



# 쉽게 풀어쓴 C언어 Express

## 제15장 전처리 및 비트연산





# 이번 장에서 학습할 내용

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

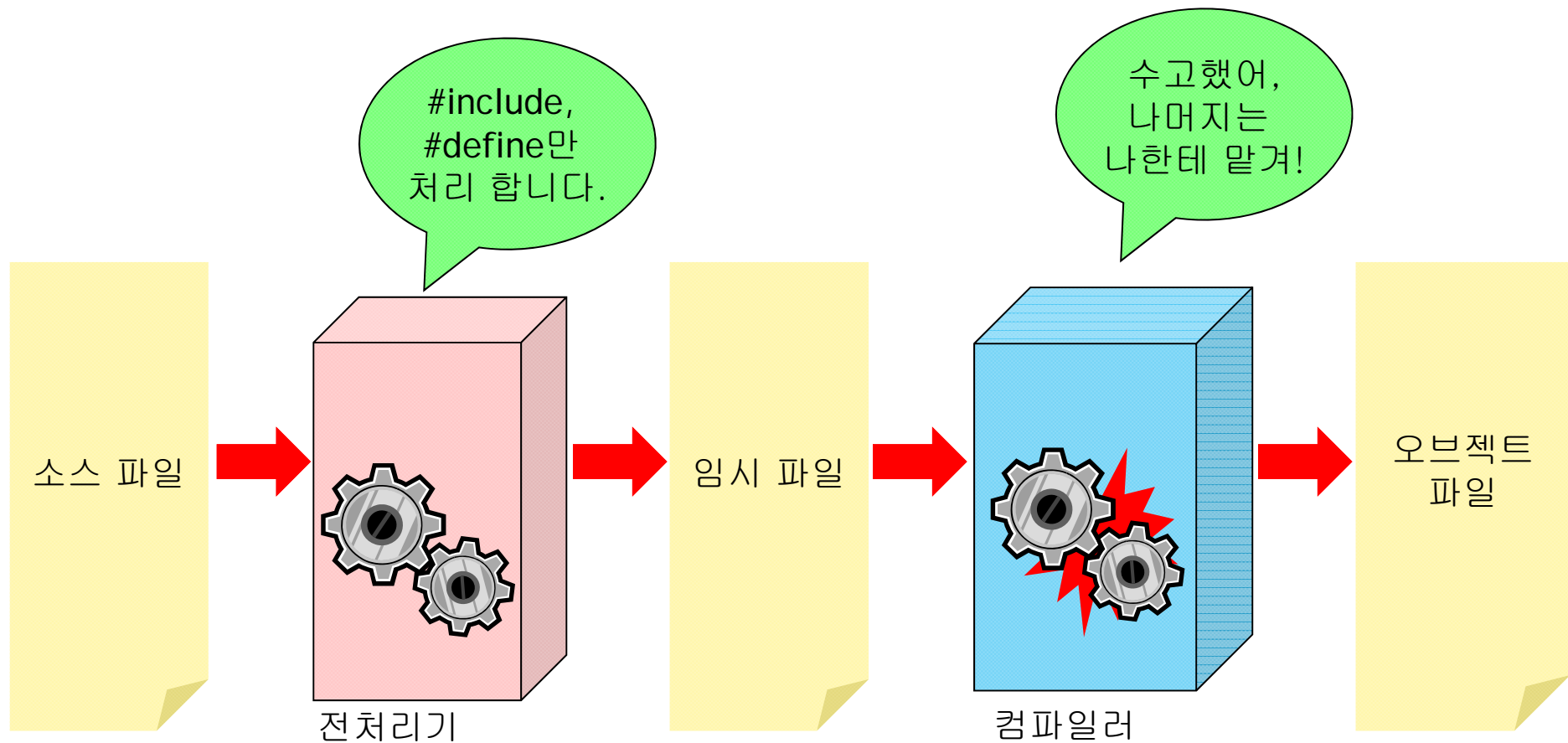
전처리와 기타  
중요한 테마에  
대하여  
학습한다.





# 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

- 단순 매크로(macro): 숫자 상수를 기호 상수로 만든 것
- (예)

```
#define MAX_SIZE 100  
#define PI 3.141592  
#define EPS 1.0e-9
```

100보다는  
MAX\_SIZE가  
이해하기 쉽  
지..

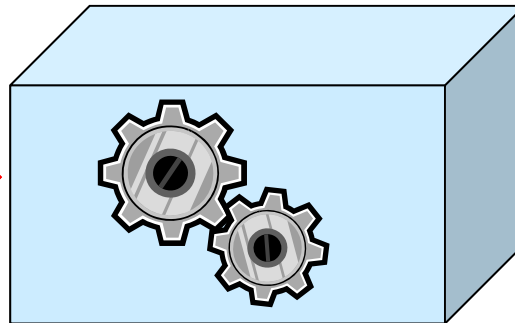




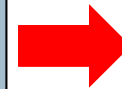
# 단순 매크로

```
#define MAX_SIZE 100
```

```
...  
while(i<MAX_SIZE)  
{  
    sum += i;  
    i++;  
}  
...
```



전처리기



```
...  
while(i<100)  
{  
    sum += i;  
    i++;  
}  
...
```



# 단순 매크로의 장점

- 프로그램의 가독성을 높인다.
- 상수의 변경이 용이하다.

```
#include <stdio.h>

int main(void)
{
    ...
    won1 = 1120 * dollar1;
    won2 = 1120 * dollar2;
    ...
}
```

Diagram showing the value 1050 being used to replace the literal 1120 in the code. Red arrows point from the 1050 in the green boxes to the 1120 in the code lines.

리터럴 상수를 사용하는 경우:  
등장하는 모든 곳을  
수정하여야한다.

```
#include <stdio.h>
#define EXCHANGE_RATE 1120

int main(void)
{
    ...
    won1 = EXCHANGE_RATE * dollar1;
    ...
    won2 = EXCHANGE_RATE * dollar2;
    ...
}
```

Diagram showing the value 1050 being used to replace the macro EXCHANGE\_RATE in the code. A red arrow points from the 1050 in the green box to the EXCHANGE\_RATE in the code lines.

기호 상수를 사용하는 경우:  
기호 상수가 정의된 곳만 수정  
하면 된다.



