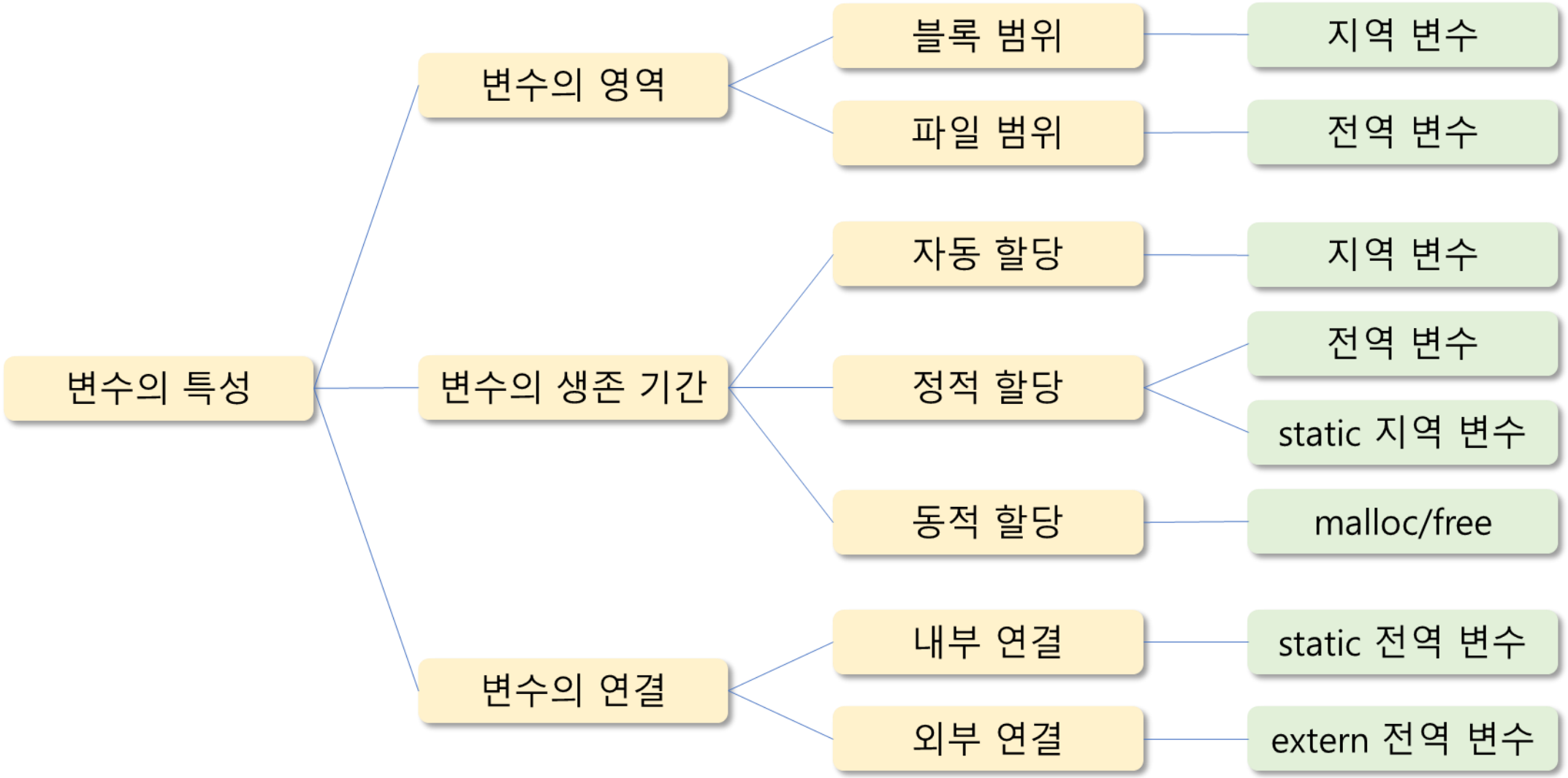
11장. 변수 및 함수의 활용

박 종 혁 교수
서울과학기술대학교 컴퓨터공학과
UCS Lab
Tel: 970-6702
Email: jhpark1@seoultech.ac.kr

목차

변수의 특성 (1/4)


변수의 특성
변수의 영역
블록 범위
지역 변수
파일 범위
전역 변수
변수의 생존 기간
자동 할당
지역 변수
정적 할당
전역 변수
static 지역 변수
동적 할당
malloc/free
변수의 연결
내부 연결
static 전역 변수
외부 연결
extern 전역 변수

변수의 특성 (2/4)

❖변수의 **영역(scope)**

- 변수가 사용될 수 있는 범위
- 변수가 선언된 위치에 의해 결정
- **블록 범위** : 지역 변수

  - 이름이 같은 변수가 여러 개 있을 때 가장 가까운 블록 내의 변수가 우선적으로 사용
- **파일 범위** : 전역 변수

변수의 특성 (3/4)

❖변수의 **생존 기간(lifetime)**

- 변수가 언제 생성되고 소멸되는지
- **자동 할당**
  - 블록에 들어갈 때 메모리가 생성되고 블록을 빠져 나갈 때 메모리가 소멸
  - 메모리의 스택(stack) 영역에 생성
- **정적 할당**
  - 프로그램이 시작될 때 메모리가 생성되고, 프로그램이 종료될 때 메모리가 소멸
  - 정적 메모리(static memory) 영역에 생성
- **동적 할당**
  - 프로그래머가 원하는 시점에 메모리를 할당하고, 원하는 시점에 해제
  - 메모리의 힙(heap) 영역에 생성

변수의 특성 (4/4)

❖변수의 **연결 특성(linkage)**

- 변수를 하나의 소스 파일에서만 사용할 수 있을지, 프로그램 전체에서 사용할 수 있는지를 결정
- 전역 변수만 연결 특성을 가질 수 있음
- **내부 연결**
  - 전역 변수를 하나의 소스 파일에서만 사용하도록 제한
  - static 키워드를 이용
- **외부 연결**
  - 전역 변수를 프로그램 전체에서 사용할 수 있도록 함
  - 전역 변수는 디폴트로 외부 연결 특성을 갖음
  - 다른 소스 파일에 선언된 전역 변수를 참조하려면 전역 변수에 대한 extern 선언이 필요함

기억 부류 지정자

❖생존 기간과 연결 특성에 영향을 주기 위한 키워드

❖auto, register, static, extern

- auto, register, static은 지역 변수의 생존 기간과 할당 위치에 영향
- static와 extern은 전역 변수와 함수의 연결 특성을 지정
- auto, register은 지역 변수에만 사용

형식

기억부류 데이터형 변수명;
기억부류 리턴형 함수명(매개변수목록);

사용예

```
register int i;
static char str[20];
extern int global;
static void do_something(void);
```

auto

- auto로 선언된 지역 변수는 블록에 들어갈 때 자동으로 생성되고, 블록을 빠져나갈 때 자동으로 소멸
- 지역 변수는 디폴트로 auto로 간주
- auto로 지정된 변수를 **자동 변수**라고 함

```
int main(void)
{
    int a = 10;     // 지역 변수는 auto를 생략해도 auto 변수이다.
    auto int b = 20;
     :
}
```

register

- 변수를 메모리에 할당하는 대신 CPU의 레지스터에 할당
- 변수를 레지스터에 할당하면 변수에 좀더 빠르게 접근할 수 있음

```
register int i;     // 자주 반복적으로 사용되는 변수를 register로 지정한다.
for (i = 0; i < 10000; i++)
    sum += i;
```

- 레지스터 변수로 선언해도 변수가 레지스터에 할당되지 않을 수도 있음
- 레지스터 변수에 대해서는 주소 구하기 연산자를 사용할 수 없음

```
register int i;
printf("%p", &i);    // register 변수의 주소를 구할 수 없으므로 컴파일 에러
```

One Definition Rule (1/2)

- 함수는 프로그램 전체에서 반드시 한번만 정의해야 함
- 반면에 선언은 여러 번 할 수 있음

함수는 한번만 정의할 수 있다.

```
void do_something(void)
{
  printf("one definition");
}

int main(void)
{
  do_something();

  return 0;
}

void do_something(void)
{
  printf("one definition");
}
```

함수의 정의

함수의 정의

함수 재정의는 컴파일 에러

함수는 여러 번 선언할 수 있다.

```
void do_something(void);

int main(void)
{
  do_something();

  return 0;
}

void do_something(void);

void do_something(void)
{
  printf("one definition");
}
```

함수의 선언

함수 선언은 여러 번 할 수 있다.

함수의 선언

One Definition Rule (2/2)

❖변수에도 ODR이 적용됨

- 변수도 한번만 정의할 수 있으며, extern 선언은 여러 번 할 수 있음
- 변수의 선언
  - 변수의 데이터형과 이름을 알려주지만 메모리를 할당하지 않음
- 변수의 정의
  - 변수의 메모리를 할당하고 초기화함
- 변수는 선언과 정의가 동시에 이루어짐

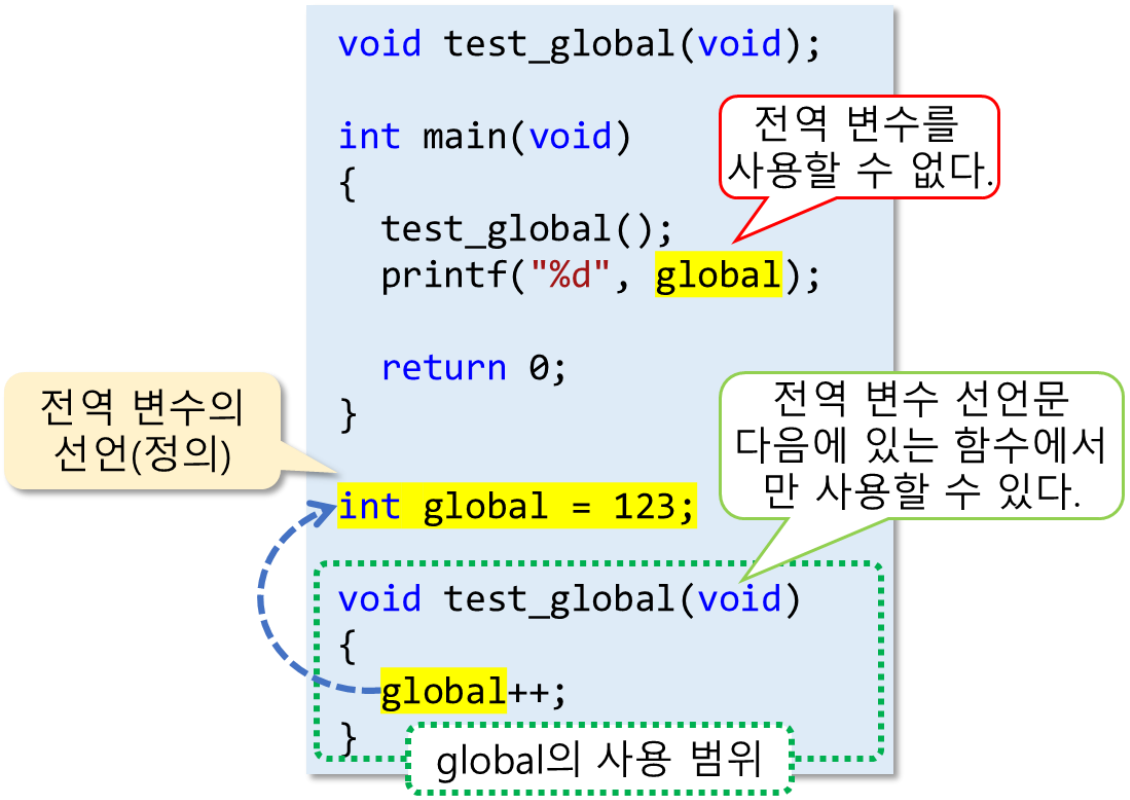
```
int a = 10;          // 변수의 선언이면서 정의
```

