


제 16장 전처리 및 다중소스 파일

이번 장에서 학습할 내용

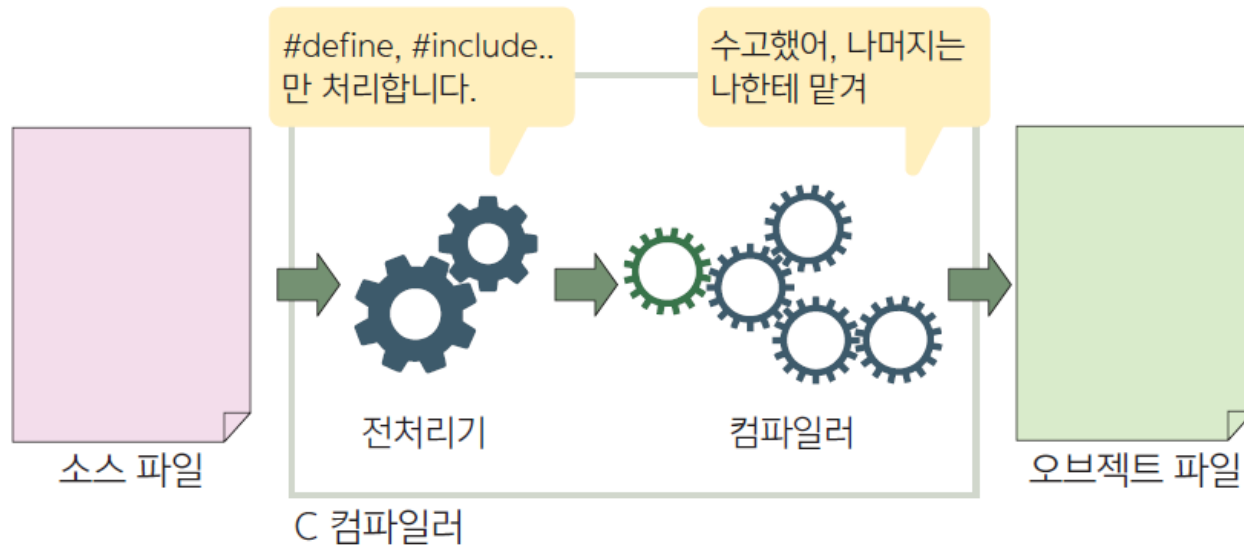
- 
- 전처리 지시어
 - 분할 컴파일
 - 명령어 라인의 매개변수
 - 디버깅 방법

전처리와 기타
중요한 테마에
대하여
학습한다.



전처리기란?

- *전처리기 (preprocessor)*는 컴파일하기에 앞서서 소스 파일을 처리하는 컴파일러의 한 부분



전처리기의 요약

지시어	의미
#define	매크로 정의
#include	파일 포함
#undef	매크로 정의 해제
#if	조건이 참일 경우
#else	조건이 거짓일 경우
#endif	조건 처리 문장 종료
#ifdef	매크로가 정의되어 있는 경우
#ifndef	매크로가 정의되어 있지 않은 경우
#line	행번호 출력
#pragma	시스템에 따라 의미가 다름

