제15장 전처리 및 비트연산

유 준 범 (JUNBEOM YOO)

jbyoo@konkuk.ac.kr
http://dslab.konkuk.ac.kr

Ver. 2.0

※ 본 강의자료는 생능출판사의 "PPT 강의자료"를 기반으로 제작되었습니다.

이번 장에서 학습할 내용

- 전처리기란 무엇인가?
- 단수 매크로
- 함수 매크로
- 내장 매크로
- 다중 소스 파일
- 비트 단위 연산자

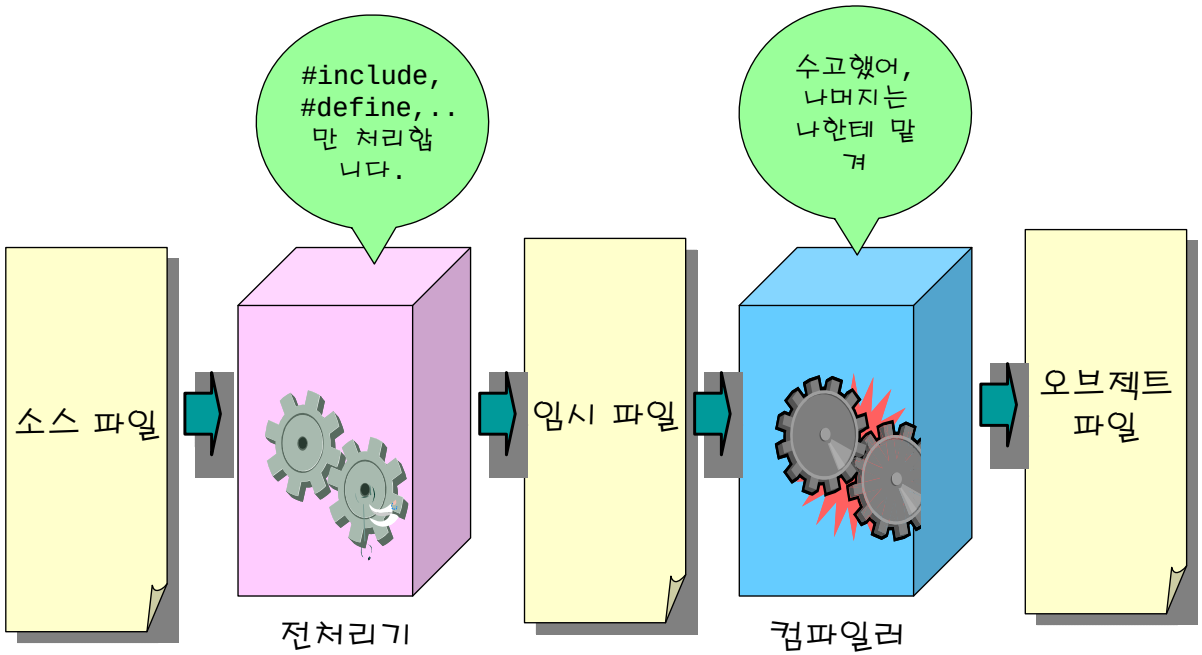
mile2km.c

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

int main(int argc, char *argv[])
{
	double mile, km;

	if( argc != 2 ){
		printf("사용 방법: mile2km 거리\n");
		return 1;
	}
	mile = atof(argv[1]);
	km = 1.609 * mile;
	printf("입력된 거리는 %f km입니다. \n", km);

	return 0;
}
```

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
c:\cprogram\mainarg\Debug>mainarg 10
입력된 거리는 16.090000 km입니다.
c:\cprogram\mainarg\Debug>
```

전처리기란?

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



전처리기의 요약

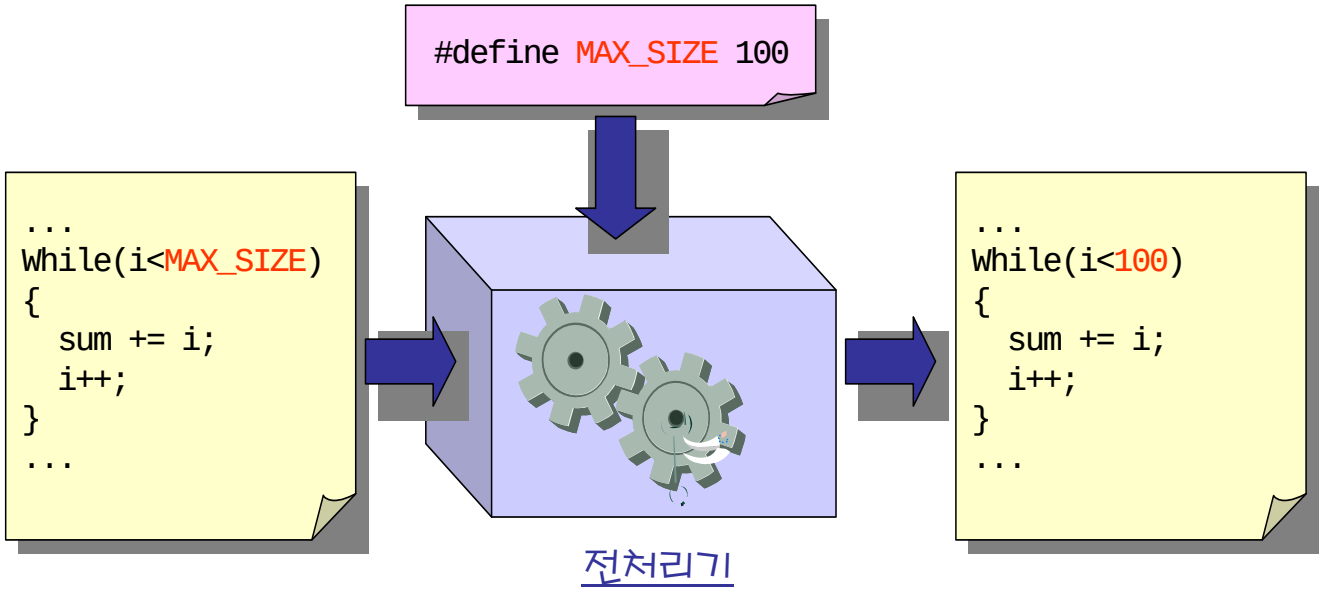
지시어	의미
#define	매크로 정의
#include	파일 포함
#undef	매크로 정의 해제
#if	조건이 참일 경우
#else	조건이 거짓일 경우
#endif	조건 처리 문장 종료
#ifdef	매크로가 정의되어 있는 경우
#ifndef	매크로가 정의되어 있지 않은 경우
#line	행번호 출력
#pragma	시스템에 따라 의미가 다름