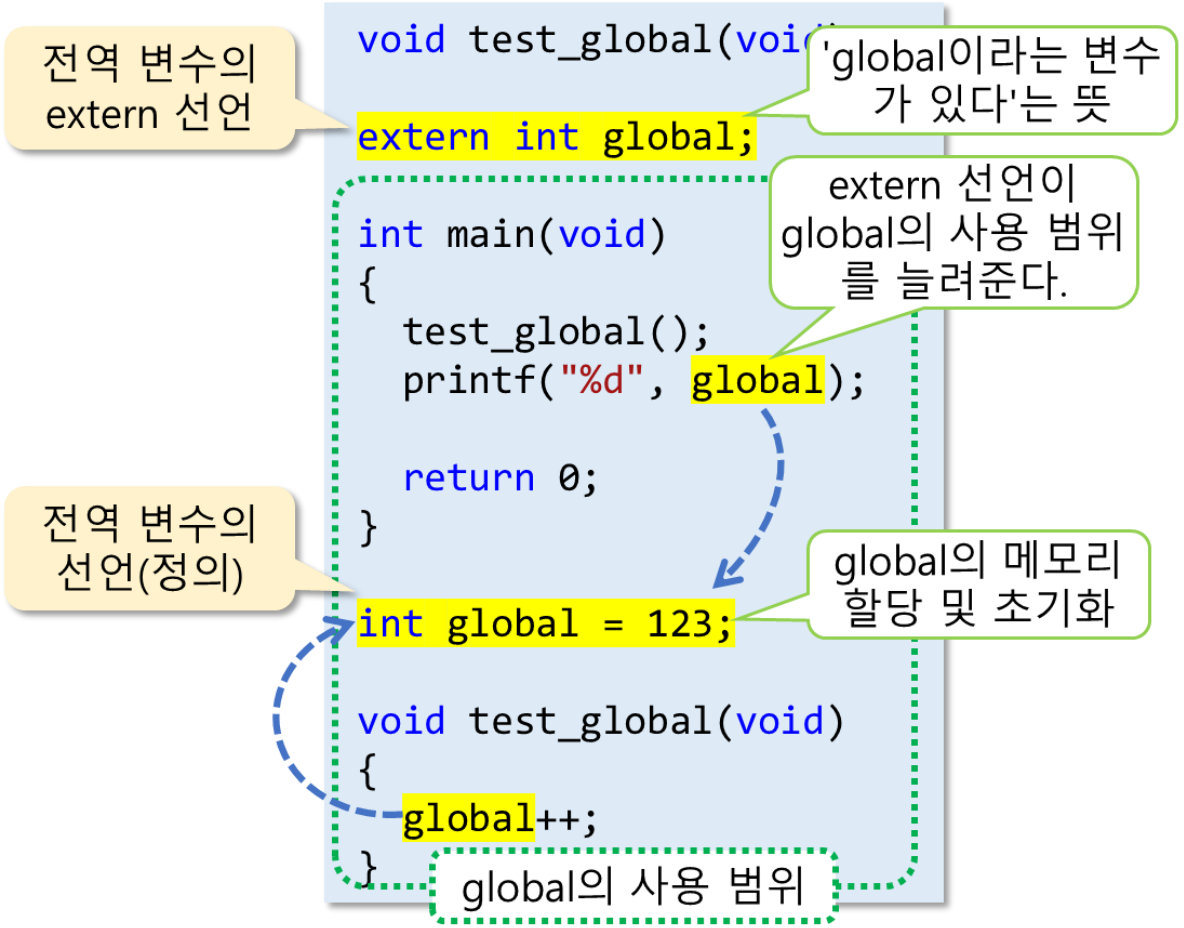
❖**extern** 선언

- extern은 변수에 대해서 정의는 하지 않고 선언만 하게 만듬
- '~라는 변수가 있다'라고 알려주지만, 메모리를 할당하지는 않음
- extern은 전역 변수에만 사용할 수 있음

전역 변수의 extern 선언 (1/5)

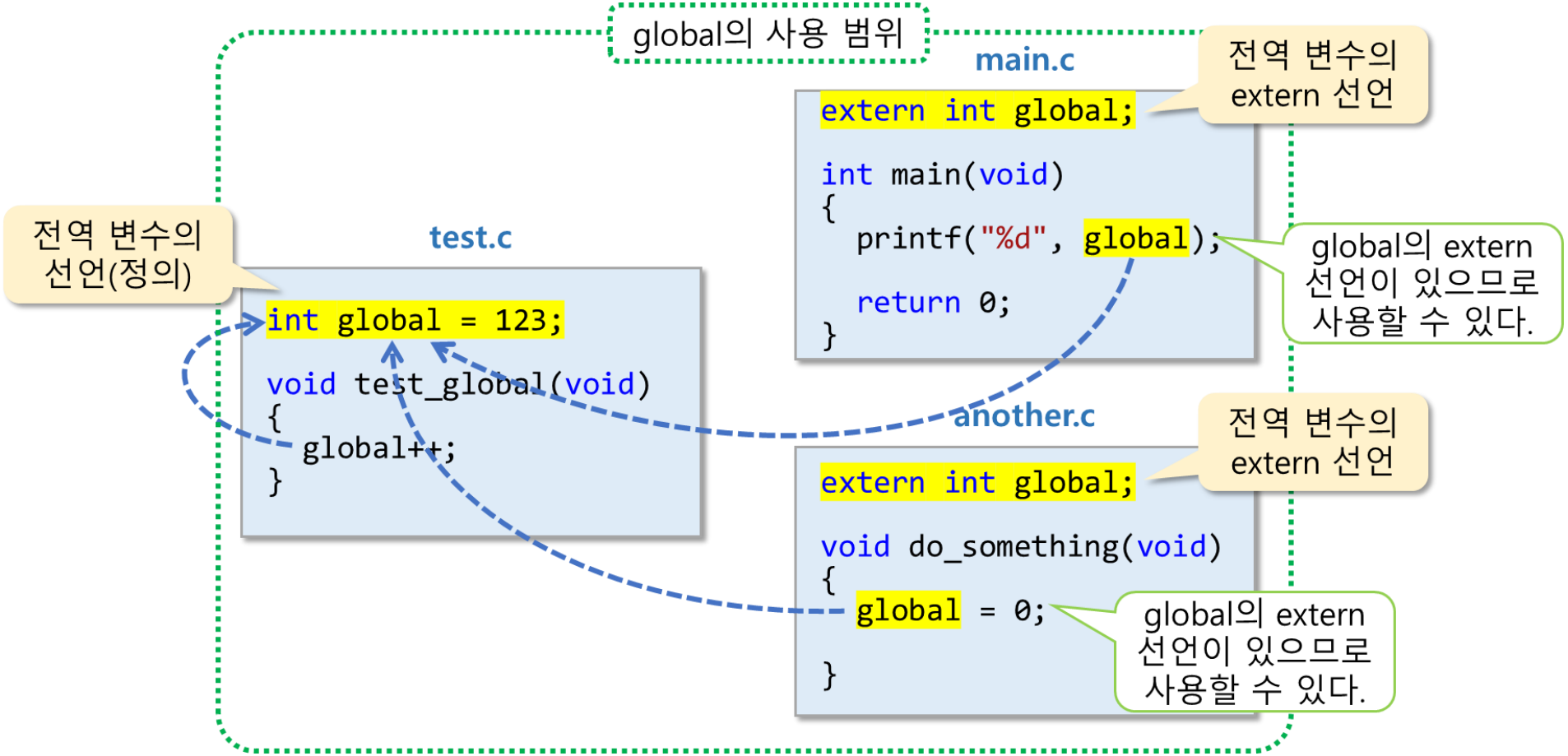
- 전방 선언(forward declaration)





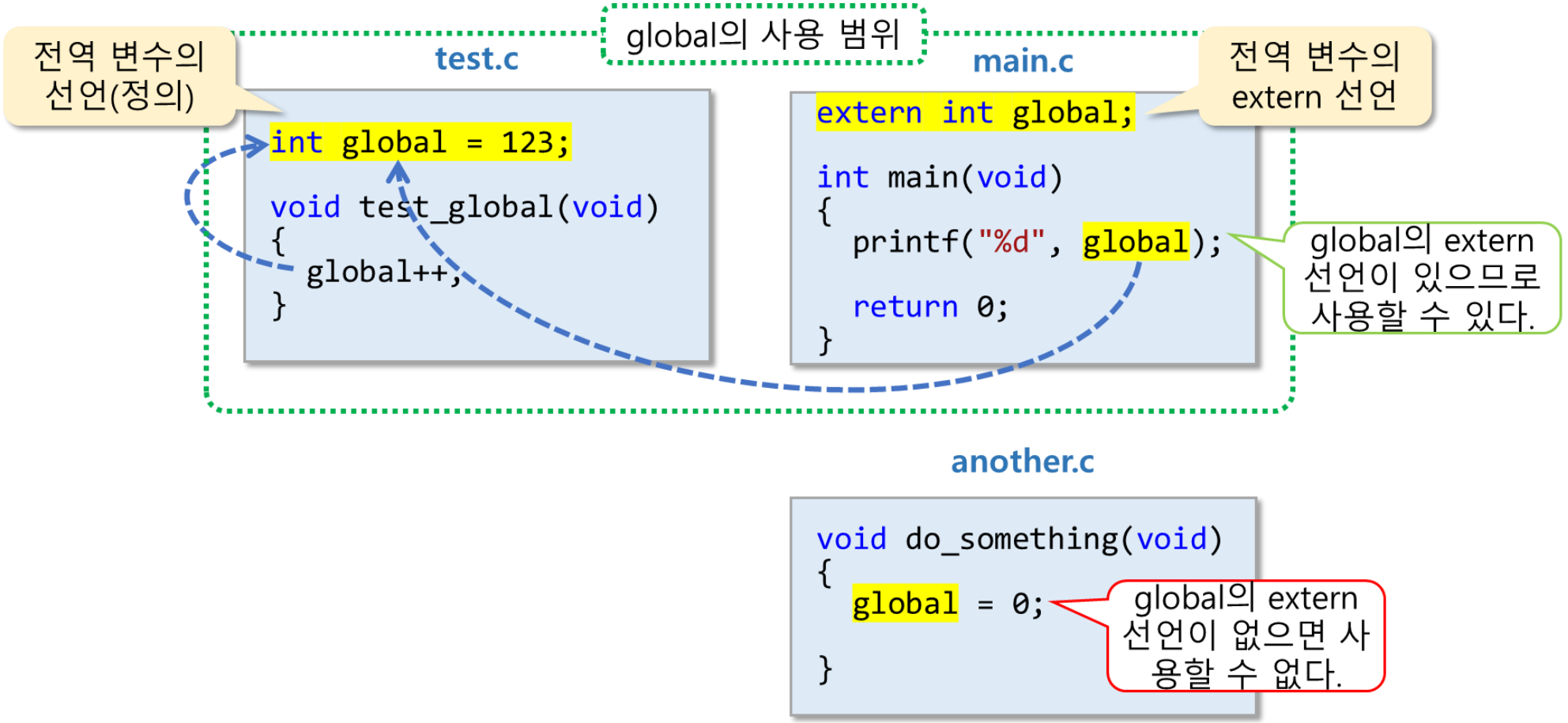
전역 변수의 extern 선언 (2/5)