# 단순 매크로의 예

```
#define PI          3.141592          // 원주율
#define TWOPI       (3.141592 * 2.0)  // 원주율의 2배
#define MAX_INT     2147483647        // 최대정수
#define EOF         (-1)              // 파일의 끝표시
#define MAX_STUDENTS 2000             // 최대 학생수
#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
```

214748364  
7보다는  
MAX\_INT  
가 낮죠



사람은  
숫자보다  
기호를 잘  
기억합니다.







# 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;
}

int main(void)
{
    int m[] = { 1, 2, 3, 4, 5, 6, 7 };

    printf("%d\n", search(m, sizeof(m)/sizeof(m[0]), 5));
    return 0;
}
```



C프로그래밍을  
다른 언어처럼  
작성할 수 있습니다.



## 중간 점검

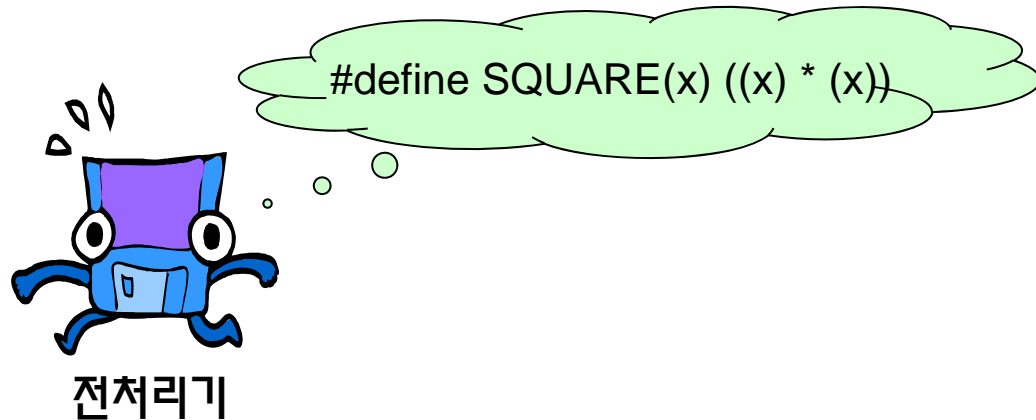
1. `#define`을 이용하여서 `1234`를 `KEY`로 정의하여 보라.
2. `#define`을 이용하여서 `scanf`를 `INPUT`으로 정의하여 보라.



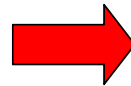


# 함수 매크로

- 함수 매크로(*function-like macro*)란 매크로가 함수처럼 매개 변수를 가지는 것
- (예) `#define SQUARE(x) ((x) * (x))`



`v = SQUARE(3);`



`v = ((3) * (3));`



## 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
```



# 함수 매크로의 장단점

- 함수 매크로의 장단점
  - 함수 호출 단계가 필요없어 실행 속도가 빠르다.
  - 소스 코드의 길이가 길어진다.
- 간단한 기능은 매크로를 사용
  - `#define MIN(x, y) ((x) < (y) ? (x) : (y))`
  - `#define ABS(x) ((x) > 0 ? (x) : -(x))`
- 매크로를 한줄 이상 연장하는 방법
  - `#define PRINT(x) if( debug==1 && \`
  - `mode==1 ) \`
  - `printf("%d", x);`



# 예제

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

```
int main(void)
{
```

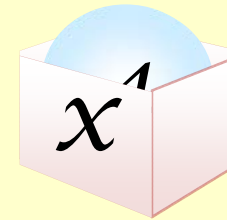
```
    int x = 2;
```

```
    printf("%d\n", SQUARE(x));
    printf("%d\n", SQUARE(3));
    printf("%f\n", SQUARE(1.2));
    printf("%d\n", SQUARE(x+3));
    printf("%d\n", 100/SQUARE(x));
    printf("%d\n", SQUARE(++x));
```

```
    return 0;
```

```
}
```

$((++x) * (++x))$



// 실수에도 적용 가능

// 논리 오류

$(4) * (4) = 16$

4  
9  
1.440000  
25  
25  
16



## # 연산자

- PRINT(x)와 같이 호출하면  
와 같이 출력하는 매크로 작성



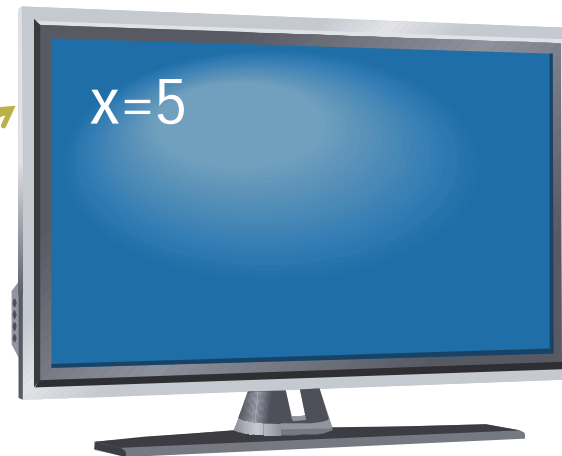
- 다음과 같이 작성하면 잘못된 결과가 나온다.
- `#define PRINT(exp) printf("exp=%d\n", exp);`







- **#**은 문자열 변환 연산자(Stringizing Operator)라고 불린다. 매크로 정의에서 매개 변수 앞에 **#**가 위치하면 매크로 호출에 의하여 전달되는 실제 인수는 큰따옴표로 감싸지고 문자열로 변환된다.
- `#define PRINT(exp) printf("#exp" = %d\n",exp);`
- `PRINT(x);`





## ## 연산자

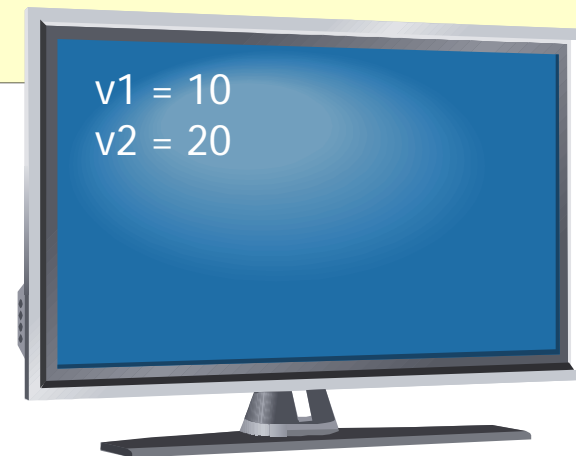
- ## 연산자는 토큰 병합 연산자 (token-pasting operator)
- `#define MAKE_NAME(n) v ## n`
- `MAKE_NAME(1)`과 같이 호출된다고 가정하자.
- 매개 변수 `n`은 `1`로 치환되고 ## 연산자에 의하여 `v`와 `1`이 합쳐져서 하나의 토큰 `v1`이 된다.