단순 매크로

Syntax

단순매크로 정의

예

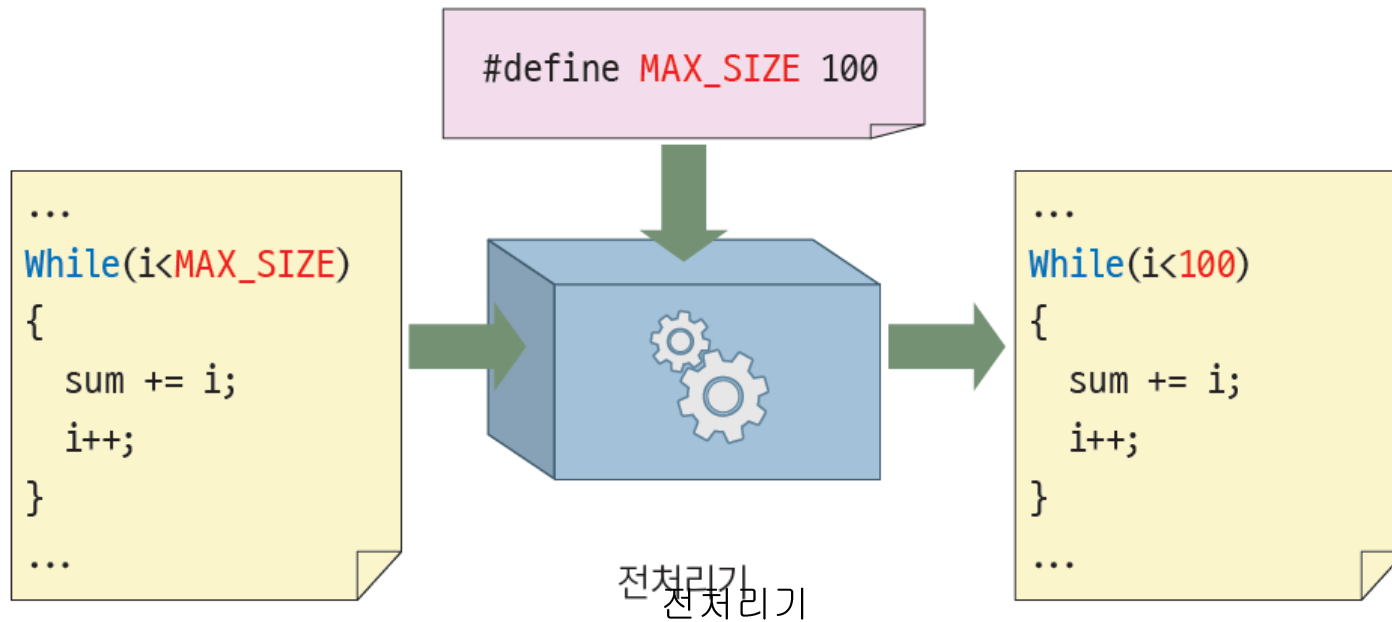
```
#define MAX_SIZE 100
```

기호상수 MAX_SIZE를
100으로 정의한다.

100보다는 MAX_SIZE가
이해하기 쉽지..



단순 매크로



단순 매크로의 장점

- 프로그램의 가독성을 높인다.
- 상수의 변경이 용이하다.

```
#define MAX_SIZE 100  
for(i=0;i<MAX_SIZE;i++)  
{  
    f += (float) i/MAX_SIZE;  
}
```



```
#define MAX_SIZE 200  
for(i=0;i<MAX_SIZE;i++)  
{  
    f += (float) i/MAX_SIZE;  
}
```

단순 매크로의 예

```
#define PI      3.141592          // 원주율
#define EOF     (-1)              // 파일의 끝표시
#define EPS     1.0e-9            // 실수의 계산 한계
#define DIGITS  "0123456789"     // 문자 상수 정의
#define BRACKET "(){[]"          // 문자 상수 정의
#define getchar()  getc(stdin)    // stdio.h에 정의
#define putchar()  putc(stdout)   // stdio.h에 정의
```


예제

- #define 지시자를 사용하여 연산자 &&를 AND로 바꾸어서 사용해보자.

```
int i = 0;  
  
while( i < n AND list[i] != key )  
    i++;  
if( i IS n )  
    return -1;  
else  
    return i;
```

어떻게 전처리를
정의해야 할까요?



예제

```
#include <stdio.h>
#define AND      &&
#define OR      ||
#define NOT      !
#define IS      ==
#define ISNOT    !=

int search(int list[], int n, int key)
{
    int i = 0;

    while( i < n AND list[i] != key )
        i++;
    if( i IS n )
        return -1;
    else
        return i;
}
```

예제

```
int main(void)
{
    int m[] = { 1, 2, 3, 4, 5, 6, 7 };
    printf("배열에서 5의 위치=%d\n", search(m, sizeof(m) / sizeof(m[0]), 5));
    return 0;
}
```

배열에서 5의 위치=4

함수 매크로

- **함수 매크로**(*function-like macro*)란 매크로가 함수처럼 매개 변수를 가지는 것

Syntax

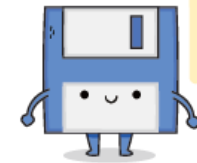
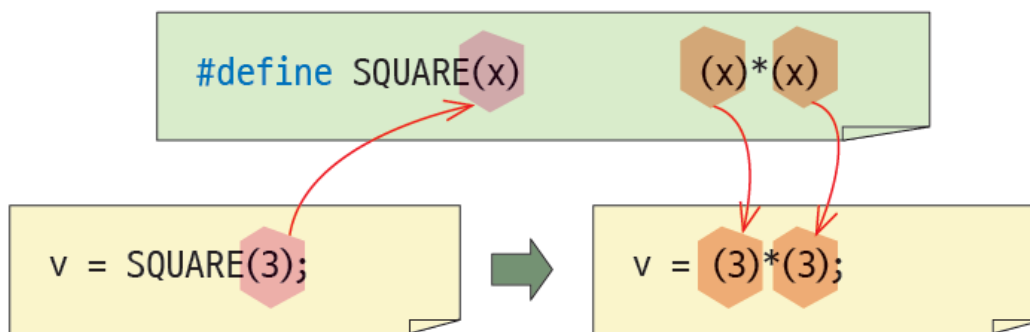
함수매크로

예

매크로 인수 SQUARE(X)는 이것과 같다.

```
#define SQUARE(x) ((x) * (x))
```

함수 매크로



전처리기

SQURAE(x)를 (x)*(x)로
변환하라는 거구나!