- 전역 변수의 extern 선언은 파일 범위를 넘어서는 외부 연결 특성을 제공



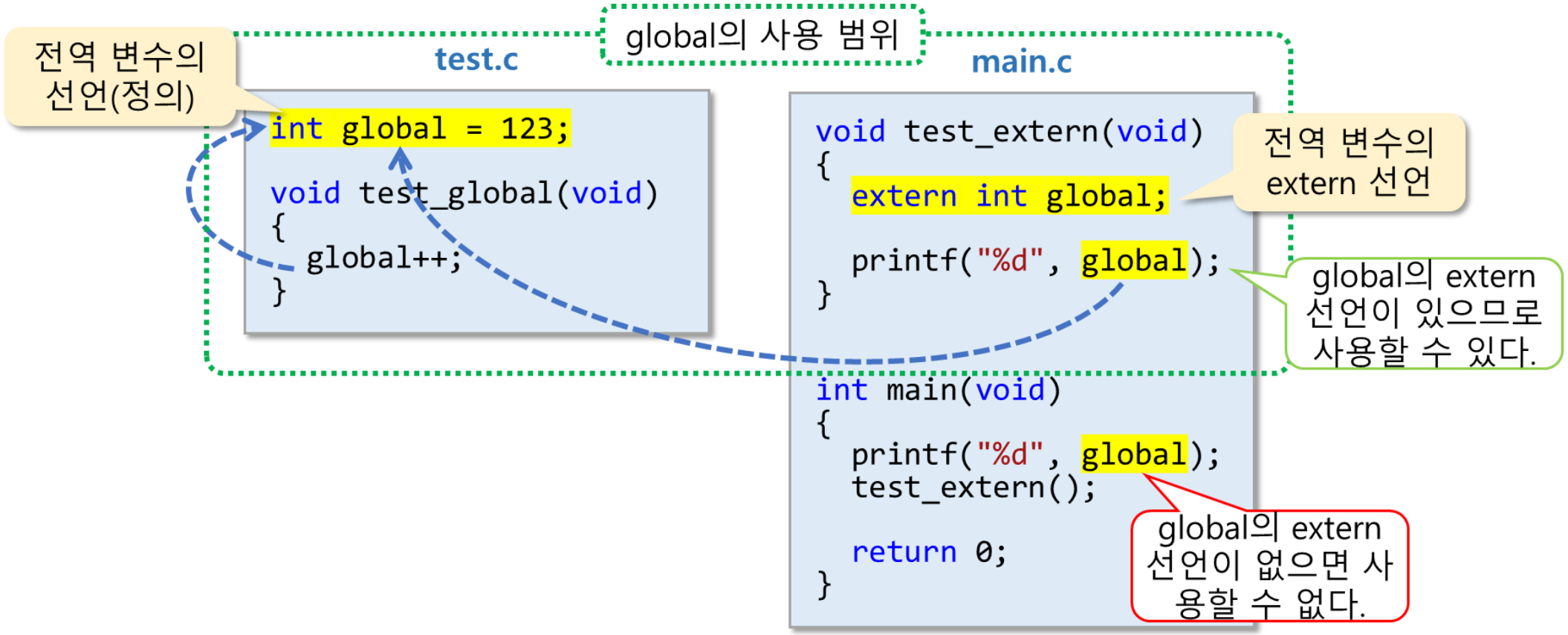
전역 변수의 extern 선언 (3/5)

- 같은 프로그램 내의 소스 파일이 여러 개일 때는 소스 파일마다 전역 변수의 extern 선언이 필요



전역 변수의 extern 선언 (4/5)

❖전역 변수의 extern 선언은 함수 안에 넣어줄 수도 있음
- 해당 함수에서만 전역 변수를 사용할 수 있음



전역 변수의 extern 선언 (5/5)

- 변수의 extern 선언 시 초기화를 하면 변수에 대한 메모리가 할당 (즉, 변수의 선언(정의)로 간주)
- 전역 변수의 선언문 세 가지

```
int global = 123;   // 변수의 선언이면서 정의(메모리 할당 및 초기화)
```

```
extern int global = 123;      // 변수의 선언이면서 정의(메모리 할당 및 초기화)
```

```
extern int global;  // 메모리는 할당되지 않고 '~라는 변수가 있다.'라고 알려준다.
```

예제 : 전역 변수의 extern 선언

test.c

```c
#include <stdio.h>

int global = 123;   // 변수의 선언이면서 정의(메모리 할당 및 초기화)

void test_global(void)
{
    global++;
}
```

main.c

```c
#include <stdio.h>

void test_global(void);       // 다른 소스 파일의 함수를 호출하려면 함수 선언 필요

extern int global;            // 'global이라는 변수가 있다.'라고 알려준다.

int main(void)
{
    test_global();
    printf("global = %d\n", global);

    return 0;
}
```

실행결과

```
global = 124
```

static

❖static 지역 변수

- 지역 변수의 생존 기간에 영향을 줌
- 자동 할당인 지역 변수의 생존 기간을 정적 할당으로 변경
- 지역 변수가 프로그램 시작 시 생성되고, 프로그램 종료 시 소멸되게 만듬

❖static 전역 변수

- 전역 변수의 연결 특성에 영향을 줌
- 전역 변수가 내부 연결 특성을 갖게 만듬

❖static 함수

- 함수의 연결 특성에 영향을 줌
- 함수가 내부 연결 특성을 갖게 만듬

예제 : static 지역 변수

```
03  void test_static(void)
04  {
05      int local = 0;           // 함수가 호출될 때마다 생성된다.
06      static int s_local = 0;  // 프로그램 시작 시 생성된다.
07
08      printf("local = %d, ", local++);
09      printf("s_local = %d\n", s_local++);
10  }     // 함수 리턴 시 local을 소멸되지만 s_local을 소멸되지 않는다.
11
12  int main(void)
13  {
14      int i = 0;
15      for (i = 0; i < 5; i++)
16          test_static();       // test_static을 5번 호출한다.
17
18      return 0;
19  }
```

실행결과

```
local = 0, s_local = 0
local = 0, s_local = 1
local = 0, s_local = 2
local = 0, s_local = 3
local = 0, s_local = 4
```

static 지역 변수 사용 예 : 누산기 프로그램

- 이전 연산의 결과를 lhs로 사용

```
void accumulator(char op, int operand)
{
    static int result = 0;
    switch (op)
    {
    case '+':
        result += operand;
        break;
     ⋮
    default:
        return;
    }
    printf("%c %d = %d\n", op, operand, result);
}
```

accumulator 함수가 호출될 때마다 이전 연산의 결과를 계속해서 이용할 수 있다.

예제 : accumulator 함수의 정의 및 호출

```c
03  void accumulator(char op, int operand);
04
05  int main(void)
06  {
07      while (1)
08      {
09          char op;
10          int num;
11          printf("연산자와 정수를 입력하세요(. 0 입력시 종료): ");
12          scanf(" %c %d", &op, &num);
13          if (op == '.' && num == 0)
14              break;
15          accumulator(op, num);
16      }
17
18      return 0;
19  }
21    void accumulator(char op, int operand)
22    {
23          static int result = 0;
24          switch (op)
25          {
26          case '+':
27              result += operand;
28              break;
29          case '-':
30              result -= operand;
31              break;
32          case '*':
33              result *= operand;
34              break;
35          case '/':
36              result /= operand;
37              break;
38          default:
39              return;
40          }
41          printf("%c %d = %d\n", op, operand, result);
42    }
```

실행결과

```
연산자와 정수를 입력하세요(. 0 입력시 종료): + 10
+ 10 = 10
연산자와 정수를 입력하세요(. 0 입력시 종료): + 30
+ 30 = 40
연산자와 정수를 입력하세요(. 0 입력시 종료): * 5
* 5 = 200
연산자와 정수를 입력하세요(. 0 입력시 종료): . 0
```

accumulator 함수의 구현

static 지역 변수로 선언하는 경우

result의 영역이 블록 범위로 제한된다.

```
void accumulator(char op,
  int operand)
{
  static int result = 0;

  switch (op)
  {
  case '+':
    result += operand;
    break;
      ⋮
  }
}

void dummy(void)
{
  result = 0;
}
```

result의 사용 범위

다른 함수에서 result를 사용하면 컴파일 에러가 발생한다.

전역 변수로 선언하는 경우

```
int result = 0;

void accumulator(char op,
  int operand)
{
  switch (op)
  {
  case '+':
    result += operand;
    break;
      ⋮
  }
}

void dummy(void)
{
  result = 0;
}
```

result의 영역은 파일 범위이다.

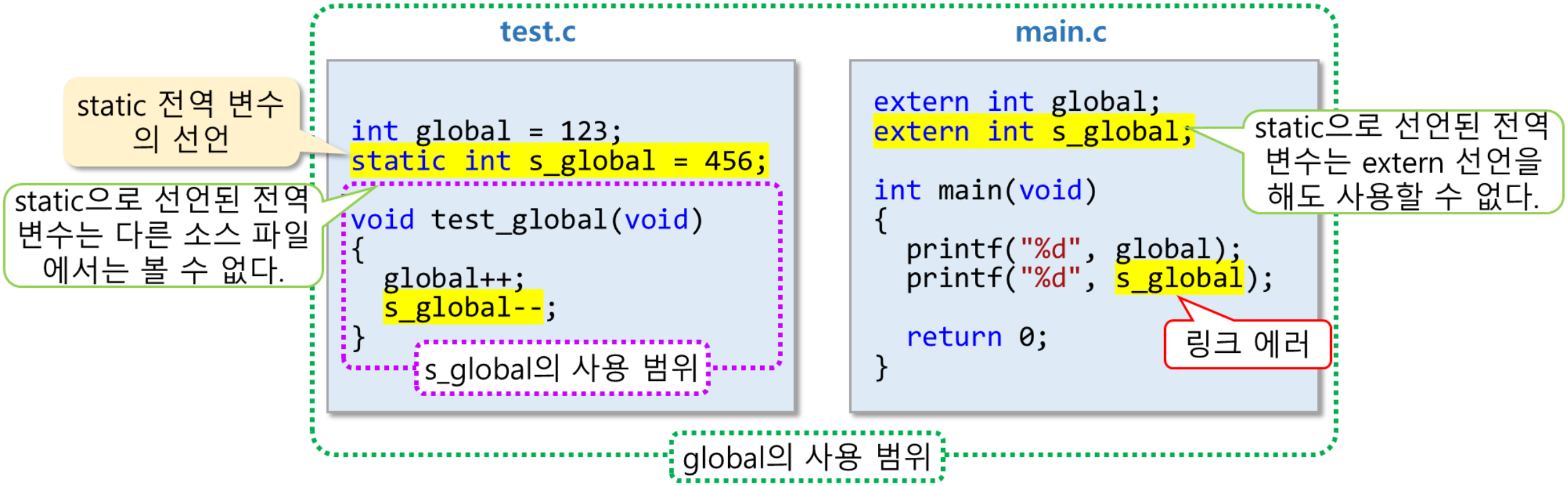
다른 함수에서도 result를 변경할 수 있다.