단순 매크로

- 단순 매크로(macro): 숫자 상수를 기호 상수로 만든 것

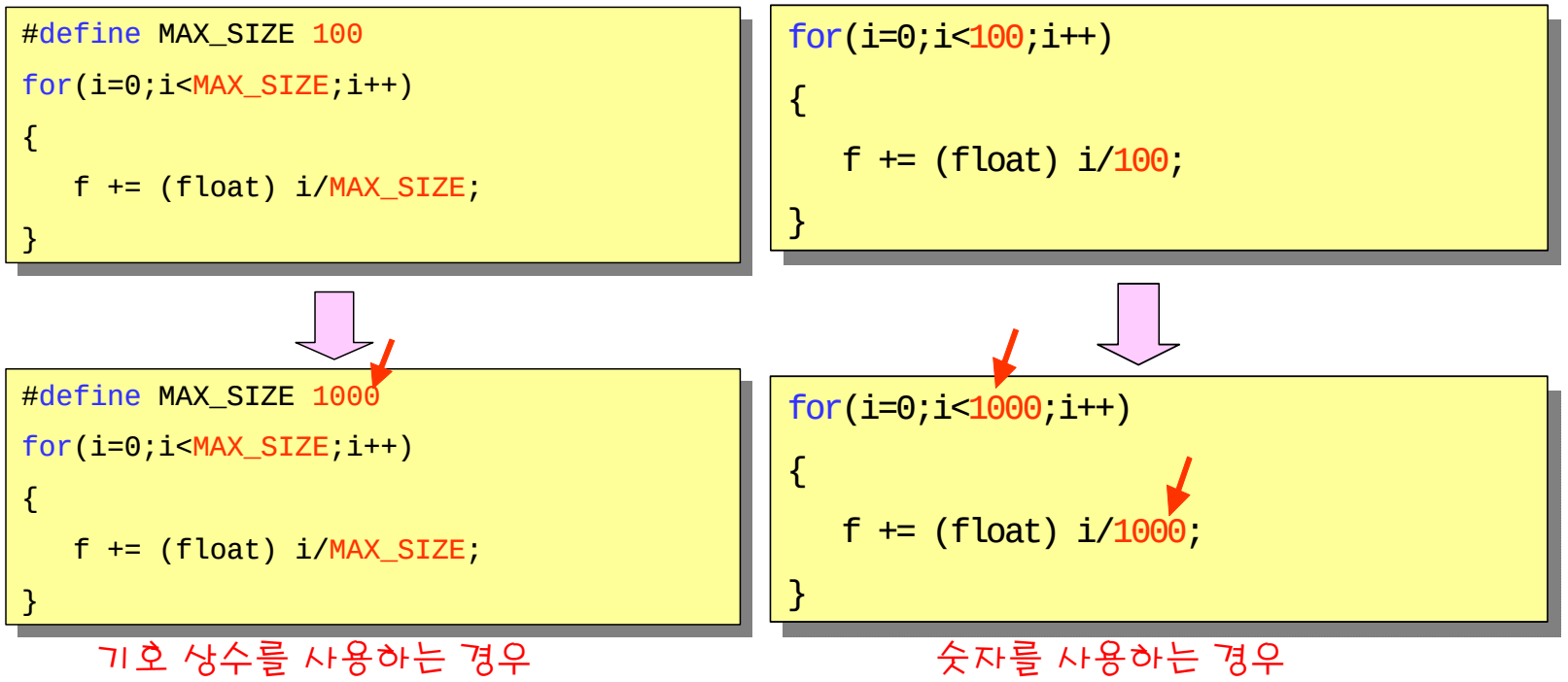
```
#define      매크로      치환텍스트
```

- (예) #define MAX_SIZE 100



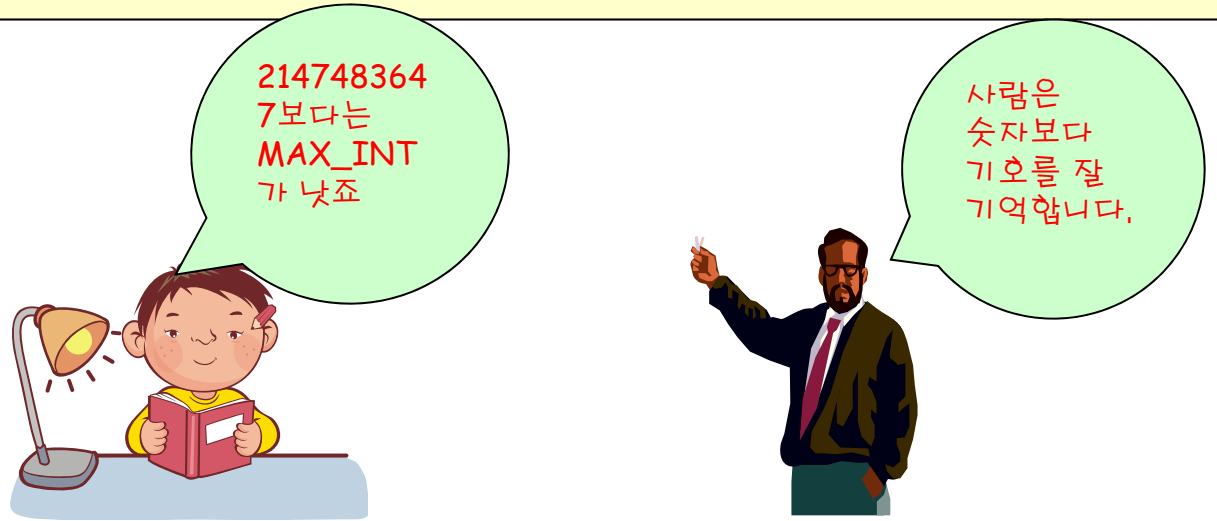
단순 매크로의 장점

- 프로그램의 가독성을 높인다.
- 상수의 변경이 용이하다.



단순 매크로의 예

```
#define     PRINT               printf
#define     PI                  3.141592            // 원주율
#define     TWOPI               (3.141592 * 2.0)    // 원주율의 2배
#define     MAX_INT   2147483647            // 최대 정수