함수 매크로의 예

```
#define SUM(x, y)      ((x) + (y))  
#define AVERAGE(x, y, z) (( (x) + (y) + (z) ) / 3 )  
#define MAX(x,y)      ( (x) > (y) ) ? (x) : (y)  
#define MIN(x,y)      ( (x) < (y) ) ? (x) : (y)
```

주의할 점

```
#define SQUARE(x)  x*x // 위험 !!
```

```
v = SQUARE(a+b);
```



```
v = a + b*a + b;
```



함수
매크로에서는
매개 변수를
괄호로
둘러싸는 것이
좋습니다.

```
#define SQUARE(x) (x)*(x)
```

```
// 올바른 형태
```


주의할 점

매크로 사용시 주의할 점

① 매크로를 정의할 때 매개 변수는 모두 사용되어야 한다.

```
#define HALFOF(y, x) ((x) / 2) // 오류!!
```

② 매크로 이름과 괄호 사이에 공백이 있으면 안 된다.

```
#define ADD (x, y) ((x) + (y)) // 오류!!
```

ADD와 (사이에 공백이 있기 때문에 전처리기는 기호 상수 정의로 생각하고 ADD라는 문자열을 (x, y) ((x) + (y))로 치환한다.



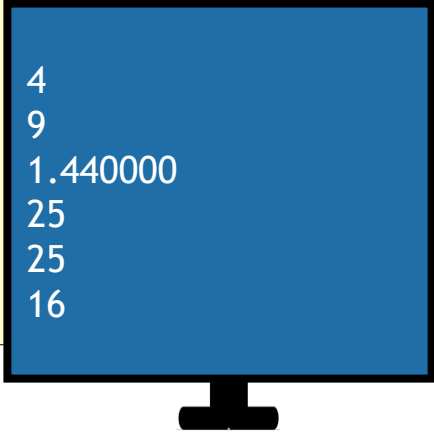
예제 #1

```
// 매크로 예제
#include <stdio.h>
#define SQUARE(x) ((x) * (x))

int main(void)
{
    int x = 2;

    printf("%d\n", SQUARE(x));
    printf("%d\n", SQUARE(3));
    printf("%f\n", SQUARE(1.2)); // 실수에도 적용 가능
    printf("%d\n", SQUARE(x+3));
    printf("%d\n", 100/SQUARE(x));
    printf("%d\n", SQUARE(++x)); // 논리 오류

    return 0;
}
```



4
9
1.440000
25
25
16

내장 매크로

- 내장 매크로: 미리 정의된 매크로

내장 매크로	설명
__DATE__	이 매크로를 만나면 현재의 날짜(월 일 년)로 치환된다.
__TIME__	이 매크로를 만나면 현재의 시간(시:분:초)으로 치환된다.
__LINE__	이 매크로를 만나면 소스 파일에서의 현재의 라인 번호로 치환된다.
__FILE__	이 매크로를 만나면 소스 파일 이름으로 치환된다.

```
printf("컴파일 날짜=%s\n", __DATE__);  
printf("치명적 에러 발생 파일 이름=%s 라인 번호= %d\n", __FILE__, __LINE__);
```



```
컴파일 날짜=Aug 23 2021  
치명적 에러 발생 파일 이름=C:\Users\Wkim\source\repos\Project14\Project14\소스.c  
라인 번호= 6
```

Lab: ASSERT 매크로

- 프로그램을 디버깅할 때 자주 사용되는 ASSERT 매크로를 작성해보자.



가정(sum == 0)이 소스 파일
C:\Users\chun\source\repos\Project21\Project21\macro4.c
12번째 줄에서 실패.

예제: ASSERT 매크로

```
#include <stdio.h>

#define ASSERT(exp)      { if (!(exp)) \
    { printf("가정(" #exp ")이 소스 파일 %s %d번째 줄에서 실패.\n", \
    __FILE__, __LINE__), exit(1);}}

int main(void)
{
    int sum=100;                // 지역 변수의 초기값은 0이 아님

    ASSERT(sum == 0);           // sum의 값은 0이 되어야 함.
    return 0;
}
```

매크로를 다음 줄로 연장
할 때 사용

가정(sum == 0)이 소스 파일 c:\user\igchun\documents\visual studio 2017\projects
12번째 줄에서 실패

함수 매크로와 함수

- 장점은 함수 매크로는 함수에 비하여 수행 속도가 빠르다는 것이다. 매크로는 호출이 아니라 코드가 그 위치에 삽입되는 것이기 때문에 함수 호출의 복잡한 단계를 거칠 필요가 없다.
- 코드의 길이를 어느 한도 이상 길게 할 수 없다. 많은 경우 한 줄이고 두세 줄 까지가 한계이다.
- 매크로를 사용하면 소스 파일의 크기가 커진다.

#ifdef

Syntax

조건부 컴파일

매크로 `DEBUG`가 정의되어 있으면 `#if`와 `#endif` 사이에 있는 모든 문장들을 컴파일한다.