result의 사용 범위

static 전역 변수

❖전역 변수의 연결 특성을 내부 연결로 지정

- 선언된 소스 파일에서만 전역 변수를 사용하도록 제한

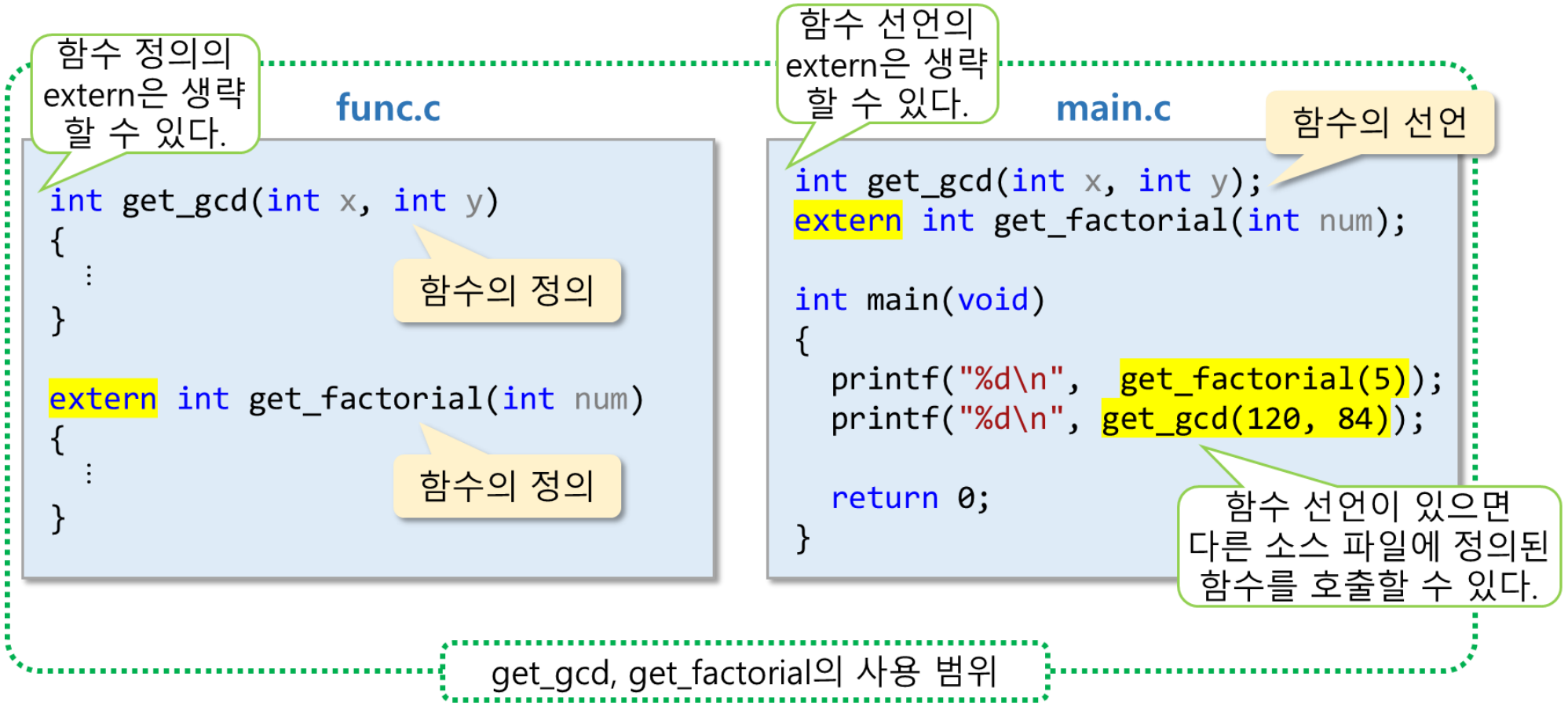


static 키워드

구분	지역 변수	static 지역 변수	전역 변수	static 전역 변수
선언 위치	함수 안	함수 안	함수 밖	함수 밖
생성 시점	변수 선언 시	프로그램 시작 시	프로그램 시작 시	프로그램 시작 시
소멸 시점	함수 리턴 시	프로그램 종료 시	프로그램 종료 시	프로그램 종료 시
사용 범위	함수 안	함수 안	프로그램 전체	선언된 소스 파일
초기화하지 않았을 때	쓰레기 값	0으로 초기화	0으로 초기화	0으로 초기화

함수의 외부 연결 특성

❖함수는 디폴트로 extern으로 선언되어 외부 연결 특성을 제공
- 함수 선언의 extern은 보통 생략



예제 : 다른 소스 파일에 정의된 함수의 호출

func.c

```c
// 함수의 정의를 모아놓은 소스 파일 func.c

int get_gcd(int x, int y)
{
    int r;
    while (y != 0) {
        r = x % y;
        x = y;
        y = r;
    }
    return x;
}

extern int get_factorial(int num)
{
    int i;
    int result = 1;

    for (i = 1; i <= num; i++)
        result *= i;
    return result;
}
```

main.c

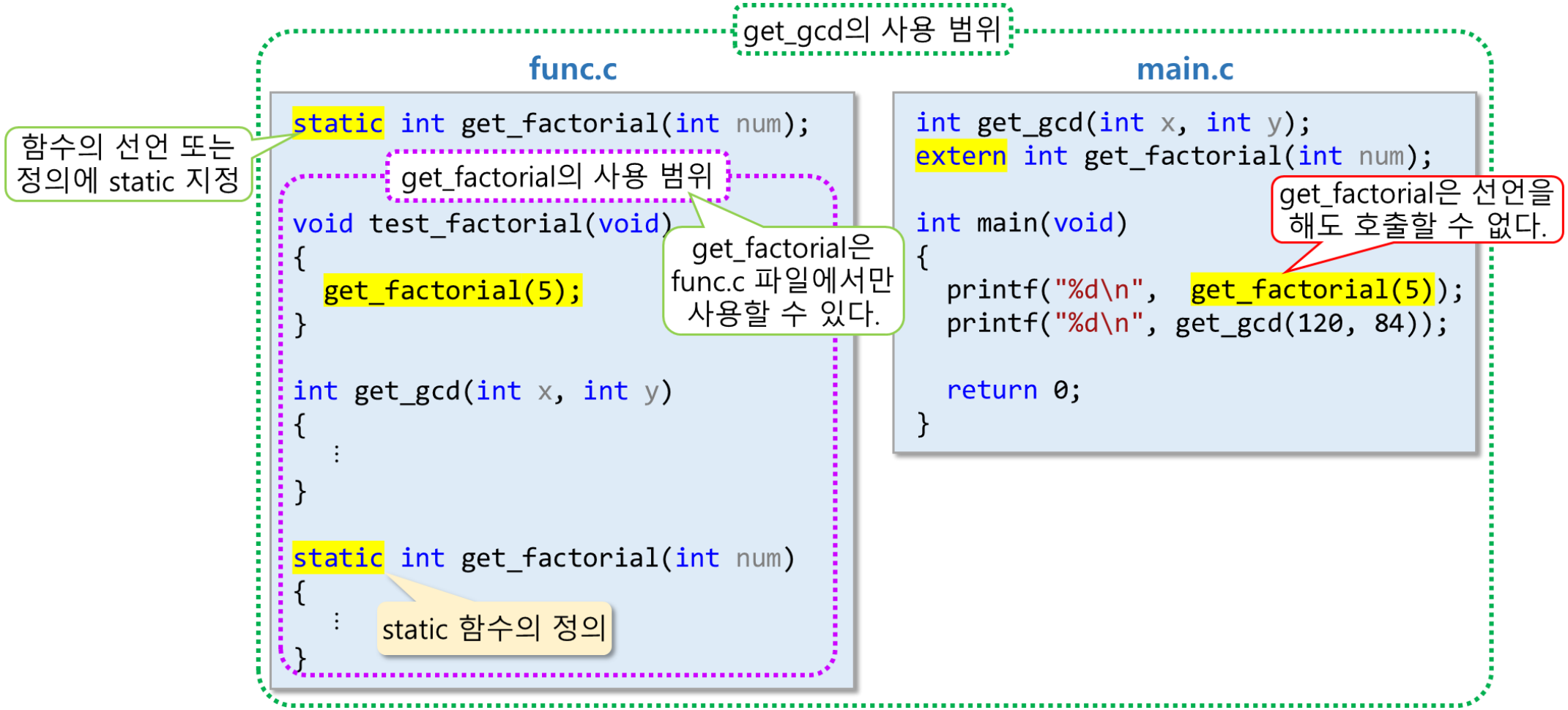
```c
#include <stdio.h>
#include <stdlib.h>                 // srand, rand 호출 시 필요
#include <time.h>                   // time 호출 시 필요

// 다른 소스 파일에 정의된 함수를 호출하려면 함수 선언이 필요하다.
int get_gcd(int x, int y);          // 함수 선언의 extern은 생략할 수 있다.
extern int get_factorial(int num);

int main(void)
{
    int i;
    srand((unsigned int)time(NULL));            // 난수의 시드를 지정한다.

    // 0~9사이의 임의의 정수에 대해서 팩토리얼을 구한다.
    for (i = 0; i < 5; i++)
    {
        int num = rand() % 10;
        printf("%2d! = %7d\n", num, get_factorial(num));
    }

    // 0~99사이의 임의의 정수 2개에 대해서 최대 공약수를 구한다.
    for (i = 0; i <= 5; i++)
    {
        int a = rand() % 100;
        int b = rand() % 100;
        printf("%2d와 %2d의 GCD = %2d\n", a, b, get_gcd(a, b));
    }

    return 0;
}
```