#define     EOF                 (-1)                // 파일의 끝 표시
#define     MAX_STUDENTS        2000                // 최대 학생 수
#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 #1

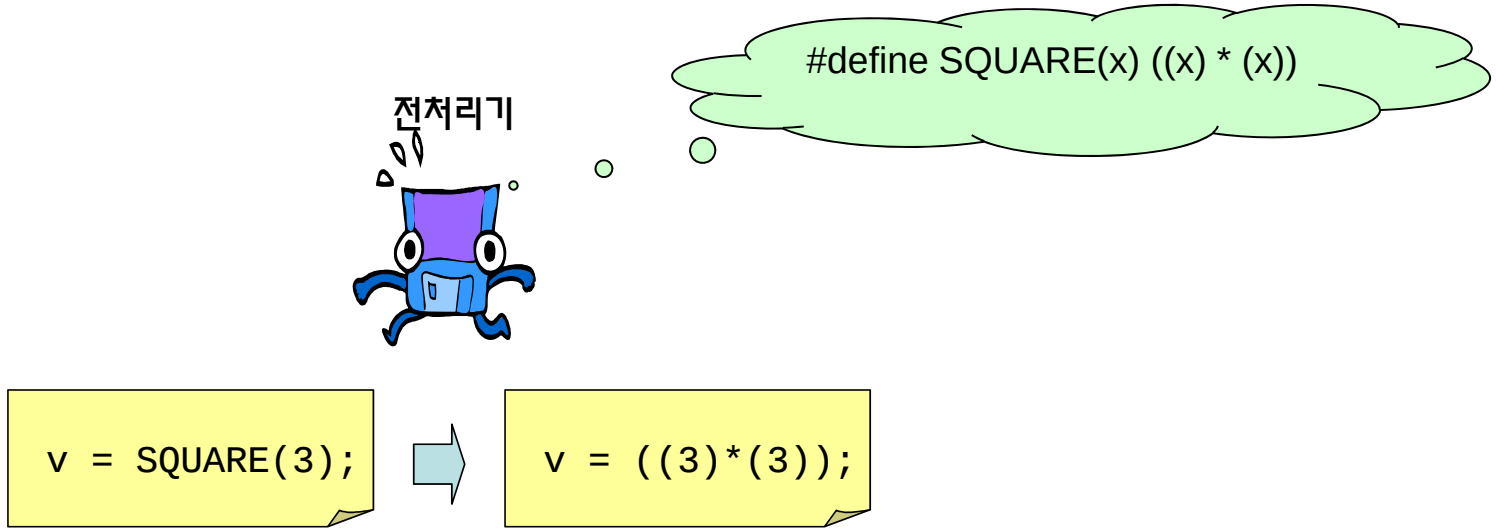
```
#include <stdio.h>
#define AND                     &&
#define OR                      ||
#define NOT                     !
#define IS                      ==
#define ISNOT                   !=