## 예제

```
#include <stdio.h>
#define MAKE_NAME(n) v ## n
#define PRINT(n) printf("v" #n " = %d\n", v ## n);

int main(void)
{
    int MAKE_NAME(1) = 10;    //
    int MAKE_NAME(2) = 20;
    PRINT(1);                 // printf("v1 = %d\n", v1);과 같다.
    PRINT(2);                 // printf("v2 = %d\n", v2);과 같다.
    return 0;
}
```





# 내장 매크로

- 내장 매크로: 미리 정의된 매크로

내장 매크로	설명
__DATE__	이 매크로를 만나면 <b>현재의 날짜</b> (월 일 년)로 치환된다.
__TIME__	이 매크로를 만나면 <b>현재의 시간</b> (시:분:초)으로 치환된다.
__LINE__	이 매크로를 만나면 소스 파일에서의 <b>현재의 라인 번호</b> 로 치환된다.
__FILE__	이 매크로를 만나면 <b>소스 파일 이름</b> 으로 치환된다.

📌 `printf("컴파일 날짜=%s\n", __DATE__);`

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



## 예제: ASSERT 매크로

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

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

```
int main(void)\n{\n
```

```
    int sum;                // 지역 변수의 초기값은 0이 아님
```

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

```
    return 0;\n}
```

매크로를 다음 줄로 연장  
할 때 사용



가정(sum == 0)이 소스 파일 c:\source\chapter15\macro3\macro3\macro3.c 11번째 줄에서 실패.



# 비트 관련 매크로

- 매크로들은 변수를 받아서 특정 비트값을 반환하거나 설정한다.
- `GET_BIT()`는 변수 `w`에서 `k`번째 비트의 값을 0 또는 1로 반환한다.
- `#define GET_BIT(w, k) (((w) >> (k)) & 0x01)`
- `SET_BIT_ON()`는 변수 `w`의 `k`번째 비트를 1로 설정하는 매크로이다.  
`#define SET_BIT_ON(w, k) ((w) |= (0x01 << (k)))`
- `SET_BIT_OFF()`는 변수 `w`의 `k`번째 비트를 0로 설정하는 매크로이다.
- `#define SET_BIT_OFF(w, k) ((w) &= ~(0x01 << (k)))`



## 예제: ASSERT 매크로

```
#include <stdio.h>

#define GET_BIT(w, k) (((w) >> (k)) & 0x01)
#define SET_BIT_ON(w, k) ((w) |= (0x01 << (k)))
#define SET_BIT_OFF(w, k) ((w) &= ~(0x01 << (k)))

int main(void)
{
    int data=0;
    SET_BIT_ON(data, 2);
    printf("%08X\n", data);
    printf("%d\n", GET_BIT(data, 2));

    SET_BIT_OFF(data, 2);
    printf("%08X\n", data);
    printf("%d\n", GET_BIT(data, 2));
    return 0;
}
```



```
00000004
1
00000000
0
```



## 중간 점검

1. 함수 매크로와 함수 중에서 속도 면에서 유리한 것은?
2. 주어진 수의 3제곱을 수행하는 함수 매크로를 정의하여 보자.







# #ifdef

- 어떤 조건이 만족되었을 경우에만 컴파일하는 조건부 컴파일 지시

**#ifdef** 매크로

문장1                    **// 매크로가 정의되었을 경우**

...

**#else**

문장2                    **// 매크로가 정의되지 않았을 경우**

...

**#endif**



# #ifdef의 예

```
#define DEBUG
```

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

```
{
```

```
#ifdef DEBUG
```

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

```
#endif
```

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

```
}
```

컴파일에 포함

```
//#define DEBUG
```

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

```
{
```

```
#ifdef DEBUG
```

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

```
#endif
```

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

```
}
```

주석 처리

컴파일에 포함  
되지 않음



# 예제

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

int main(void)
{
#ifdef DELUXE
    printf("딜럭스 버전입니다. \n");
#endif
    return 0;
}
```



딜럭스 버전입니다.



# 예제

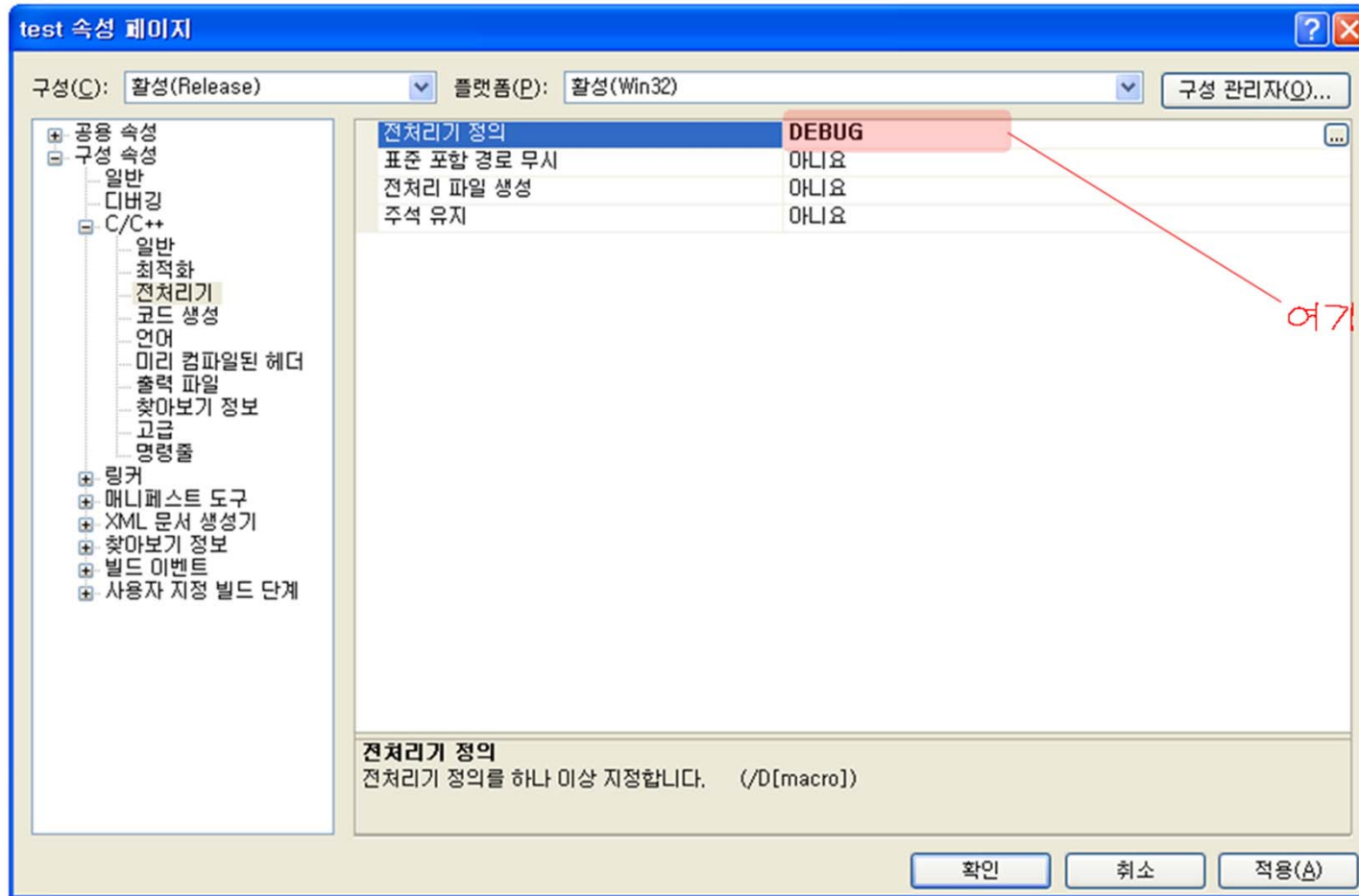
```
#include <stdio.h>
#define LINUX
int main(void)
{
#ifdef LINUX
    ...
#else
    ...
#endif
    return 0;
}
```

LINUX 버전

WINDOWS 버전



# Visual C++에서 설정하는 방법





# #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.



## 중간 점검

1. 전처리기 지시자 `#ifdef`을 사용하여 TEST가 정의되어 있는 경우에만 화면에 “TEST”라고 출력하는 문장을 작성하여 보자.





# #if

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

**형식**    `#if` 조건  
          문장들  
          `#endif`

**설명**    매크로의 값이 참이면 `#if`와 `#endif`사이에 있는 모든 문장들을 컴파일한다.

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





# #if-#else-#endif

형식