실행결과

```
 4! =      24
 6! =     720
 5! =     120
 4! =      24
 2! =       2
24와 24의 GCD = 24
97와 32의 GCD =  1
34와 75의 GCD =  1
51와 39의 GCD =  3
97와  2의 GCD =  1
43와 33의 GCD =  1
```

static 함수

- static 함수는 함수가 정의된 소스 파일 밖에서는 호출할 수 없음



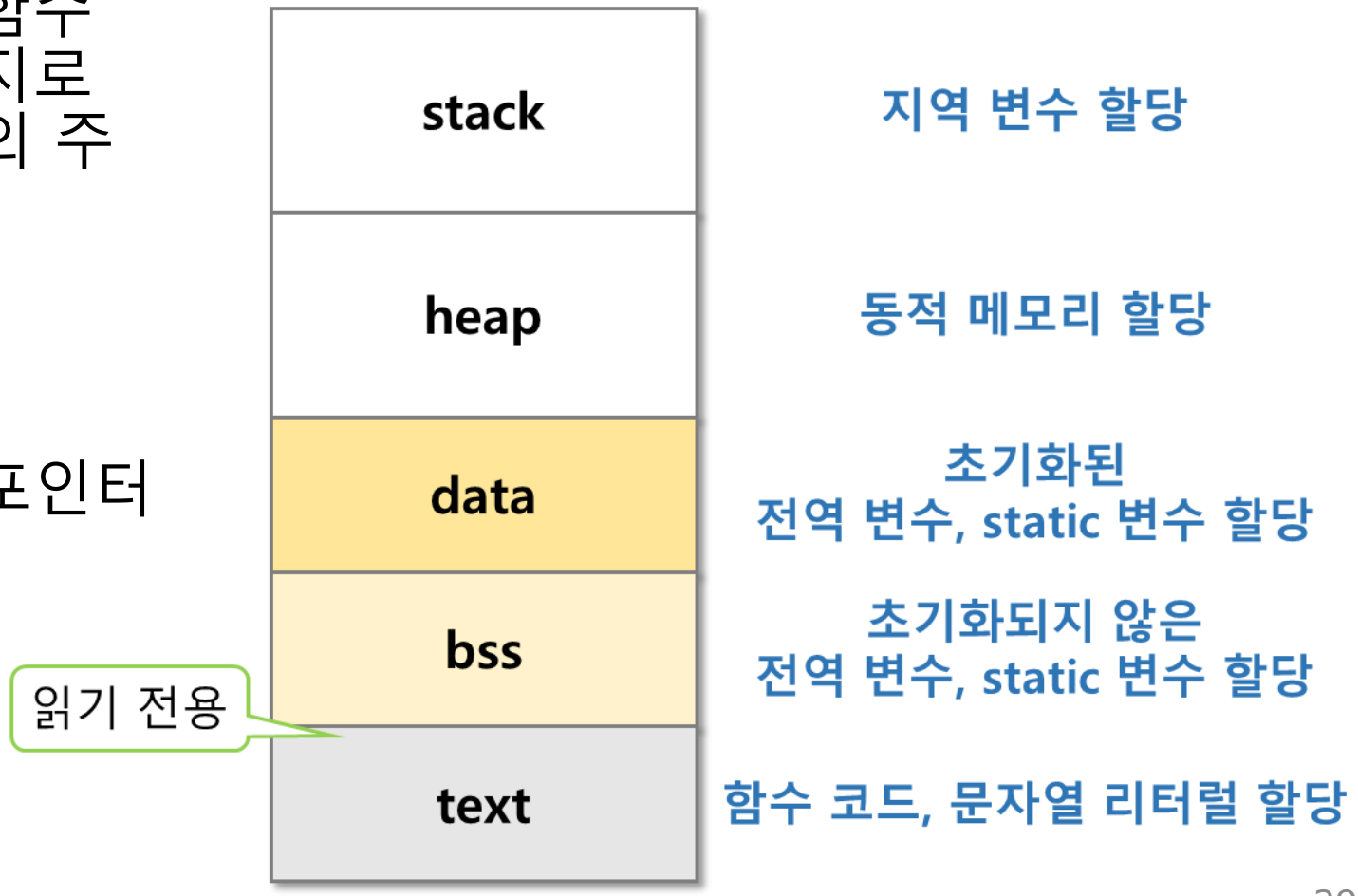
함수와 전역 변수를 사용하기 위한 가이드라인

1. 소스 파일은 기능 단위로 나누어 작성
   - 관련된 함수와 전역 변수를 모아서 소스 파일을 구성
   - 소스 파일명도 의미 있는 이름으로 정하는 것이 좋음
2. 소스 파일에서 외부로 노출해야 하는 함수나 전역 변수는 **extern**으로 정의
   - 이때 extern 키워드는 생략할 수 있음
   - 헤더 파일에 함수 선언과 전역 변수의 extern 선언을 넣어줌
   - 소스 파일명이 *test.c*면 헤더 파일명은 *test.h*로 지정함
3. 소스 파일 내부에서만 사용되는 함수나 전역 변수는 **static**으로 정의
4. 프로그램의 나머지 부분에서 *test.c*에 있는 함수나 전역 변수를 사용하려면 헤더 파일인 *test.h*를 포함

프로그램의 메모리 레이아웃

❖함수에도 메모리 주소가 있음
- 텍스트 세그먼트에 있는 함수 코드도 메모리의 특정 번지로 로드되서 실행 중에 함수의 주소로 사용됨

❖함수 포인터
- 함수의 주소를 저장하는 포인터

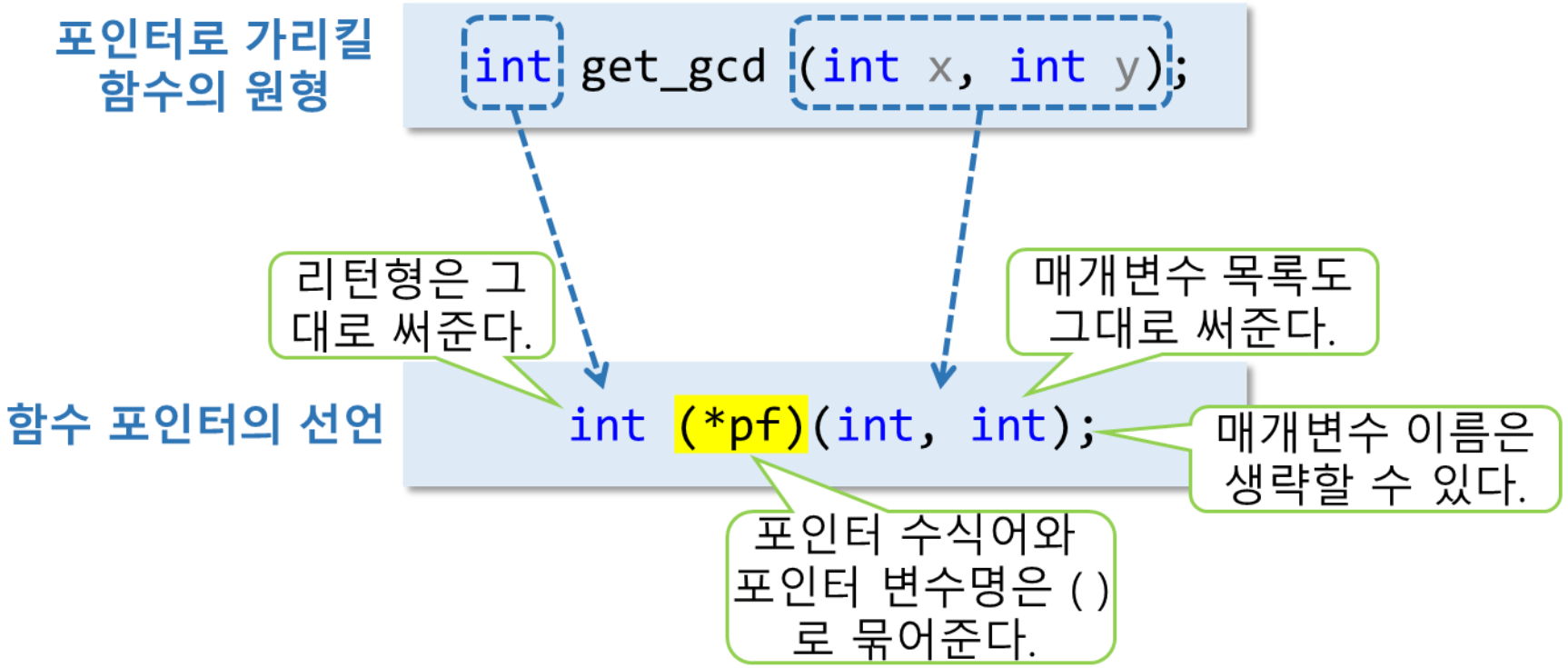


함수 포인터의 선언 (1/2)

형식 리턴형 (*포인터명)(매개변수목록);

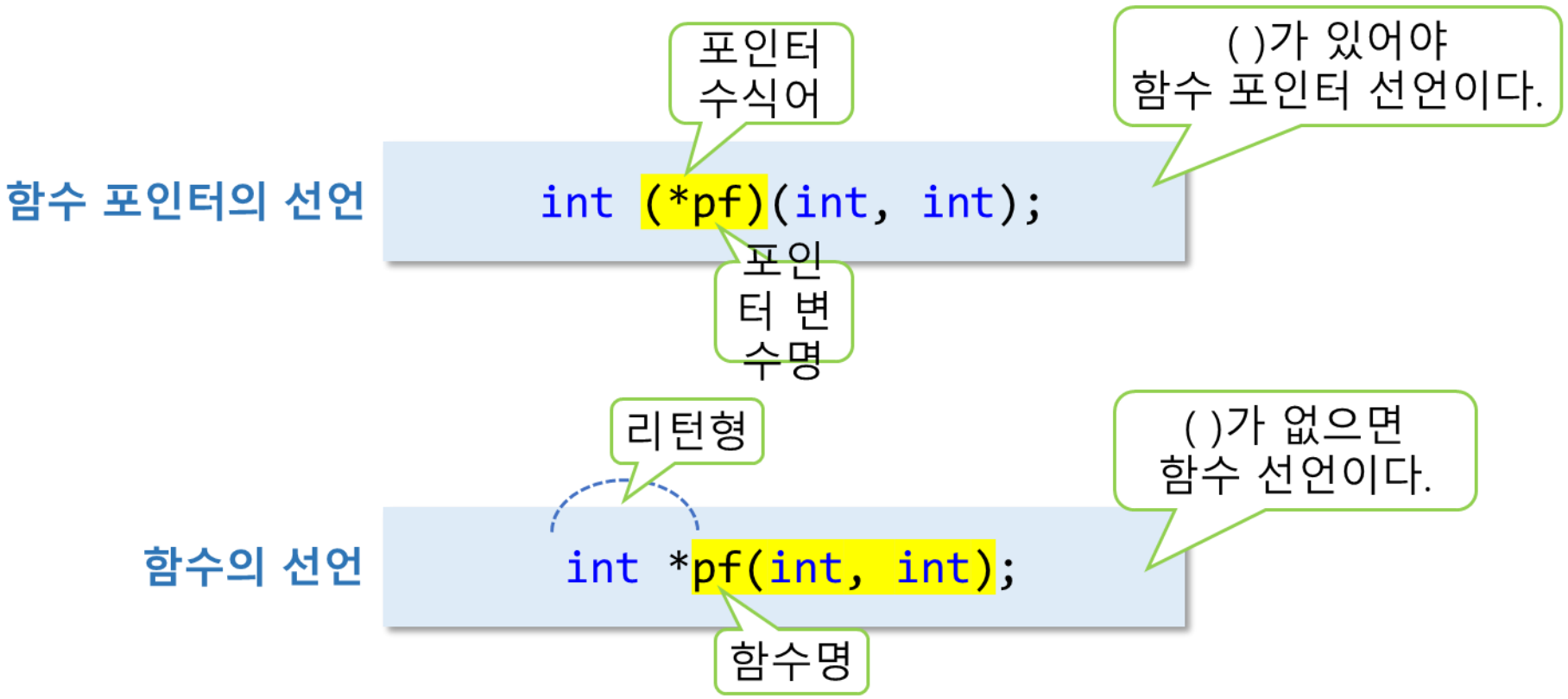
사용예

```
int(*pf)(int, int) = get_gcd;
void(*pprint)(const POINT*) = NULL;
```



함수 포인터의 선언 (2/2)