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;
}
```

함수 매크로

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

#define 매크로(매개변수1, 매개변수2,...) 치환텍스트

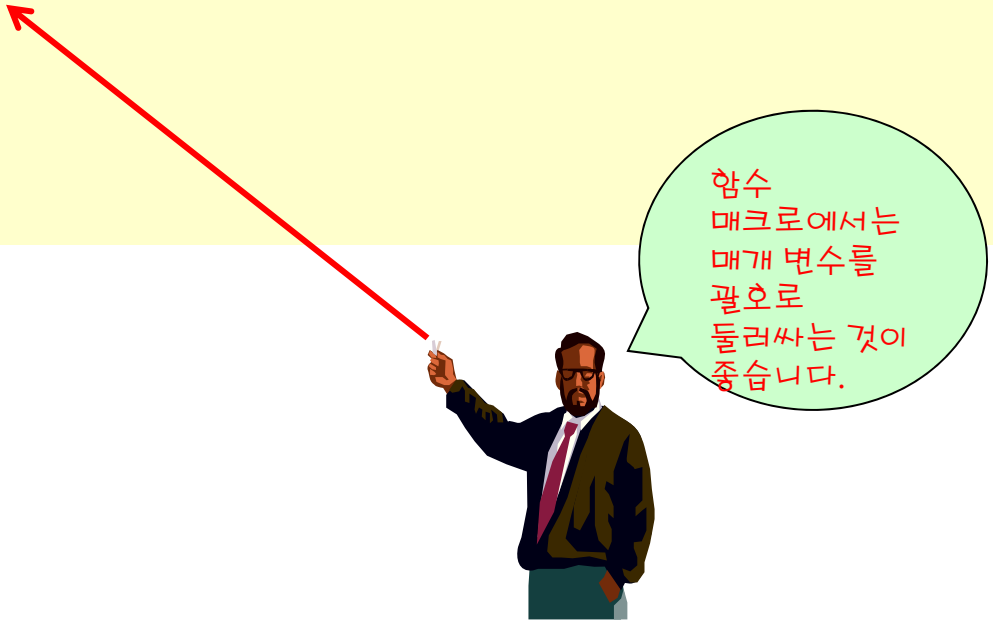
- (예) #define SQUARE(x) ((x) * (x))



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)
```

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

v = SQUARE(a+b);

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



예제 #1

```
1.  // 매크로 예제
2.  #include <stdio.h>
3.  #define SQUARE(x)  ((x) * (x))
4.
5.  int main(void)
6.  {
7.          int x = 2;
8.
9.          printf("%d\n", SQUARE(x));
10.         printf("%d\n", SQUARE(3));
11.         printf("%f\n", SQUARE(1.2));          // 실수에도 적용 가능
12.         printf("%d\n", SQUARE(x+3));
13.         printf("%d\n", 100/SQUARE(x));
14.         printf("%d\n", SQUARE(++x));          // 논리 오류
15.
16.         return 0;
17. }
```

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

```
4
9
1.440000
25
25
16
```

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))

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.

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

예제: ASSERT 매크로

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

2.  #define DEBUG

3.  #ifdef DEBUG
4.  #define ASSERT(exp)             { if (!(exp)) \
5.          { printf("가정(" #exp ")이 소스 파일 %s %d번째 줄에서 실패.\n"\
6.          ,__FILE__, __LINE__), exit(1);}}
7.  #else
8.  #define ASSERT(exp)
9.  #endif

10. int main(void)
11. {
12.         int sum;                    // 지역 변수의 초기값은 0이 아님

13.         ASSERT(sum == 0);           // sum의 값은 0이 되어야 함.
14.         return 0;
15. }
```

매크로를 다음 줄로 연장할 때 사용

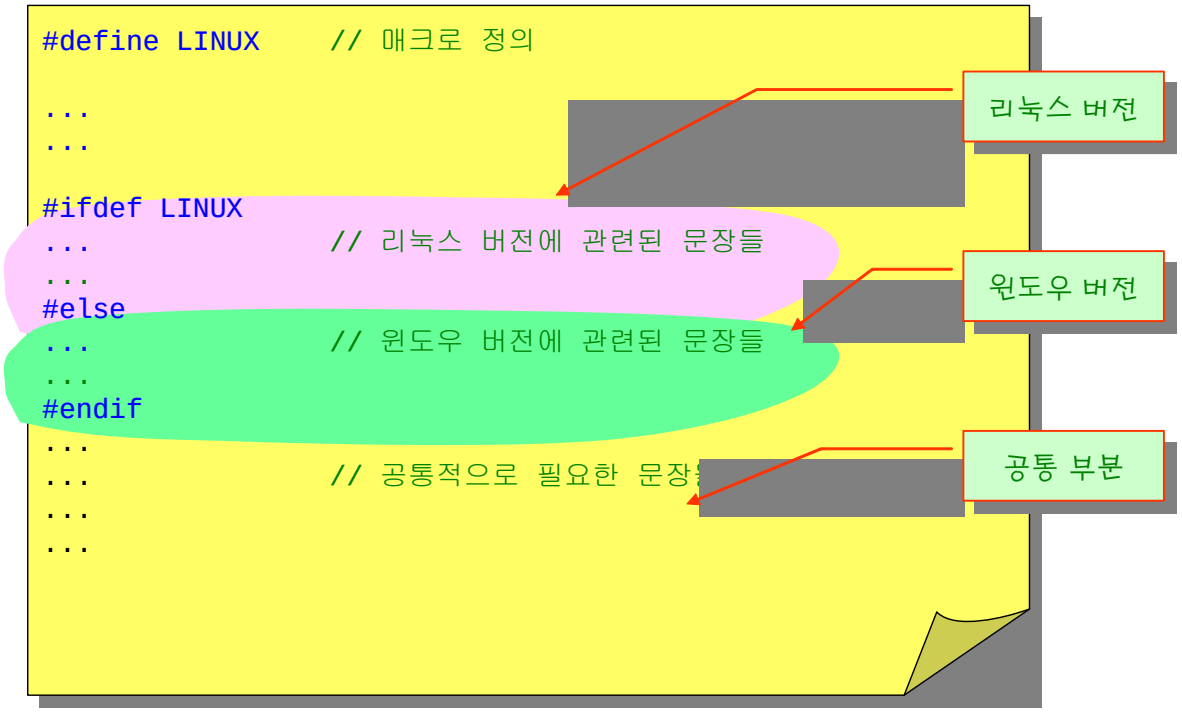
```
가정(sum == 0)이 소스 파일 c:\cprogram\test\test.c 17번째 줄에서 실패.
```

#ifdef

- 조건부 컴파일을 지시
- 어떤 조건이 만족되었을 경우에만 컴파일