```
#if 조건1
    문장1
#elif 조건2
    문장2
#else
    문장3
#endif
```

- (예)

```
#if NATION == 1
#include "korea.h"
#elif NATION == 2
#include "china.h"
#else
#include "usa.h"
#endif
```



## 다양한 예

- `#if (VERSION > 3)`    `// 가능! 버전이 3 이상이면 컴파일`
- ...
- `#endif`
  
- `#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 !!`
  
- `#if (VERSION > 300 || defined(DELUXE) )`



# 조건부 컴파일을 이용하는 디버깅

```
#define DEBUG          1

...
#if DEBUG == 1
printf("현재 counter의 값은 %d입니다.\n", counter);
#endif
```



# 조건부 컴파일을 이용하는 디버깅

```
#define DEBUG
```

```
...
```

```
#ifdef DEBUG
```

```
printf("현재 counter의 값은 %d입니다.\n", counter);
```

```
#endif
```

```
...
```

```
#if defined(DEBUG)
```

```
printf("현재 counter의 값은 %d입니다.\n", counter);
```

```
#endif
```



# 다수의 라인을 주석처리

```
#if 0  // 여기서부터 시작하여  
...  
...  
#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
```



## 중간 점검

1. **#if**를 사용하여 **DEBUG**가 2일 경우에만 “**DEBUG**”가 나오도록 문장을 작성하라.
2. **#if**를 사용하여 **DEBUG**가 2이고 **LEVEL**이 3인 경우에만 “**DEBUG**”가 나오도록 문장을 작성하라.





# 헤더 파일 이중 포함 방지

```
/**  
 *stdio.h - definitions/declarations for standard I/O routines  
 ****/  
#ifndef _INC_STDIO  
#define _INC_STDIO  
....  
....  
#endif
```



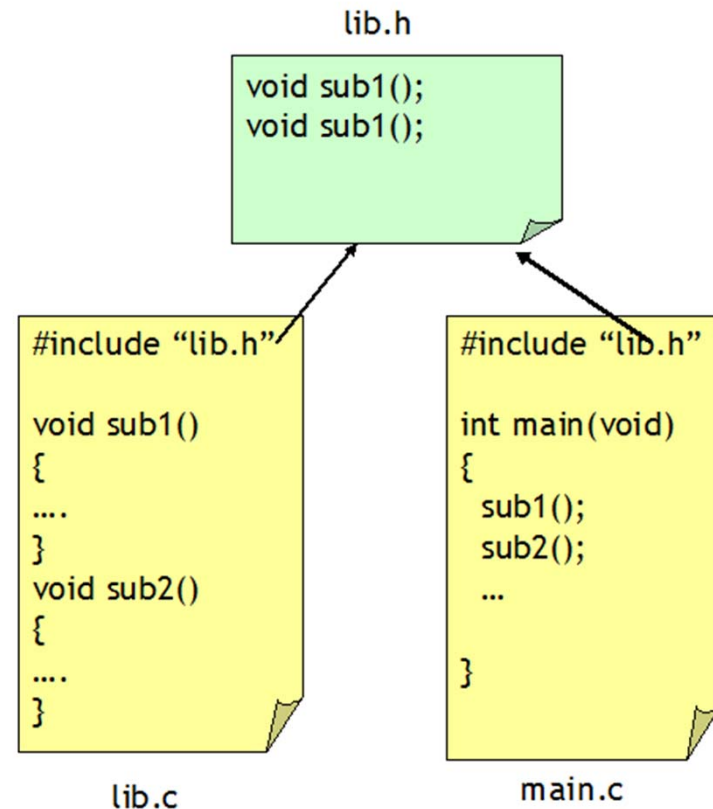
헤더 파일이  
포함되면  
매크로가  
정의되어서  
이중 포함을  
방지합니다.





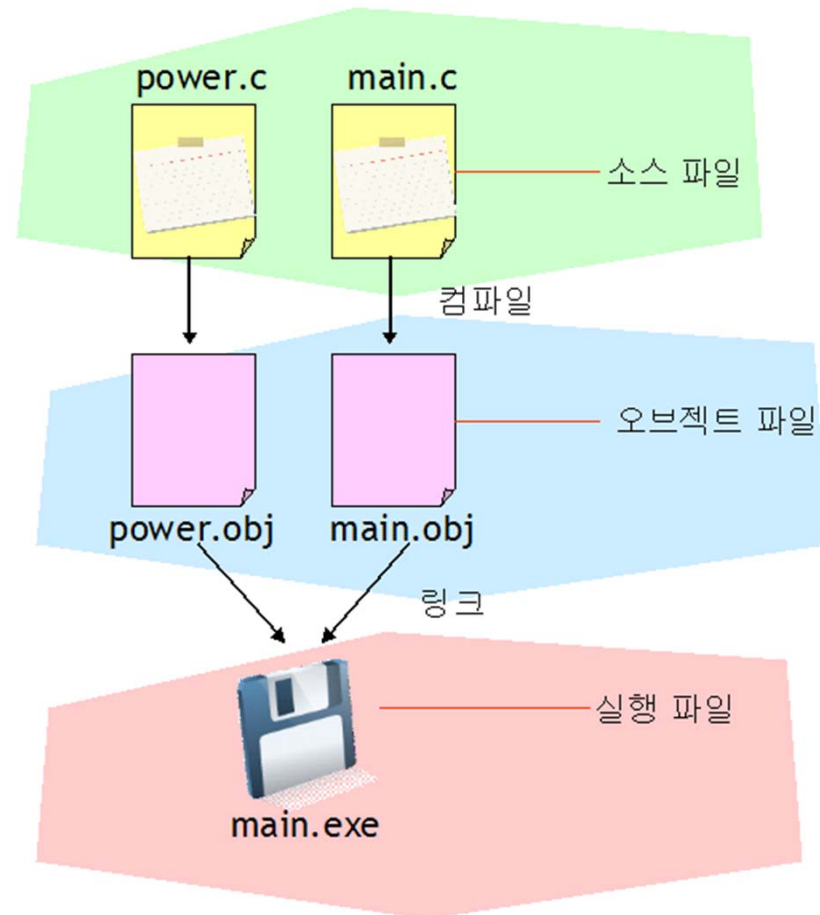
# 다중 소스 파일

- 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

*multiple\_source.c*