- 포인터 수식어와 포인터 변수명을 반드시 ( )로 묶어주어야 함



함수 포인터의 초기화

- 아직 가리키는 함수가 없으면 널 포인터로 초기화함

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

- 함수의 주소를 구하려면 주소 연산자(&)와 함께 함수의 이름을 사용

```
int (*pf)(int, int) = &get_gcd;
```

- 함수의 주소를 구할 때는 & 연산자 없이 함수 이름만 사용해도 됨

```
int (*pf)(int, int) = get_gcd;
```

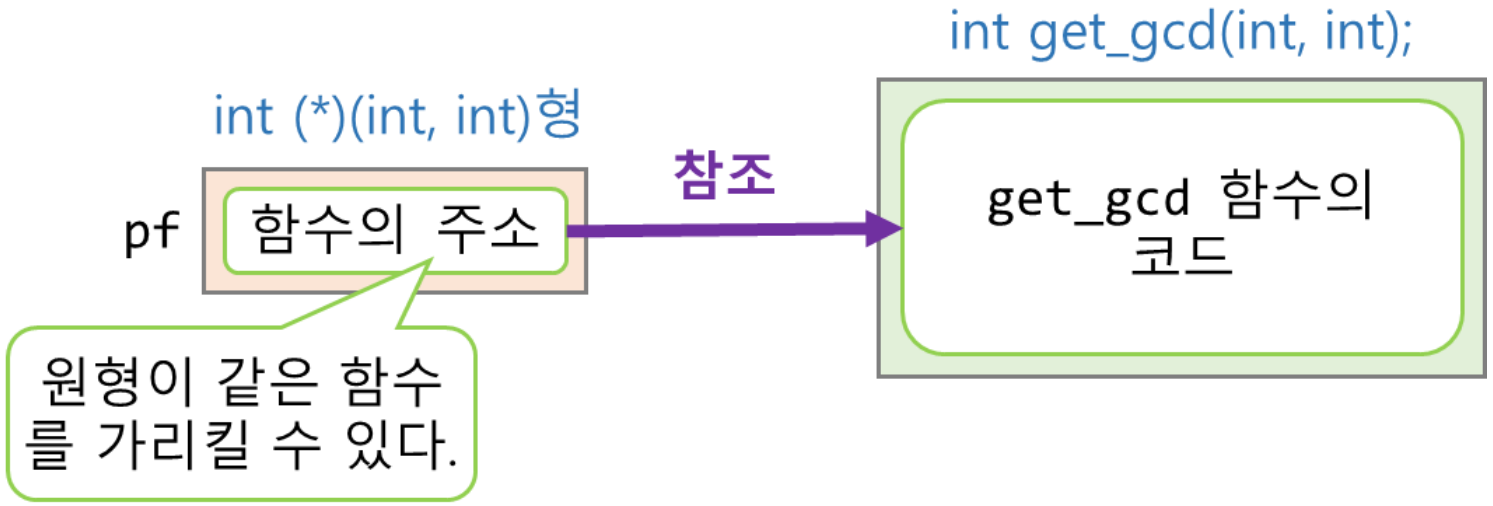
함수 포인터를 이용한 함수 호출

- 역참조 연산자를 이용해서 함수를 호출할 수 있음

```
printf("%d", (*pf)(10, 20));
```

- 역참조 연산자 없이 함수 포인터를 직접 함수 이름인 것처럼 사용할 수도 있음

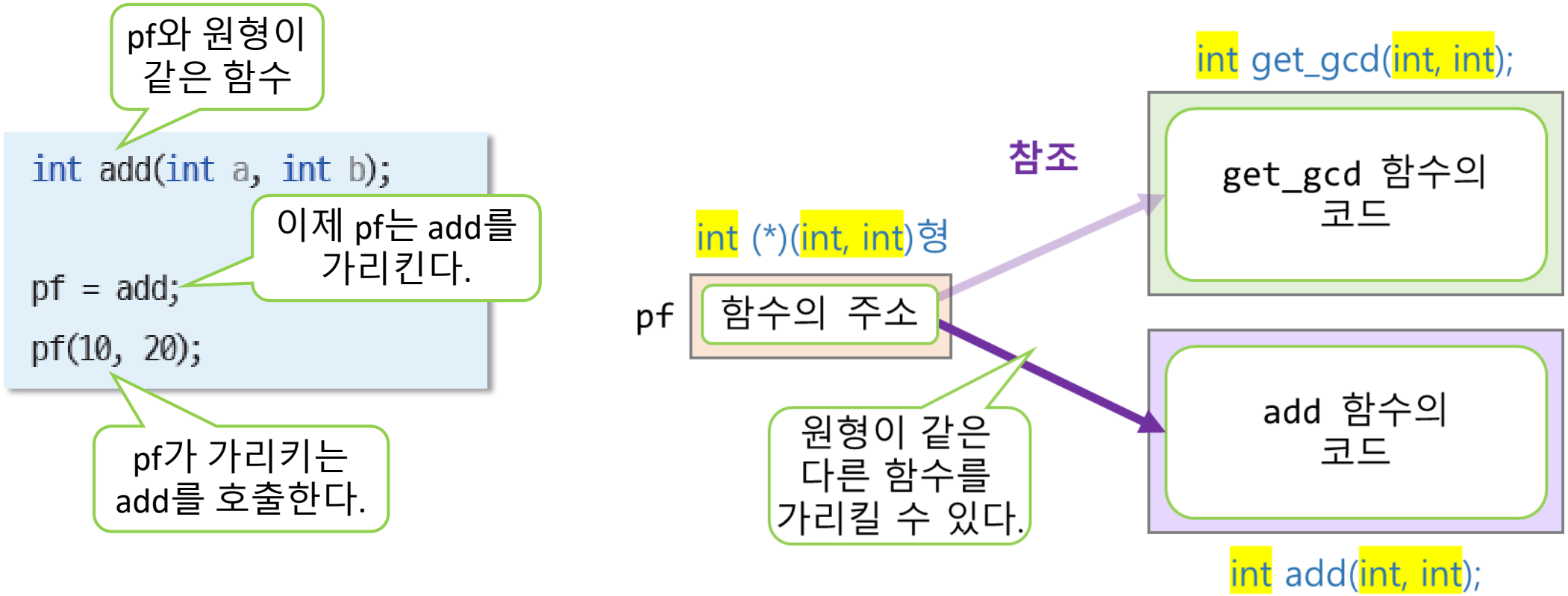
```
printf("%d", pf(10, 20));
```



함수 포인터의 변경

❖함수 포인터도 변수이므로 값을 변경할 수 있음

- 함수 포인터가 다른 함수를 가리키게 만들 수 있음
- 함수 포인터의 원형과 함수 포인터가 가리키는 함수의 원형이 같아야 함



예제 : 함수 포인터의 선언 및 사용

```c
typedef struct point
{
    int x, y;
} POINT;

void print_point(const POINT *pt)      // 구조체를 매개변수로 갖는 함수
{
    printf("(%d, %d)", pt->x, pt->y);
}

int get_gcd(int x, int y)
{
    if (x % y == 0)
        return y;
    return get_gcd(y, x % y);
}
int add(int a, int b)
{
    return a + b;
}

int main(void)
{
    int(*pf)(int, int) = &get_gcd;
    void(*pprint)(const POINT*) = print_point;
    POINT pt = { 10, 20 };

    // pf가 get_gcd를 가리키므로 get_gcd를 호출한다.
    if (pf)         // pf는 포인터이므로 널 포인터인지 검사하는 것이 안전하다.
        printf("GCD = %d\n", pf(10, 20));
    pf = add;
    // pf가 이제 add를 가리키므로 add를 호출한다.
    printf("10 + 20 = %d\n", (*pf)(10, 20));

    // pprint가 가리키는 print_point를 호출한다.
    pprint(&pt);

    return 0;
}
```

실행결과

```
GCD = 10
10 + 20 = 30
(10, 20)
```

함수 포인터형 (1/2)

형식 typedef 리턴형 (*포인터형명)(매개변수목록);

사용예

```
typedef int(*FUNCPTR)(int, int);
typedef void(*PPRT)(const POINT*);
```

함수 포인터형의 변수로 가리킬 함수의 원형

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

함수 포인터형의 정의

```
typedef int (*FUNCPTR)(int, int);
```

- 리턴형은 그대로 써준다.
- 매개변수 목록도 그대로 써준다.
- 매개변수 이름은 생략할 수 있다.
- 포인터 수식어와 함수 포인터형명은 ( )로 묶어준다.

FUNCPTR은 데이터형 이름이다.

함수 포인터형 (2/2)

- 함수 포인터형의 변수를 선언할 수 있음 → 함수 포인터 변수

```
FUNCPTR pf = NULL; // pf는 함수 포인터 변수이다.
```

```
pf = get_gcd;                          // get_gcd 함수의 주소를  pf에 저장한다.
printf("GCD = %d\n", pf(10, 20));      // 함수 포인터로 함수를 호출한다.
```

- 매개변수가 구조체형일 때는 구조체 정의 후에 함수 포인터형을 정의해야 함

```
typedef struct point {
    int x, y;
} POINT;

typedef void(*PFPRINT)(const POINT*);
```

구조체를 먼저 정의해야 한다.

매개변수가 구조체형인 경우

예제 : 함수 포인터형의 정의 및 사용

```c
typedef struct point
{
    int x, y;
} POINT;

typedef int(*FUNCPTR)(int, int);
typedef void(*PFPRINT)(const POINT*);

void print_point(const POINT *pt)      // 구조체를 매개변수로 갖는 함수
{
    printf("(%d, %d)", pt->x, pt->y);
}

int get_gcd(int x, int y)
{
    if (x % y == 0)
        return y;
    return get_gcd(y, x % y);
}

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

int main(void)
{
    FUNCPTR pf = get_gcd;              // pf는 함수 포인터 변수이다.
    PFPRINT pprint = print_point;      // pprint는 함수 포인터 변수이다.
    POINT pt = { 10, 20 };

    // pf가 get_gcd를 가리키므로 get_gcd를 호출한다.
    if (pf)         // pf는 포인터이므로 널 포인터인지 검사하는 것이 안전하다.
        printf("GCD = %d\n", pf(10, 20));

    pf = add;
    // pf가 이제 add를 가리키므로 add를 호출한다.
    printf("10 + 20 = %d\n", (*pf)(10, 20));

    // pprint가 가리키는 print_point를 호출한다.
    pprint(&pt);

    return 0;
}
```

실행결과

```
GCD = 10
10 + 20 = 30
(10, 20)
```

함수 포인터 배열 (1/3)

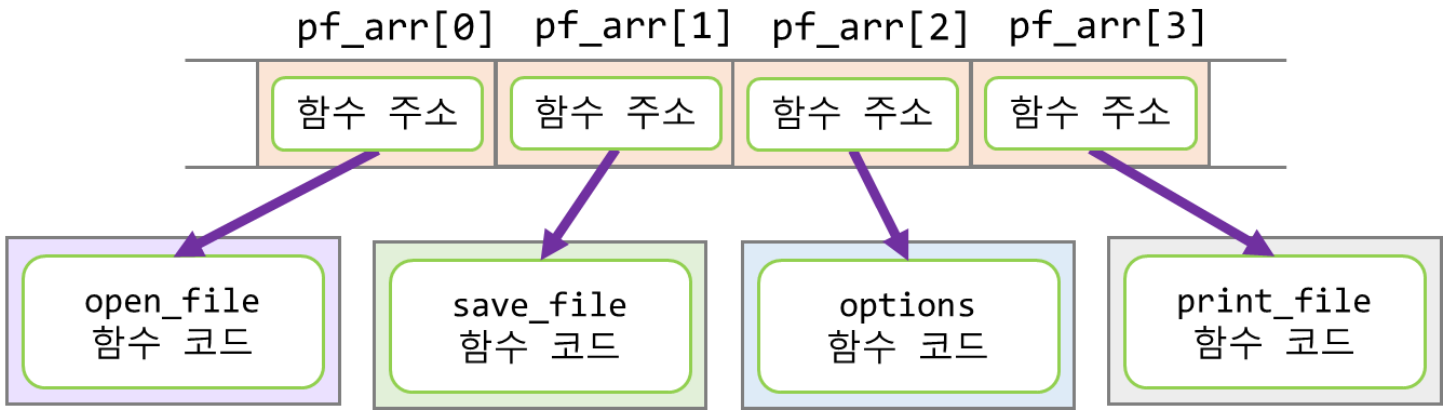
- 원형이 같고 함께 사용되는 함수들을 함수 포인터 배열에 원소로 저장하고 사용할 수 있음

```
void open_file(void);
void save_file(void);
void options(void);
void print_file(void);

PFUNC pf_arr[4] = {
    open_file, save_file, options, print_file
};
```

함수들의 원형이 같다.