예

```
#ifdef DEBUG
    printf("value=%d\n", value);
#endif
```

#ifdef의 예

```
int average(int x, int y)
{
    printf("x=%d, y=%d\n", x, y);

    return (x+y)/2;
}
```

```
int average(int x, int y)
{
    #ifdef DEBUG
        printf("x=%d, y=%d\n", x, y);
    #endif
    return (x+y)/2;
}
```

DEBUG가 선언되었을
때만 출력문을
포함시킵니다.



매크로 선언 위치

```
#define DEBUG
```

```
int average(int x, int y) {
```

컴파일에 포함

```
#ifdef DEBUG
    printf("x=%d, y=%d\n", x, y);
#endif
```

```
    return (x+y)/2;
```

```
}
```

```
int average(int x, int y) {
```

컴파일에 포함되지
않음

```
#ifdef DEBUG
    printf("x=%d, y=%d\n", x, y);
#endif
```

```
    return (x+y)/2;
```

```
}
```

Lab: 리눅스 버전과 윈도우 버전 분리

- 예를 들면 어떤 회사에서 리눅스와 윈도우즈 버전의 프로그램을 개발하였다고 하자



리눅스 버전입니다.

예제

```
#include <stdio.h>
```

```
#define LINUX
```

```
int main(void)
```

```
{
```

```
#ifdef LINUX
```

```
    printf("리눅스 버전입니다. \n");
```

```
#else
```

```
    printf("윈도우 버전입니다. \n");
```

```
#endif
```

```
    return 0;
```

```
}
```

LINUX 버전

WINDOWS 버전

#ifndef, #undef

- #ifndef
 - 어떤 매크로가 정의되어 있지 않으면 컴파일에 포함된다.

```
#ifndef LIMIT
#define LIMIT 1000
#endif
```

LIMIT가 정의되어 있지 않으면

LIMIT를 정의해준다.

- #undef
 - 매크로의 정의를 취소한다.

```
#define SIZE 100
..
#undef SIZE
#define SIZE 200
```

SIZE의 정의를 취소한다.

#if

- 기호가 참으로 계산되면 컴파일
- 조건은 상수이어야 하고 논리, 관계 연산자 사용 가능

Syntax

조건부 컴파일

예

```
#if DEBUG==1  
    printf("value=%d\n", value);  
#endif
```

매크로 DEBUG의 값이 1이면 #if와 #endif 사이에 있는 모든 문장들을 컴파일한다.

#if-#else-#endif

```
#define NATION 1

#if NATION == 1
    printf("안녕하세요?");
#elif NATION == 2
    printf("你好吗?");
#else
    printf("Hello World!");
#endif
```

다양한 예

```
#if (AUTHOR == KIM) // 가능!! KIM은 다른 매크로  
#if (VERSION*10 > 500 && LEVEL == BASIC) // 가능!!  
#if (VERSION > 3.0) // 오류 !! 버전 번호는 300과 같은 정수로 표시  
#if (AUTHOR == "CHULSOO") // 오류 !!
```

다수의 라인을 주석처리

```
#if 0 // 여기서부터 시작하여
void test()
{
/* 여기에 주석이 있다면 코드 전체를 주석 처리하는 것이 쉽지 않다. */
sub();
}
#endif // 여기까지 주석 처리된다.
```


예제

- 정렬 알고리즘을 선택

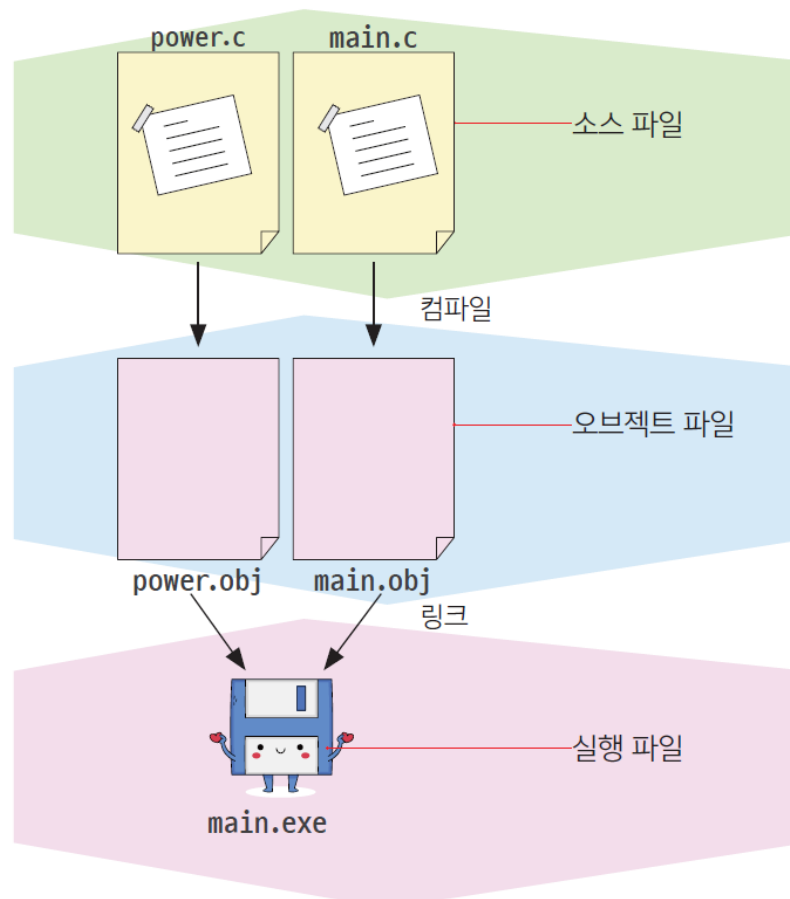
```
#define SORT_METHOD 3

#if (SORT_METHOD == 1)
...    // 선택정렬구현
#elif (SORT_METHOD == 2)
...    // 버블정렬구현
#else
...    // 퀵정렬구현
#endif
```

