int f(int, int);`인 함수를 가리키는 함수 포인터 `p`가 있을 때 `p`로 함수를 호출하는 코드 중 맞는 것을 모두 고르시오.

- ① `p(10, 20);`
- ② `(*p)(10, 20);`
- ③ `*p(10, 20);`
- ④ `p();`

2. 원형이 `void f(int x, double y);`인 함수를 가리키는 함수 포인터를 올바르게 선언한 것은?

- ① `void p(int, double);`
- ② `void *p(int, double);`
- ③ `void (*p)(int, double);`
- ④ `void *p;`

3. 원형이 `void f(int x, double y);`인 함수를 가리키는 함수 포인터형의 정의는?

- ① `typedef void p(int, double);`
- ② `typedef void *p(int, double);`
- ③ `typedef void (*p)(int, double);`
- ④ `typedef void *p;`