```
#ifdef 매크로
문장1       // 매크로가 정의되었을 경우
#else
문장2       // 매크로가 정의되지 않았을 경우
#endif
```

- (예)



#if

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

```
#if 조건
문장들
#endif
```

```
#define METHOD 1
...
#if METHOD == 1
printf("방법 1이 선택되었습니다.\n");
#endif
```

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

else if를 줄인 것

다양한 예

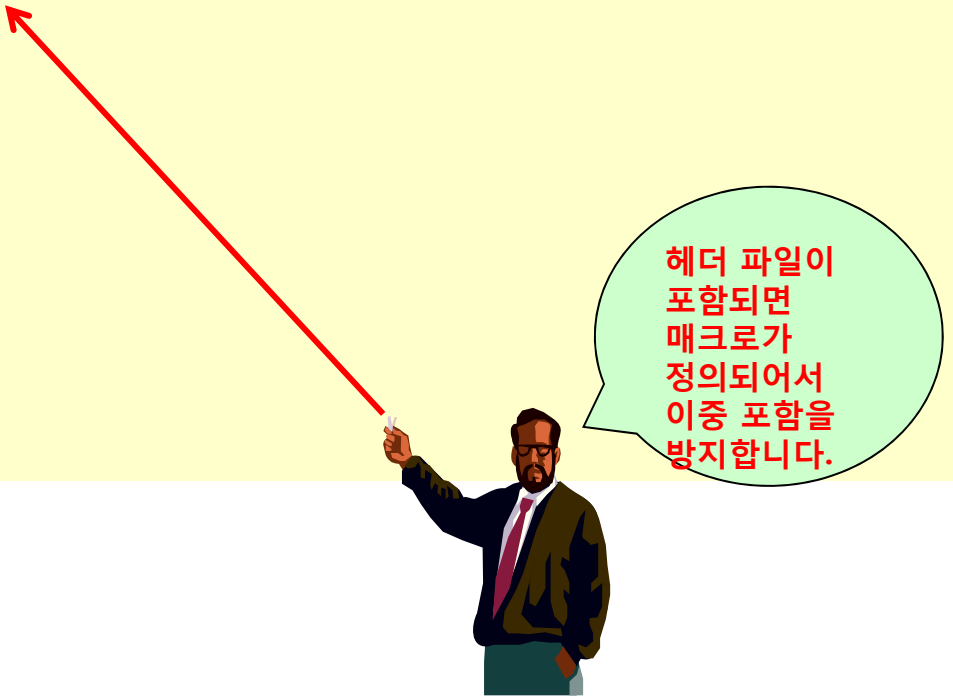
```
#if (VERSION > 3)                    // 버전이 3 이상이면 컴파일
...
#endif

#if (VERSION > 3.0)                  // 오류 !! 버전 번호는 300과 같은 정수로 표시

#if (AUTHOR == "CHULSOO")            // 오류 !!

#if (AUTHOR == KIM)                  // 가능!! KIM은 다른 매크로

#if (VERSION*10 > 500 && LEVEL == BASIC)      // 가능!!

#if (VERSION > 300 || defined(DELUXE) )

#if 0                                // 소스의 일부분을 주석 처리하는 방법
...
#endif
```

조건부 컴파일을 이용하는 디버깅

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

#define DEBUG
...
#ifdef DEBUG
printf("현재 counter의 값은 %d입니다.₩n", counter);
#endif
...
#if defined(DEBUG)
printf("현재 counter의 값은 %d입니다.₩n", counter);
#endif
```

헤더 파일 이중 포함 방지

```
/***
*stdio.h - definitions/declarations for standard I/O routines
****/

#ifndef _INC_STDIO
#define _INC_STDIO
....
....
#endif
```



#undef, #pragma

- #undef: 매크로의 정의를 취소

```
1.  #include <stdio.h>
2.  #define DEBUG

3.  int main(void)
4.  {
5.  #ifdef DEBUG
6.          printf("DEBUG이 정의되었습니다.\n");
7.  #endif

8.  #undef DEBUG                                    // DEBUG 매크로의 정의를 취소

9.  #ifdef DEBUG
10.         printf("DEBUG이 정의되었습니다.\n");    // 컴파일되지 않는다.
11. #endif
12.         return 0;
13. }
```

#include

- 지정된 파일을 읽어서 그 위치에 삽입
  - #include <.....> 표준 디렉토리에서 파일을 찾는다.
  - #include ".....” 현재 디렉토리에서 파일을 찾는다.