```
// 다중 소스 파일
#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;
}
```

*power.h*

```
// power.c에 대한 헤더 파일
#ifndef POWER_H
#define POWER_H

double power(int x, int y);
#endif
```

*power.c*

```
// 다중 소스 파일
#include "power.h"
double power(int x, int y)
{
    double result = 1.0;
    int i;

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

    return result;
}
```



# 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

graphics.h

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

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

graphics.c

공급자

헤더 파일 포함

```
#include "graphics.h"
```

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

main.c

사용자

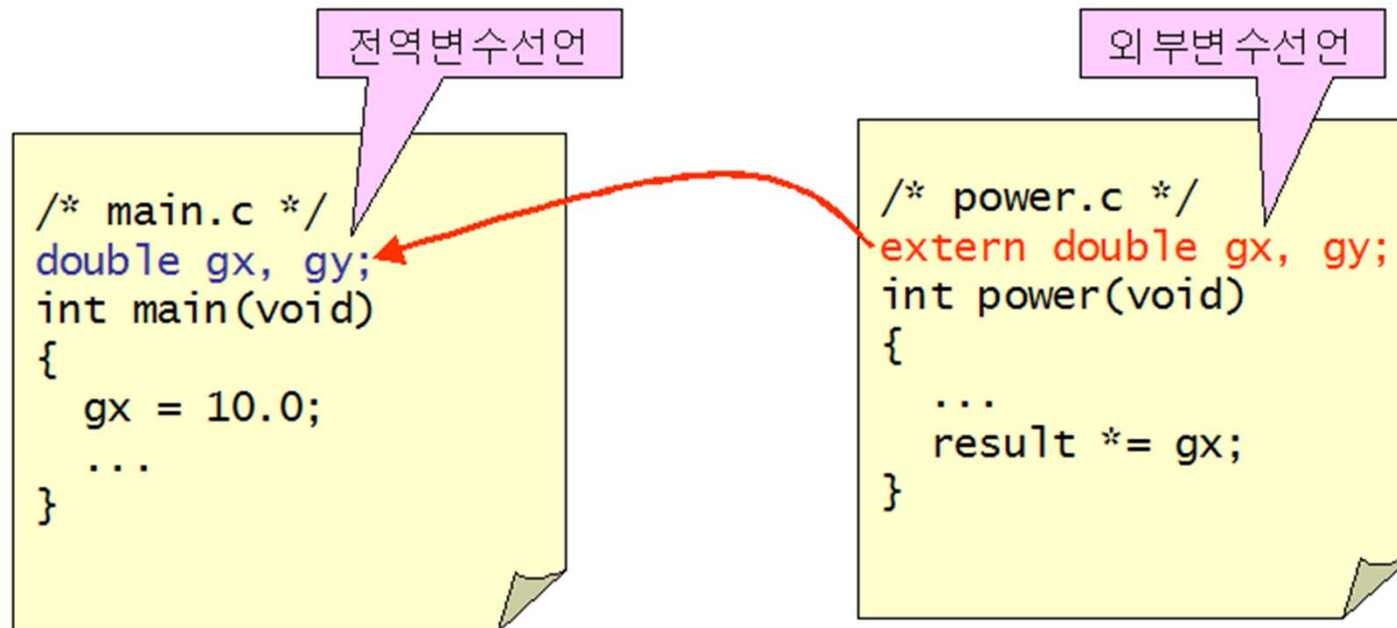
```
#include "graphics.h"
```

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

draw.c

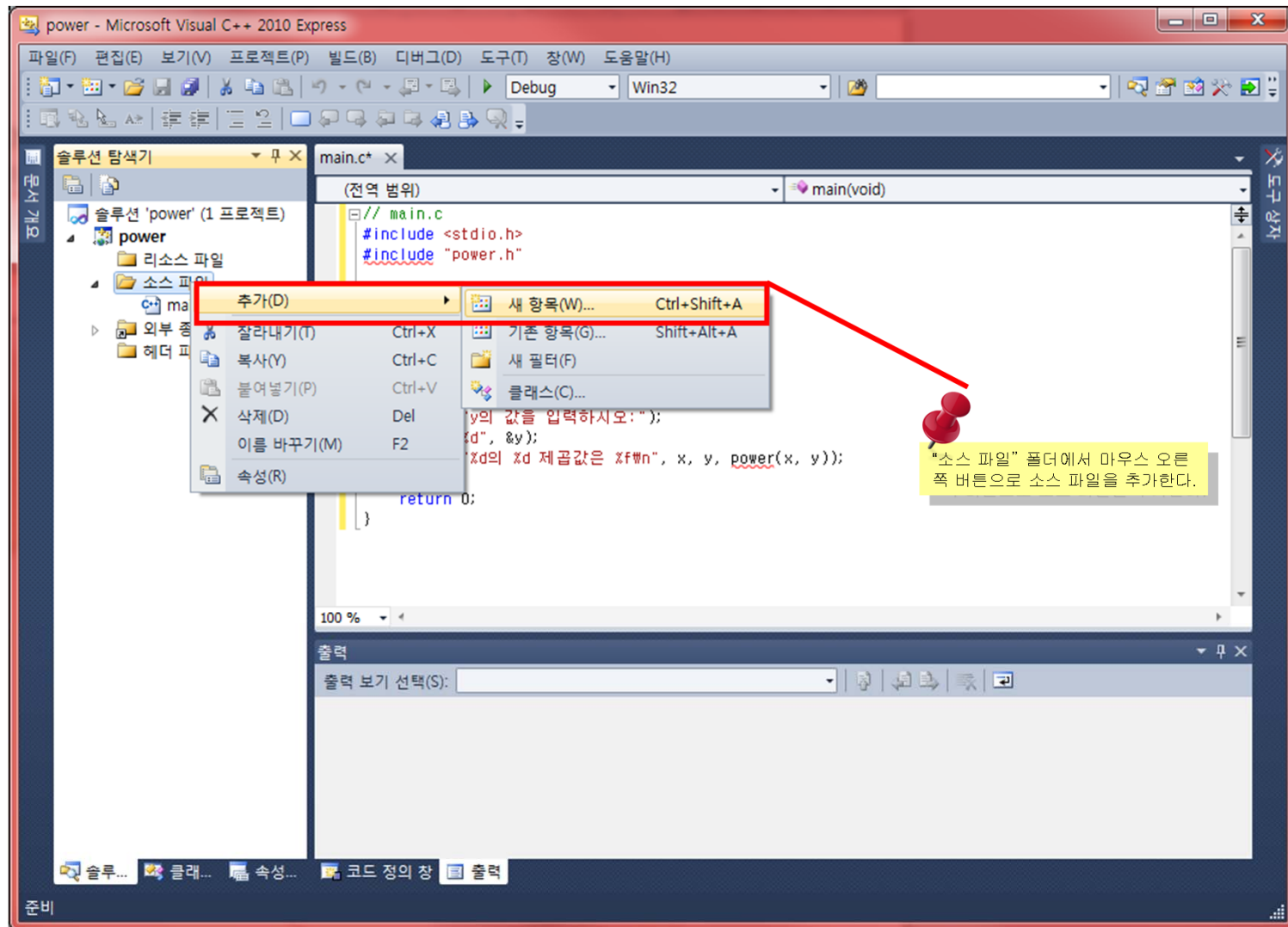


# 다중 소스 파일에서 외부 변수



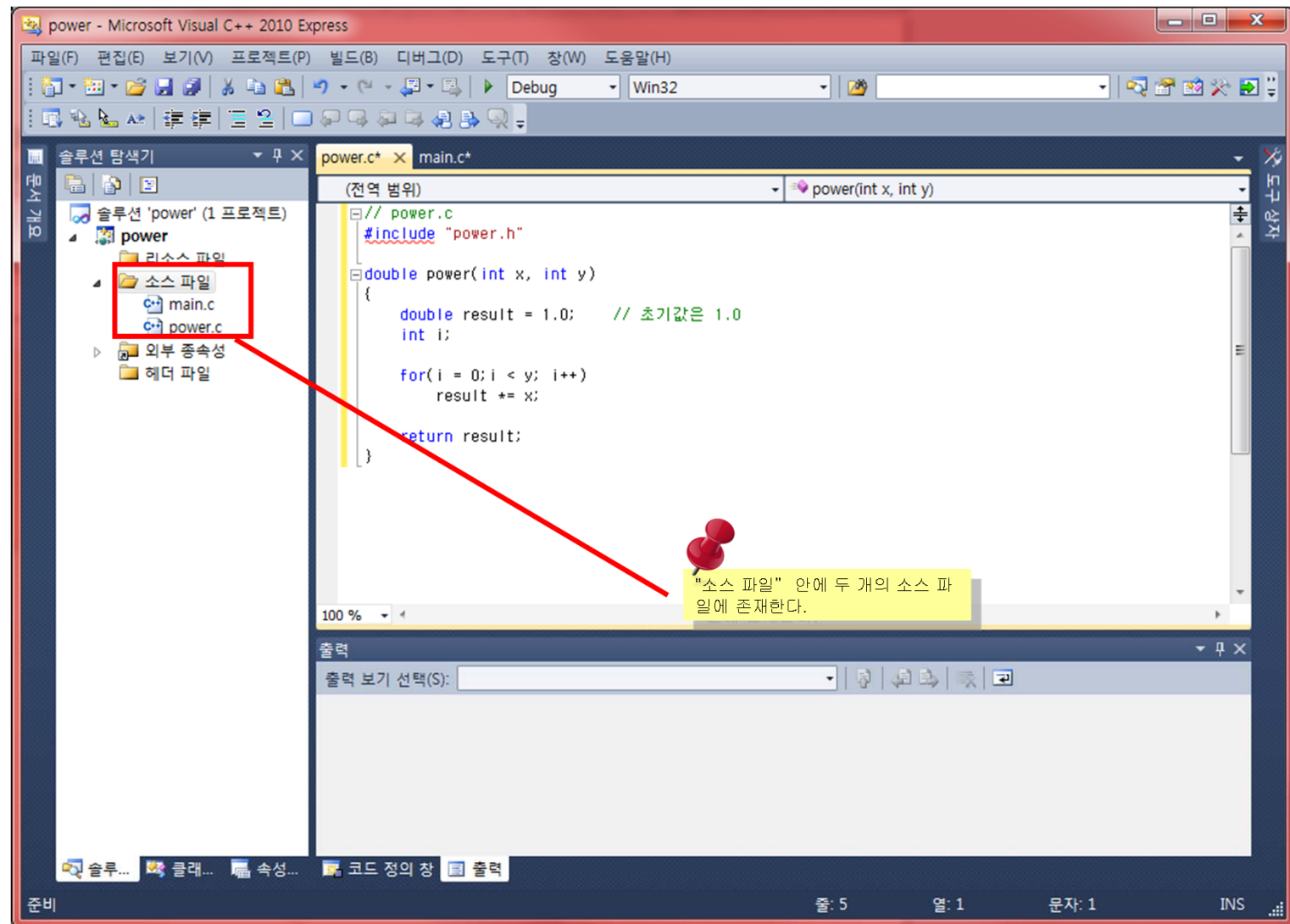


# 비주얼 C++에서 다중 소스 파일





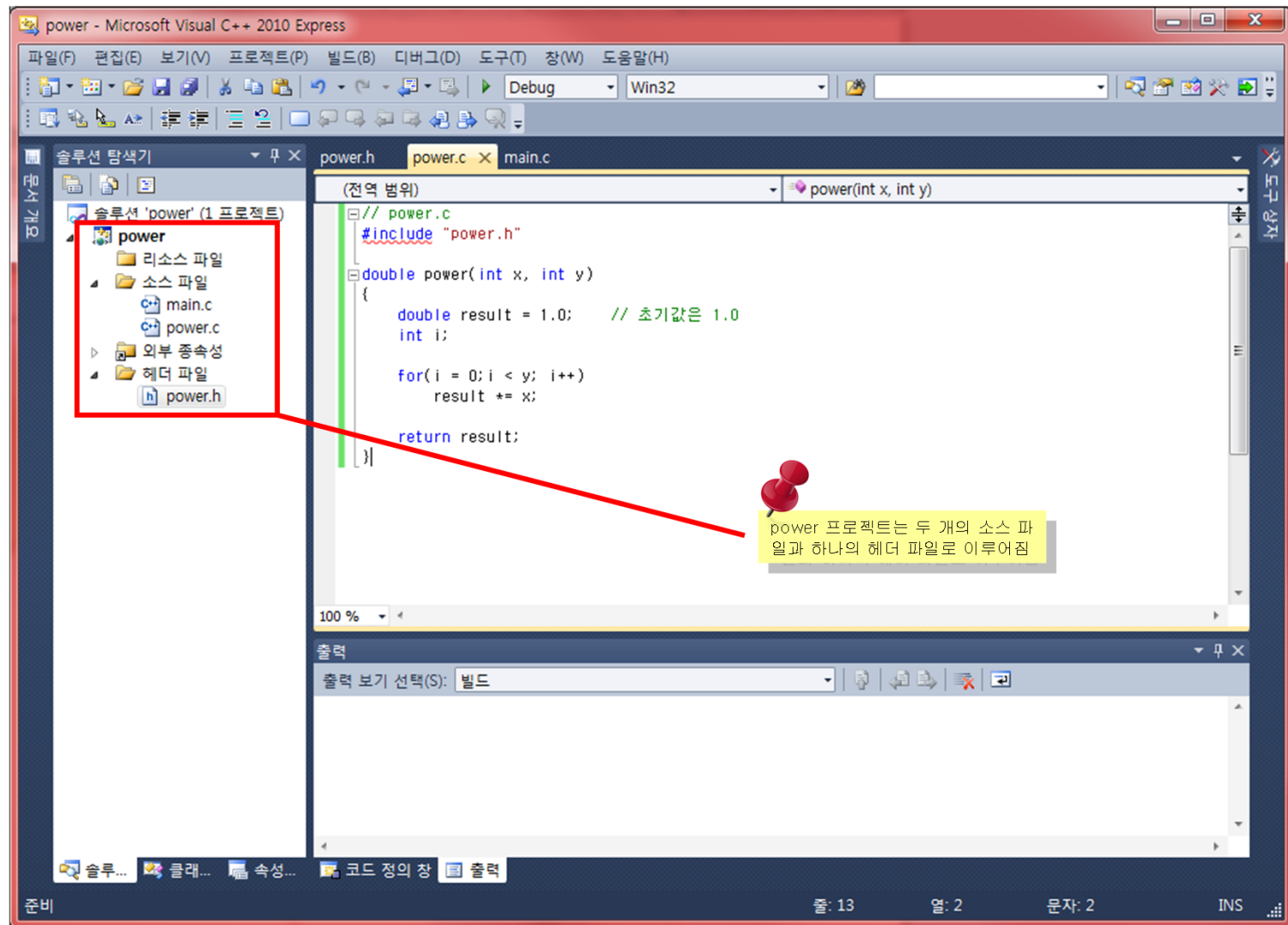
# 비주얼 C++에서 다중 소스 파일







# 비주얼 C++에서 다중 소스 파일





# 헤더 파일 이중 포함 방지

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