다중 소스 파일

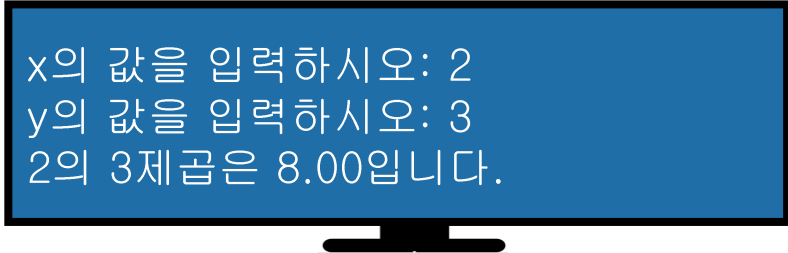
- *단일 소스 파일*
 - 파일의 크기가 너무 커진다.
 - 소스 파일을 다시 사용하기가 어려움
- *다중 소스 파일*
 - 서로 관련된 코드만을 모아서 하나의 소스 파일로 할 수 있음
 - 소스 파일을 재사용하기가 간편함

다중 소스 파일

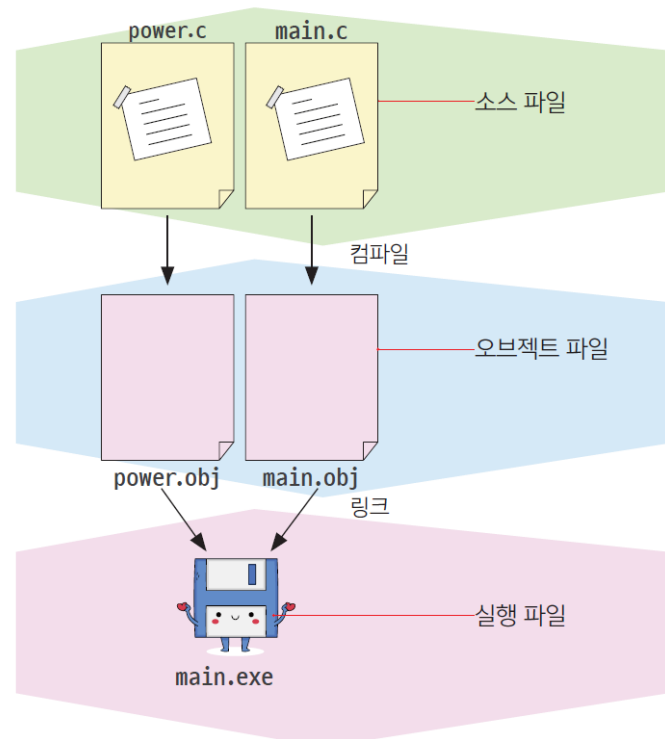


예제:

- 거듭 제곱을 구하는 함수 `power()`를 만들고 이것을 `power.c`에 저장하여 보자. 그리고 `main.c`를 만들고 여기에 `main()` 함수를 정의한 다음, `main()`에서 `power()`를 호출한다.



x의 값을 입력하시오: 2
y의 값을 입력하시오: 3
2의 3제곱은 8.00입니다.



예제

```
// main.c
#include <stdio.h>
#include "power.h"

int main(void)
{
    int x, y;

    printf("x의 값을 입력하시오:");
    scanf("%d", &x);
    printf("y의 값을 입력하시오:");
    scanf("%d", &y);
    printf("%d의 %d 제곱값은 %f\n", x, y, power(x, y));

    return 0;
}
```

예제

```
// power.c
#include "power.h"

double power(int x, int y)
{
    double result = 1.0; // 초기값은 1.0
    int i;

    for (i = 0; i < y; i++)
        result *= x;

    return result;
}
```

예제

```
#pragma once  
// power. h  
  
double power(int x, int y);           // 함수 원형 정의
```