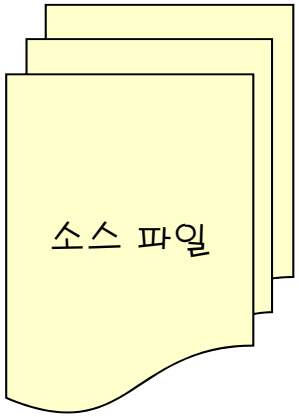
- 표준 디렉토리
  - INCLUDE라는 환경변수가 지정하는 디렉토리이다.
  - 보통은 "C:₩Program Files₩Microsoft Visual Studio₩VC98₩include"

- 하부 디렉토리를 지정할 수 있다.
  - #include "graphic/point.h"

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



예제 #1

multiple_source.c

```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

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

double power(int x, int y);

#endif
```

power.c

```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;
}
```

비트 단위 연산자

- C에서는 비트 단위의 연산도 가능하다.
- 비트 단위 연산자는 정수에만 적용이 가능하다.

연산자	연산자의 의미	설명
&	비트 AND	두개의 피연산자의 해당 비트가 모두 1이면 1, 아니면 0
|	비트 OR	두개의 피연산자의 해당 비트중 하나만 1이면 1, 아니면 0
^	비트 XOR	두개의 피연산자의 해당 비트의 값이 같으면 0, 아니면 1
<<	왼쪽으로 이동	지정된 개수만큼 모든 비트를 왼쪽으로 이동한다.
>>	오른쪽으로 이동	지정된 개수만큼 모든 비트를 오른쪽으로 이동한다.
~	비트 NOT	0은 1로 만들고 1은 0로 만든다.

비트 논리곱 연산자

비트1	비트2	비트1 & 비트2
0	0	0
0	1	0
1	0	0
1	1	1

변수1 00000000 00000000 00000000 00001101 (13)
변수2 00000000 00000000 00000000 00001110 (15)

(변수1 AND 변수2) 00000000 00000000 00000000 00001100 (12)

변수1 00000000 00000000 00000000 00001101 (13)
마스크 00000000 00000000 00000000 00000111 (7)

(마스크 연산후) 00000000 00000000 00000000 00000101 (5)

bit_AND.c

```
1.  // 비트 단위 AND
2.  #include <stdio.h>

3.  int main(void)
4.  {
5.          int x = 13;             // 00000000 00000000 00000000 00001101
6.          int y = 15;             // 00000000 00000000 00000000 00001110
7.          int z = x & y;          // 00000000 00000000 00000000 00001100

8.          printf("%08X \n", z);

9.          return 0;
10. }
```

```
0000000D
```

비트 논리합 연산자

비트1	비트2	비트1 | 비트2
0	0	0
0	1	1
1	0	1
1	1	1

```
                변수1 00000000 00000000 00000000 00001001 (9)
                변수2 00000000 00000000 00000000 00001010 (10)
-------------------------------------------------------------
(변수1 OR 변수2)   00000000 00000000 00000000 00001011 (11)
```

```
                변수1 00000000 00000000 00000000 00001001 (9)
              비트설정 00000000 00000000 00000000 00000111 (7)
-------------------------------------------------------------
  (비트 OR 연산후) 00000000 00000000 00000000 00001111 (15)
```

bit_OR.c

```
1.  // 비트 단위 OR
2.  #include <stdio.h>

3.  int main(void)
4.  {
5.          int x = 9;      // 00000000 00000000 00000000 00001001
6.          int y = 10;     // 00000000 00000000 00000000 00001010
7.          int z = x | y;  // 00000000 00000000 00000000 00001011

8.          printf("%08X \n", z);

9.          return 0;
10. }
```

```
0000000B
```

비트 배타 논리합 연산자

비트1	비트2	비트1 ^ 비트2
0	0	0
0	1	1
1	0	1
1	1	1

변수1 00000000 00000000 00000000 00001001 (9)
변수2 00000000 00000000 00000000 00001010 (10)

(변수1 XOR 변수2) 00000000 00000000 00000000 00000011 (3)

비트 부정 연산자

비트1	~비트1
0	1
1	0

부호비트가 반전되었기 때문에 음수가 된다.