```
#include "rect.h"
```

```
#include "rect.h"
```

구조체의 정의가 이중으로 포함되어서 오류가 발생한다.

```
#define DEBUG
```

```
void draw_rect(const RECT *r)
```

```
{
```

```
#ifdef DEBUG
```

```
    printf("draw_area(x=%d, y=%d, w=%d, h=%d) \n", r->x, r->y, r->w, r->h);
```

```
#endif
```

```
}
```



# 헤더 파일 이중 포함 방지

```
#ifndef RECT_H  
#define RECT_H
```

RECT\_H가 정의되어 있지 않은  
경우에만 포함시킨다.

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

RECT\_H 매크로를 정의한다.

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

```
#endif
```



## 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.





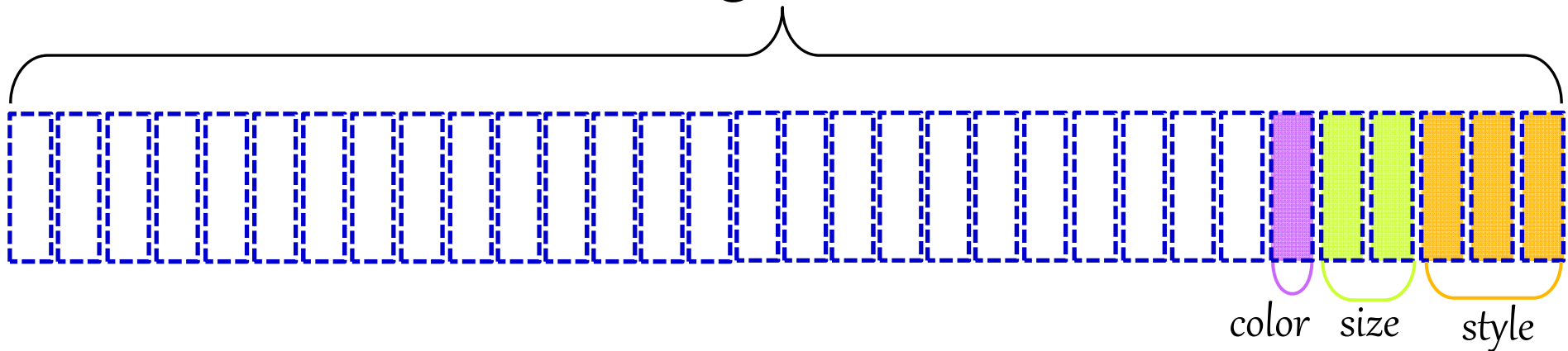
# 비트 필드 구조체

- 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;  
};
```