함수의 주소로 배열의 원소를 초기화한다.

함수 포인터 배열



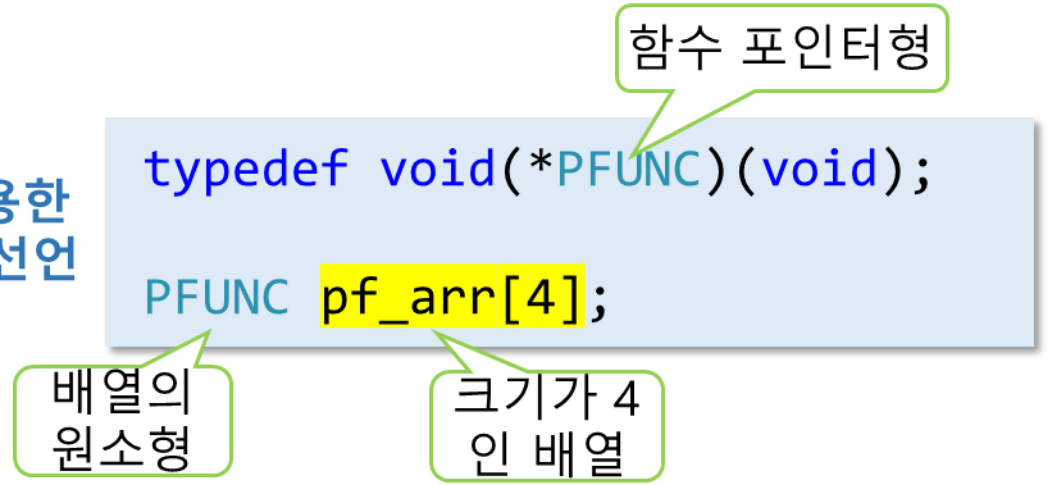
함수 포인터 배열 (2/3)

- 원형이 같은 함수들의 주소를 함수 포인터 배열에 저장한 경우, 함수 포인터 배열을 이용해서 함수들을 호출할 수 있음

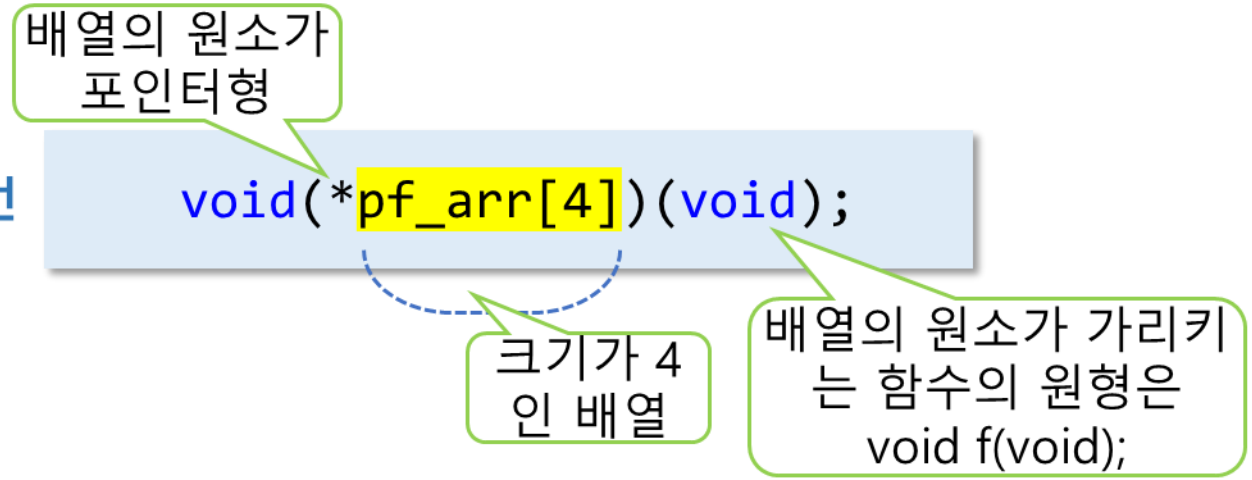
```
int selected = 0;
scanf("%d", &selected);          // 파일 열기가 0번, 파일 저장이 1번, … 메뉴인 경우
if (selected >= 0 && selected < size)
    pf_arr[selected]();          // 메뉴 번호가 0~3번인 경우 메뉴 번호를 인덱스로 하는
                                 // 함수 포인터 배열의 원소로 함수를 호출한다.
```

함수 포인터 배열 (3/3)

함수 포인터형을 이용한
함수 포인터 배열의 선언



함수 포인터 배열의 선언



예제 : 함수 포인터 배열의 사용

```c
void open_file(void)
{
    printf("파일을 오픈합니다....\n");
}

void save_file(void)
{
    printf("파일을 저장합니다....\n");
}

void options(void)
{
    printf("옵션을 설정합니다....\n");
}

void print_file(void)
{
    printf("파일을 인쇄합니다....\n");
}

typedef void(*PFUNC)(void);   // 함수 포인터형

int main(void)
{
    // 함수의 주소로 함수 포인터 배열을 초기화한다.
    // 메뉴 번호와 메뉴를 선택했을 때 호출할 함수의 인덱스가 같아야 한다.
    PFUNC pf_arr[] = { open_file, save_file, options, print_file };
    //void(*pf_arr[])(void) = {
    //    open_file, save_file, options, print_file
    //};

    const char *menu_str[] = {          // 메뉴 출력에 사용할 문자열 포인터 배열
        "파일 열기", "파일 저장", "옵션", "인쇄", "종료"
    };
    enum {OPEN_FILE, CLOSE_FILE, OPTIONS, PRINT, QUIT};    // 열거 상수
    int size = sizeof(pf_arr) / sizeof(pf_arr[0]);

    while (1)
    {
        int i;
        int selected = 0;
        for (i = 0; i < size + 1; i++) // 종료 메뉴까지 출력해야 한다.
            printf("%d. %s\n", i, menu_str[i]);
        printf("선택? ");
        scanf("%d", &selected);
        if (selected == QUIT)
            break;
        if (selected >= 0 && selected < size) {
            // 선택된 메뉴 번호를 함수 포인터 배열의 인덱스로 사용한다.
            pf_arr[selected]();
        }
        else {
            printf("잘못 선택하셨습니다.\n");
        }
    }
    return 0;
}
```

실행결과

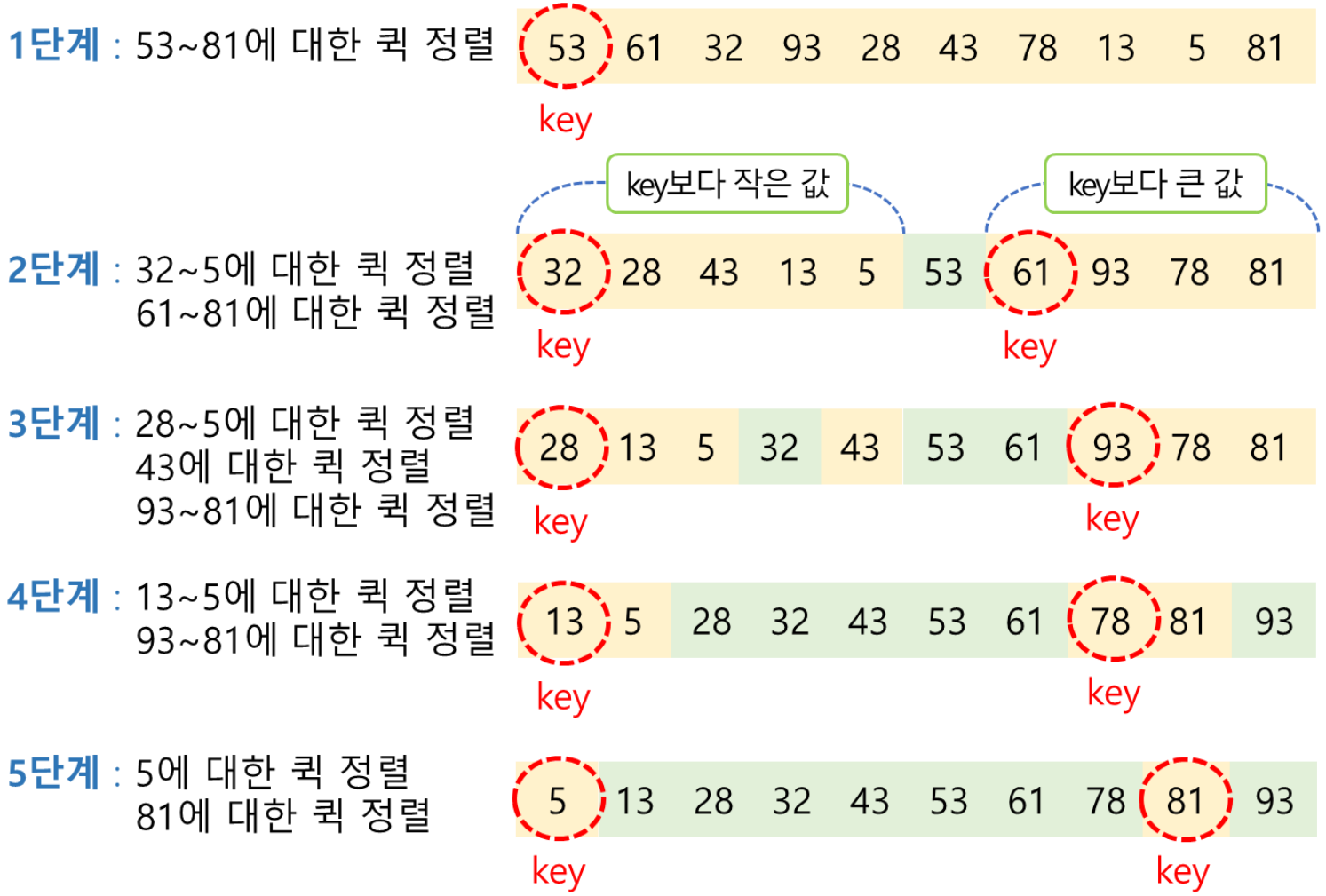
```
0. 파일 열기
1. 파일 저장
2. 옵션
3. 인쇄
4. 종료
선택? 0
파일을 오픈합니다....
0. 파일 열기
1. 파일 저장
2. 옵션
3. 인쇄
4. 종료
선택? 1
파일을 저장합니다....
0. 파일 열기
1. 파일 저장
2. 옵션
3. 인쇄
4. 종료
선택? 4
```