변수1 00000000 00000000 00000000 00001001 (9)

(NOT 변수1) 11111111 11111111 11111111 11110110 (-10)

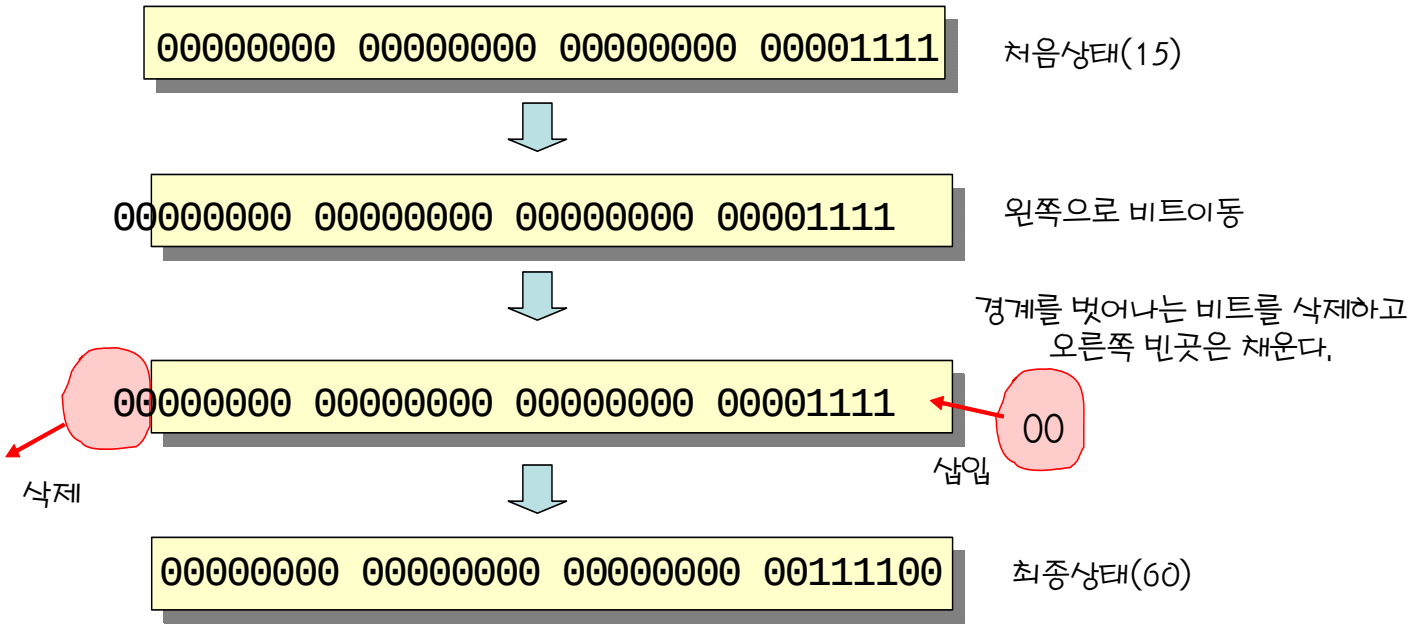
bit_NOT.c

```
1.  // 비트 단위 NOT
2.  #include <stdio.h>

3.  int main(void)
4.  {
5.          int x = 9;      // 00000000 00000000 00000000 00001001
6.          int z = ~x;     // 11111111 11111111 11111111 11110110

7.          printf("%08X (%d)\n", z, z);

8.          return 0;
9.  }
```

```
FFFFFFF6 (-10)
```