헤더 파일을 사용하지 않으면

```
void draw_line(...)  
{  
    ...  
}  
void draw_rect(...)  
{  
    ...  
}  
void draw_circle(...)  
{  
    ...  
}
```

graphics.c

공급자

함수 원형 정의가 중복되어 있음

```
void draw_line(...);  
void draw_rect(...);  
void draw_circle(...);
```

```
int main(void)  
{  
    draw_rect(...);  
    draw_circle(...);  
    ...  
    return 0;  
}
```

main.c

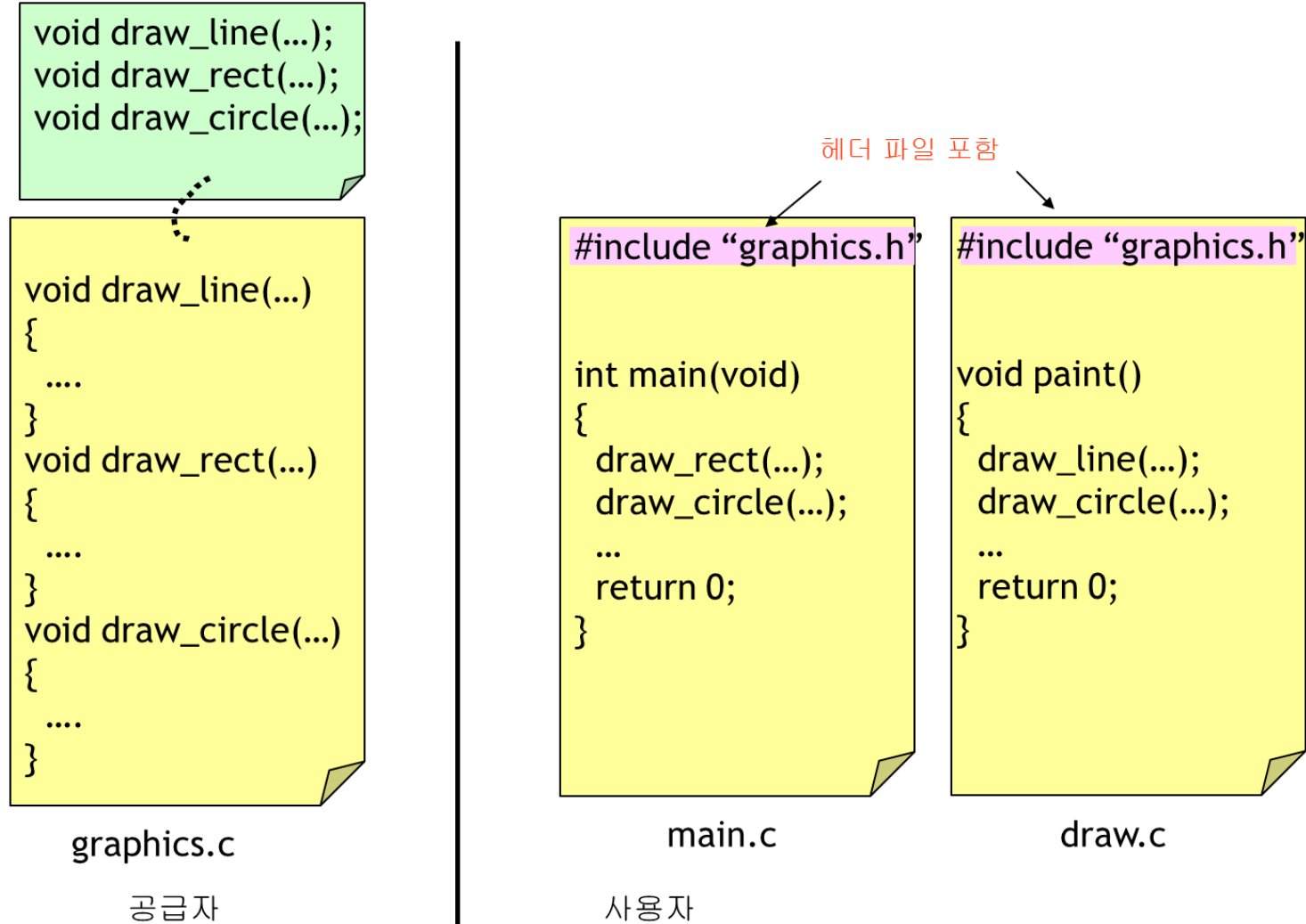
사용자

```
void draw_line(...);  
void draw_rect(...);  
void draw_circle(...);
```

```
void paint()  
{  
    draw_line(...);  
    draw_circle(...);  
    ...  
    return 0;  
}
```

draw.c

헤더 파일을 사용하면



다중 소스 파일에서 외부 변수

외부 소스 파일에 선언된 변수를 사용하려면 extern을 사용한다.

```
double gx, gy;
```

```
int main(void)
{
    gx = 10.0;
    ...
}
```