추가 자료

함수 포인터의 활용 : qsort 함수 (1/2)

❖퀵 정렬

- 정렬할 배열의 원소 중 하나를 선택한 다음 그 값을 키(key)라고 함
- 정렬할 배열의 원소들을 키보다 작은 값, 키보다 큰 값의 두 그룹으로 나눔
- 두 그룹에 대해서 각각 다시 퀵 정렬을 수행
- 그룹 내에 값이 하나만 남을 때까지 계속 이 작업을 반복



함수 포인터의 활용 : qsort 함수 (2/2)

❖qsort : 표준 C 라이브러리 함수

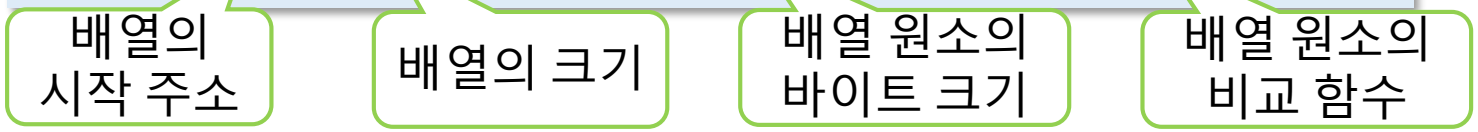
```
void qsort(void *ptr, size_t count, size_t size,
      int(*compare)(const void *, const void *));
```

- ptr : 정렬할 배열의 시작 주소 (기본형 배열, 구조체 배열 등)
- count : 정렬할 배열에 들어있는 원소의 개수
- size : 배열 원소의 바이트 크기
- compare : 배열의 원소들과 키 값을 비교할 때 호출할 함수의 주소
  - 함수 원형이 정해져 있다.
  - e1, e2 : qsort 함수에 인자로 전달된 배열 원소를 가리키는 포인터
  - e1이 가리키는 원소와 e2가 가리키는 원소를 비교해서 e1 > e2면 0보다 큰 값 리턴, e1 < e2면 0보다 작은 값 리턴, e1 == e2면 0 리턴

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

qsort 함수 : int 배열의 정렬

```
qsort(arr, ARR_SIZE, sizeof(arr[0]), compare_int);
```




```c
// int 배열의 원소를 비교하는 함수
int compare_int(const void *e1, const void *e2)
{
    // e1, e2는 int의 주소이므로 const int*형으로 형 변환해서 사용한다.
    const int *p1 = (const int*)e1;
    const int *p2 = (const int*)e2;
    return (*p1 - *p2);
}
```

예제 : qsort 함수를 이용한 int 배열의 정렬

```c
#include <stdlib.h>
#include <time.h>

#define ARR_SIZE 10

int compare_int(const void *e1, const void *e2);
void print_array(const int arr[], int size);      // 배열 원소를 출력하는 함수

int main(void)
{
    int arr[ARR_SIZE] = { 0 };
    int i;

    srand((unsigned int)time(NULL));

    // 배열을 0~99 사이의 임의의 정수로 채운다.
    for (i = 0; i < ARR_SIZE; i++)
        arr[i] = rand() % 100;

    puts("<< 정렬 전 >>");
    print_array(arr, ARR_SIZE);

    qsort(arr, ARR_SIZE, sizeof(arr[0]), compare_int);

    puts("<< 정렬 후 >>");
    print_array(arr, ARR_SIZE);

    return 0;
}
```

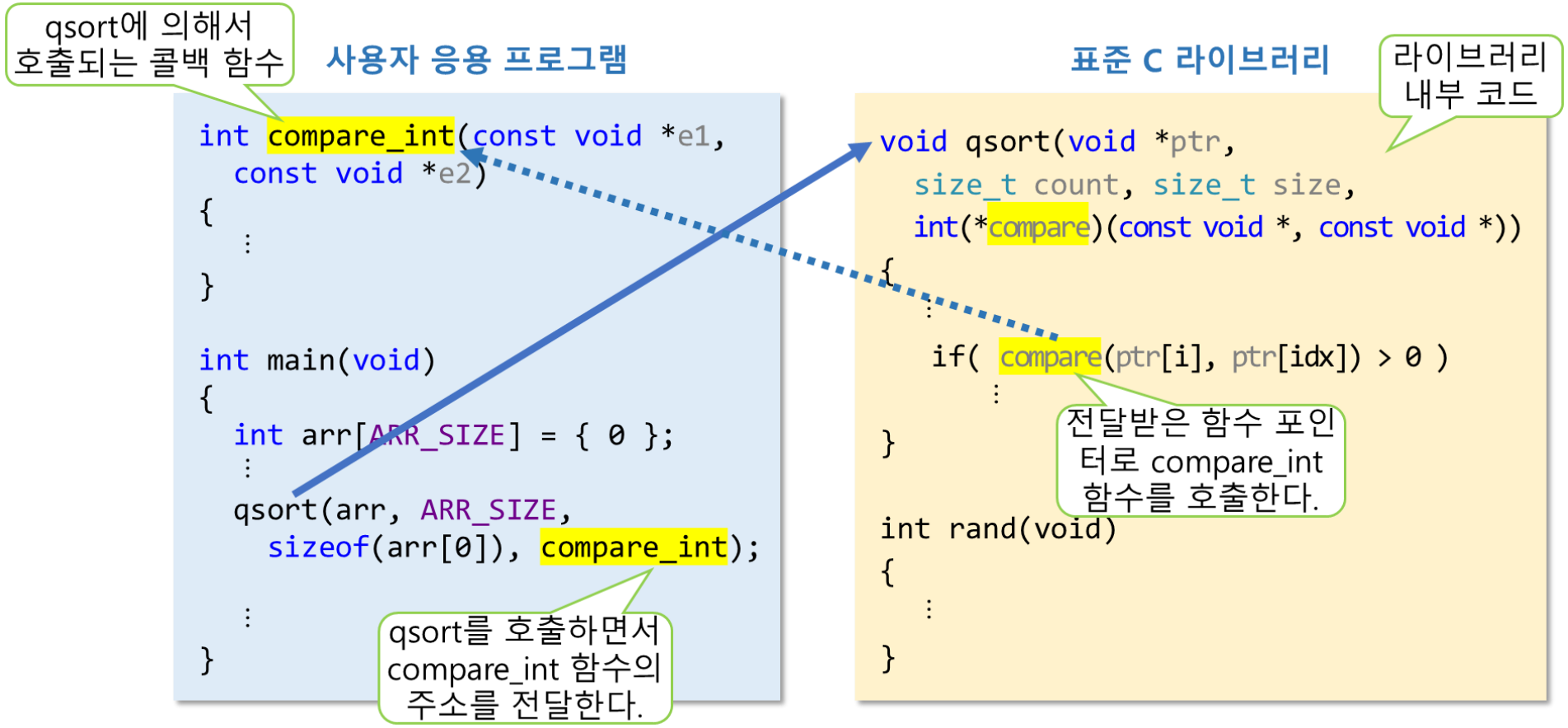
```c
// int 배열의 원소를 비교하는 함수
int compare_int(const void *e1, const void *e2)
{
    // e1, e2는 int의 주소이므로 const int*형으로 형 변환해서 사용한다.
    const int *p1 = (const int*)e1;
    const int *p2 = (const int*)e2;
    return (*p1 - *p2);
}

void print_array(const int arr[], int size)
{
    int  i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

실행결과

```
<< 정렬 전 >>
70 8 32 92 51 62 61 80 91 16
<< 정렬 후 >>
8 16 32 51 61 62 70 80 91 92
```

콜백(callback) 함수

- 프로그래머가 정의한 함수의 주소를 라이브러리 함수를 호출할 때 전달해서 특정 조건일 때 호출하도록 등록



qsort 함수 : CONTACT 구조체 배열의 정렬

```
qsort(arr, size, sizeof(CONTACT), compare_by_name);
```

구조체 배열의 시작 주소

```
// 이름 순 정렬
int compare_by_name(const void *e1, const void *e2)
{
    // e1, e2는 CONTACT의 주소이므로 const CONTACT*형으로 형 변환
    const CONTACT *p1 = (const CONTACT*)e1;
    const CONTACT *p2 = (const CONTACT*)e2;
    return strcmp(p1->name, p2->name);
}
```

예제 : CONTACT 구조체 배열의 정렬

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define STR_SIZE 20

typedef struct contact
{
    char name[STR_SIZE];
    char phone[STR_SIZE];
    int ringtone;
} CONTACT;

int compare_by_name(const void *e1, const void *e2);
void print_contacts(const CONTACT *arr, int size);
int main(void)
{
    CONTACT arr[] = {        // 초기화된 배열
        {"김석진", "01011112222", 0},
        {"전정국", "01012345678", 1},
        {"박지민", "01077778888", 2},
        {"김남준", "01098765432", 9},
        {"민윤기", "01011335577", 5},
        {"정호석", "01024682468", 7},
        {"김태형", "01099991111", 3}
    };
    int size = sizeof(arr) / sizeof(arr[0]);      // 배열의 크기

    puts("<< 정렬 전 >>");
    print_contacts(arr, size);

    puts("<< 이름 순 정렬 >>");
    qsort(arr, size, sizeof(CONTACT), compare_by_name);
    print_contacts(arr, size);

    return 0;
}

// 이름 순 정렬
int compare_by_name(const void *e1, const void *e2)
{
    // e1, e2는 CONTACT의 주소이므로 const CONTACT*형으로 형 변환
    const CONTACT *p1 = (const CONTACT*)e1;
    const CONTACT *p2 = (const CONTACT*)e2;

    return strcmp(p1->name, p2->name);
}
void print_contacts(const CONTACT *arr, int size)
{
    int i;

    printf(" 이름   전화번호   벨\n");
    for (i = 0; i < size; i++)
    {
        printf("%6s %11s %d\n",
            arr[i].name, arr[i].phone, arr[i].ringtone);
    }
}
```

실행결과

```
<< 정렬 전 >>
 이름   전화번호   벨
김석진 01011112222 0
전정국 01012345678 1
박지민 01077778888 2
김남준 01098765432 9
민윤기 01011335577 5
정호석 01024682468 7
김태형 01099991111 3
<< 이름 순 정렬 >>
 이름   전화번호   벨
김남준 01098765432 9
김석진 01011112222 0
김태형 01099991111 3
민윤기 01011335577 5
박지민 01077778888 2
전정국 01012345678 1
정호석 01024682468 7
```

질의 및 응답

참고문헌

- 천정아,『Core C Programming』, 연두에디션(2019)
- C가 보이는 그림책, ANK Co., Ltd. , 성안당 (2018)
- Greg Perry, Dean Miller『어서와 C언어는 처음이지』, 천인국 옮김, 인피니티북스(2015)
- KELLEY (역 : 김명호 외),『A Book on C』, 홍릉과학출판사 (2003)
- 윤성우,『 열혈 C 프로그래밍』, 오렌지미디어
- 천인국,『 쉽게 풀어쓴 C언어 Express』, 생능출판사
- 서현우,『 뇌를 자극하는 C 프로그래밍』, 한빛미디어
- 강성수,『 쾌도난마 C프로그래밍』, 북스홀릭
- 고응남,『 C프로그래밍 기초와 응용실습』, 정익사