unsigned int





# 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;
```

```
}
```



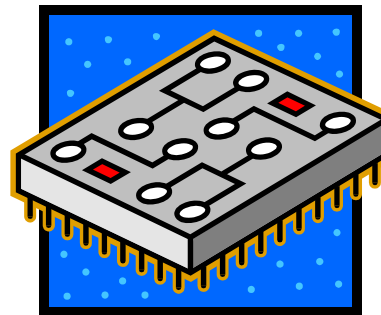
```
style=5 size=3 color=1  
sizeof(p1)=4  
p1=ccccccfd
```



# 비트 필드 사용시에 주의점

```
struct product {  
    long code; // ① 일반 멤버도 가능하다.  
    unsigned style : 3;  
    unsigned : 5; // ② 자리만 차지한다.  
    unsigned size : 2;  
    unsigned color : 1;  
    unsigned : 0; // ③ 현재 워드의 남아있는 비트를 버린다.  
    unsigned state : 3; // 여기서부터는 다음 워드에서 할당된다.  
};
```

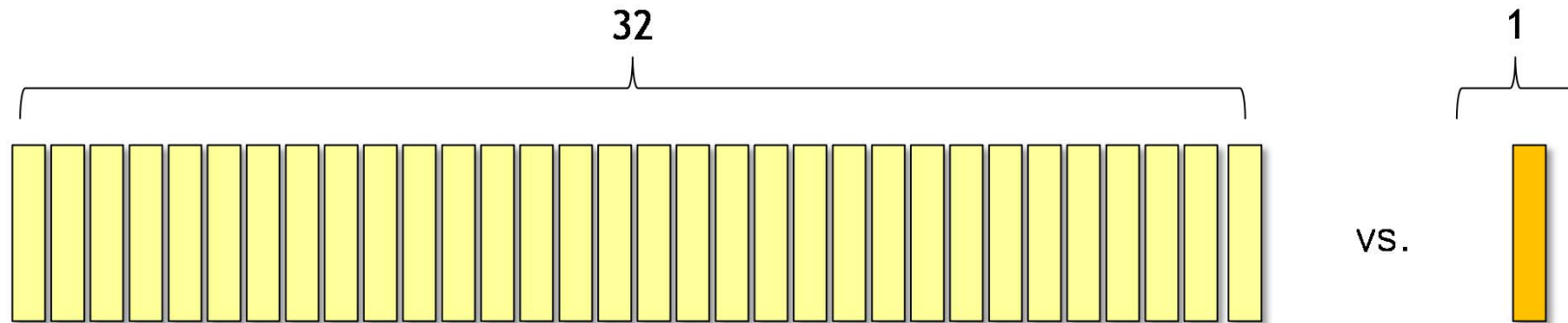
•비트 필드의 응용 분야: 하드웨어 포트 제어





# 비트 필드의 장점

- 메모리가 절약된다.
  - **ON** 또는 **OFF**의 상태만 가지는 변수를 저장할 때 32비트의 **int**형 변수를 사용하는 것보다는 1비트 크기의 비트 필드를 사용하는 편이 훨씬 메모리를 절약한다.

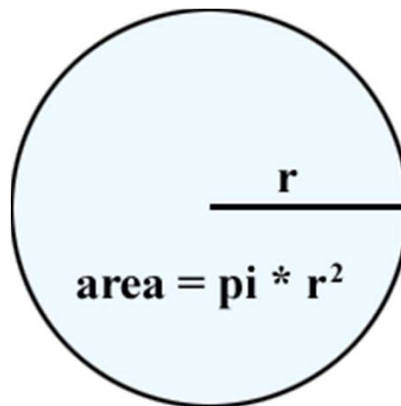






# 실습: 전 처리기 사용하기

- 원의 면적을 구하는 프로그램을 미국 버전과 한국 버전으로 작성한다.
- 미국 버전에서는 모든 메시지가 영어로 출력되고 단위도 인치가 된다.
- 한국 버전에서는 모든 메시지가 한글로 출력되고 단위도 **cm**가 된다.
- **SQUARE()** 함수 매크로도 억지로 사용하여 보자.





## 실행 결과



Please enter radius of a circle(inch) : 100  
area(100.000000) is called  
area of the circle is 31415.920000

원의 반지름을 입력하시오(cm): 100  
area(100.000000)가 호출되었음  
원의 면적은 31415.920000입니다.



# 실습 예제

```
#include <stdio.h>
#define USA
#define DEBUG
#ifdef PI
#define PI 3.141592
#endif
#ifdef SQUARE
#define SQUARE(r)    (r)*(r)
#endif
double area(double radius)
{
    double result=0.0;
#ifdef DEBUG
#ifdef USA
        printf("area(%f) is called \n", radius);
    #else
        printf("area(%f)가 호출되었음radius);
    #endif
#endif
    result = PI*SQUARE(radius);
    return result;
}
```



# 실습 예제

```
int main(void){
    double radius;
#ifdef USA
    printf("Please enter radius of a circle(inch) : ");
#else
    printf("원의 반지름을 입력하십시오");
#endif
    scanf("%lf", &radius);
#ifdef USA
    printf("area of the circle is %f \n", area(radius));
#else
    printf("원의 면적은 %f입니다\n", area(radius));
#endif
    return 0;
}
```



# 도전문제

- 버전을 나타내는 매크로를 정의하고 버전이 **100** 이하이면 원의 면적을 계산할 수 없다는 메시지를 출력하고 종료하게끔, 위의 프로그램을 수정하여 보자.
- `__DATE__`와 `__LINE__`을 출력하여 보자.





## 중간 점검

1. 구조체의 일종으로 멤버들의 크기가 비트 단위로 나누어져 있는 구조체는 \_\_\_\_\_이다.
2. 비트 필드 구조체를 정의하는 경우, 자료형은 \_\_\_\_\_이나 \_\_\_\_\_을 사용하여야 한다.





# Q & A