main.c

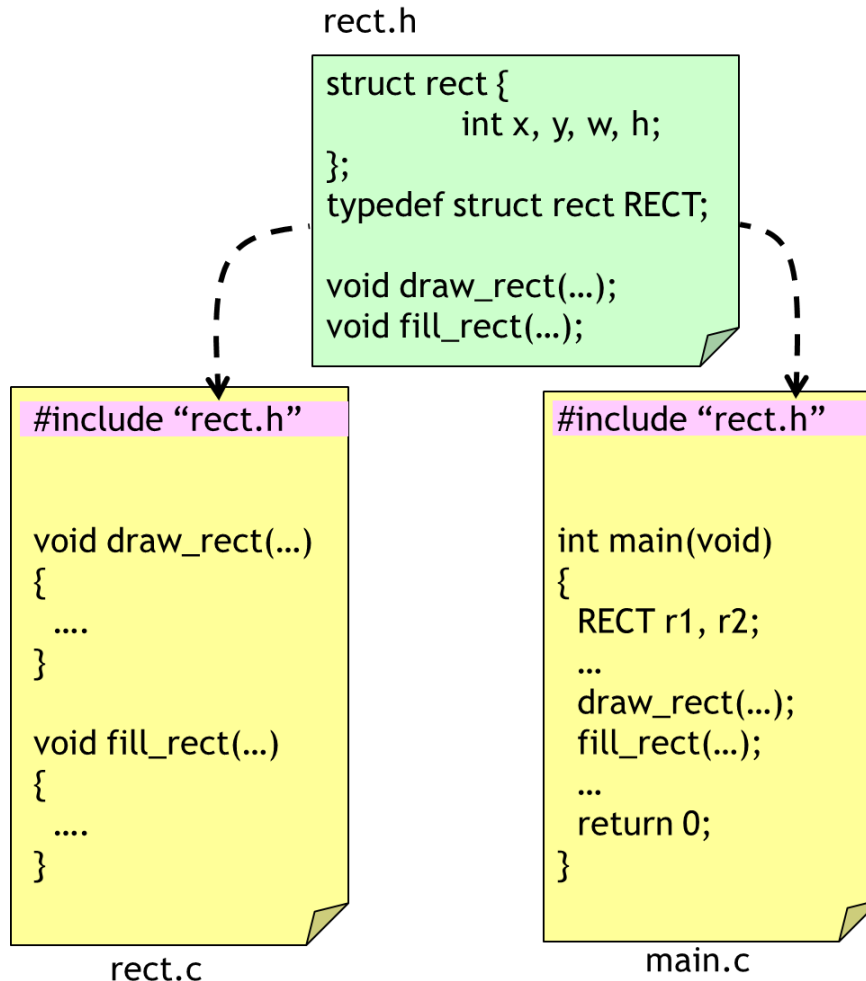
```
extern double gx, gy;
```

```
int power(void)
{
    ...
    result *= gx;
}
```

power.c

예제

- 다음과 같은 프로그램을 다중 소스로 작성해보자.



rect.h

```
#pragma once
struct rect {
    int x, y, w, h;
};

typedef struct rect RECT;

void draw_rect(const RECT*);
double calc_area(const RECT*);
void move_rect(RECT*, int, int);
```

rect.c 1/2

```
#include <stdio.h>
#include "rect.h"
#define DEBUG

void draw_rect(const RECT* r)
{
#ifdef DEBUG
    printf("draw_rect(x=%d, y=%d, w=%d, h=%d) \n", r->x, r->y, r->w, r->h);
#endif
}
```

rect.c 2/2

```
double calc_area(const RECT* r)
{
    double area;
    area = r->w * r->h;
#ifdef DEBUG
    printf("calc_area()=%f \n", area);
#endif
    return area;
}

void move_rect(RECT* r, int dx, int dy)
{
#ifdef DEBUG
    printf("move_rect(%d, %d) \n", dx, dy);
#endif
    r->x += dx;
    r->y += dy;
}
```

main.c

```
#include <stdio.h>
#include "rect.h"

int main(void)
{
    RECT r = { 10,10, 20, 20 };
    double area = 0.0;

    draw_rect(&r);
    move_rect(&r, 10, 20);
    draw_rect(&r);
    area = calc_area(&r);
    draw_rect(&r);
    return 0;
}
```

실행 결과

```
draw_rect(x=10, y=10, w=20, h=20)  
move_rect(10, 20)  
draw_rect(x=20, y=30, w=20, h=20)  
calc_area()=400.00  
draw_rect(x=20, y=30, w=20, h=20)
```


Lab: 헤더파일 중복막기

- 구조체 정의가 들어 있는 헤더 파일을 소스 파일에 2번 포함시키면 컴파일 오류가 발생한다. 이것을 막기 위하여 `#ifndef` 지시어를 사용할 수 있다

```
#ifndef STUDENT_H
#define STUDENT_H

struct STUDENT {
    int number;
    char name[10];
};
#endif
```

소스 파일에서 여러 번 포함시켜도 컴파일 오류가 발생하지 않음

TIP

- 최근의 C언어에서는 다음과 같은 문장을 헤더 파일의 첫 부분에 추가하여도 동일한 효과를 낸다. 비주얼 스튜디오에서 헤더 파일을 추가하면 자동으로 첫 부분에 추가된다.

`#pragma once`

중간 점검

1. 다음 문장의 참 거짓을 말하라. "여러 소스 파일을 이용하는 것보다 하나의 소스 파일로 만드는 편이 여러모로 유리하다."
2. 팩토리얼을 구하는 함수가 포함된 소스 파일과 관련 헤더 파일을 제작하여 보자.
3. 2차원 공간에서 하나의 점을 나타내는 point 구조체를 정의하는 헤더 파일을 작성하여 보자.

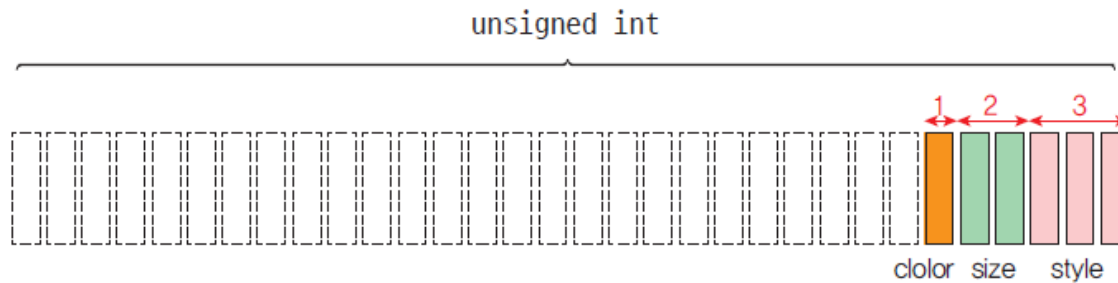


비트 필드 구조체

- 멤버가 비트 단위로 나누어져 있는 구조체

```
struct 태그이름 {  
    자료형 멤버이름1: 비트수;  
    자료형 멤버이름2: 비트수;  
    ...  
};
```

```
struct product {  
    unsigned style : 3;  
    unsigned size : 2;  
    unsigned color : 1;  
};
```



bit_field.c

```
// 비트 필드 구조체
#include <stdio.h>

struct product {
    unsigned style : 3;
    unsigned size : 2;
    unsigned color : 1;
};

int main(void)
{
    struct product p1;

    p1.style = 5;
    p1.size = 3;
    p1.color = 1;

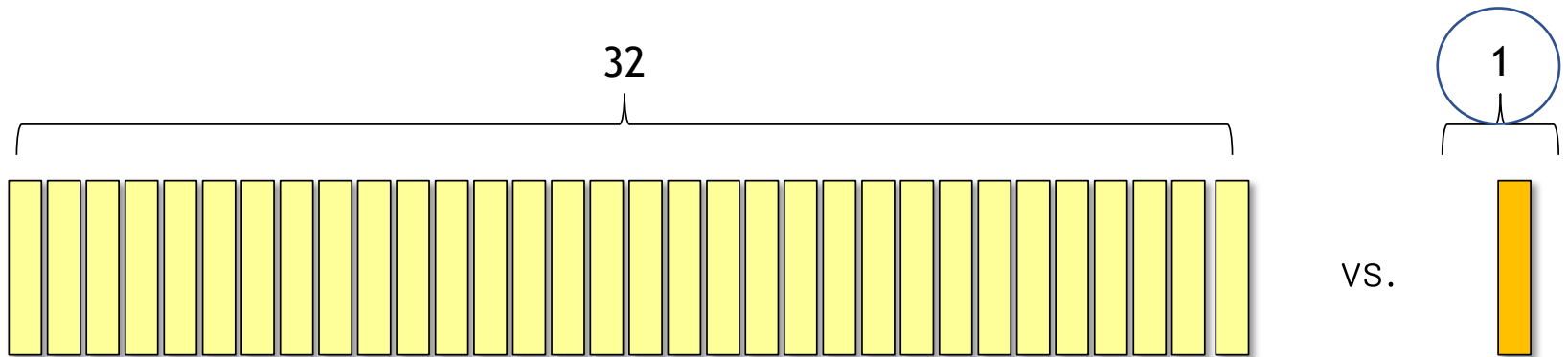
    printf("style=%d size=%d color=%d\n", p1.style, p1.size, p1.color);
    printf("sizeof(p1)=%d\n", sizeof(p1));
    printf("p1=%x\n", p1);

    return 0;
}
```

style=5 size=3 color=1
sizeof(p1)=4
p1=ccccccfd

비트 필드의 장점

- 메모리가 절약된다.
 - ON 또는 OFF의 상태만 가지는 변수를 저장할 때 32비트의 int형 변수를 사용하는 것보다는 1비트 크기의 비트 필드를 사용하는 편이 훨씬 메모리를 절약한다.



Q & A