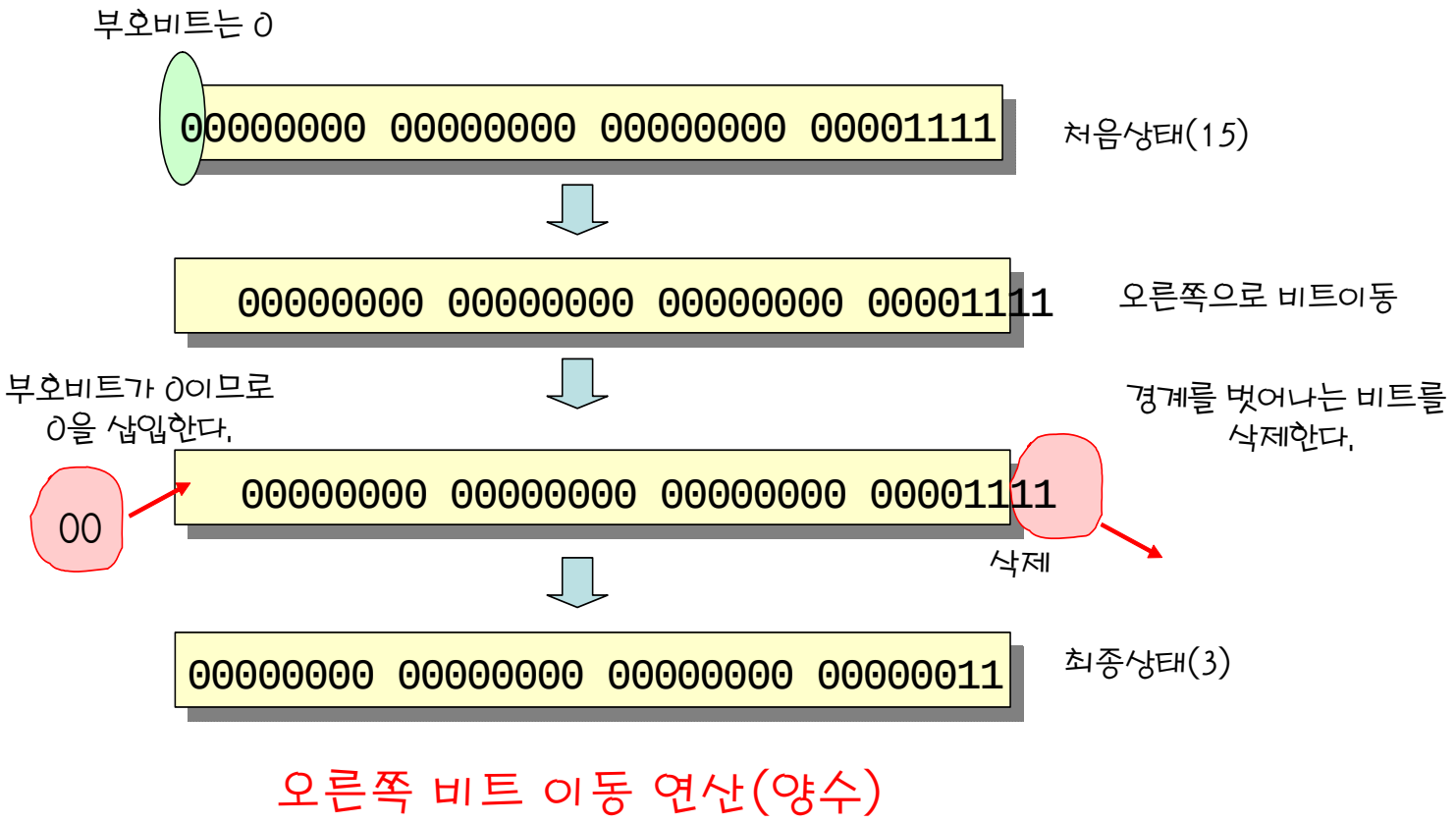
비트 이동 연산자

연산자	기호	설명
왼쪽 비트 이동	<<	x << y x의 비트들을 y 칸만큼 왼쪽으로 이동
오른쪽 비트 이동	>>	x >> y x의 비트들을 y 칸만큼 오른쪽으로 이동



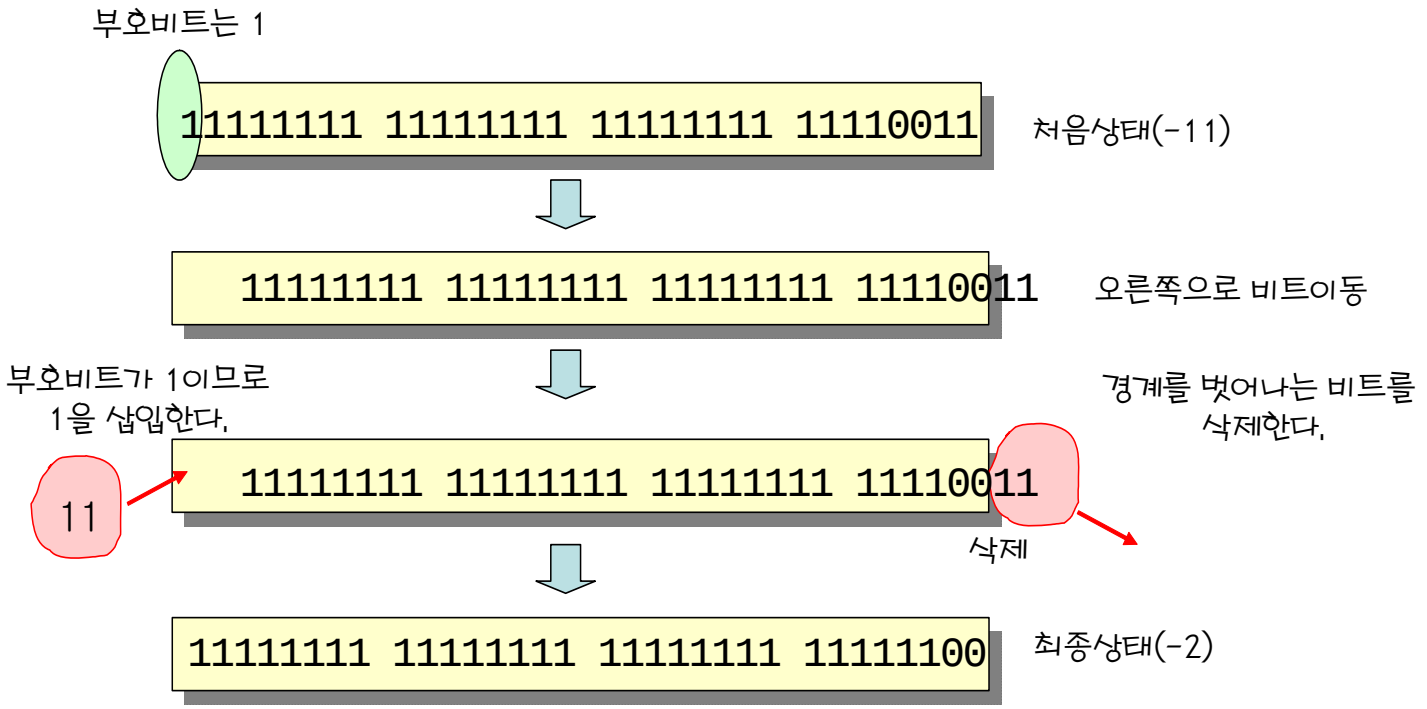
왼쪽 비트 이동 연산

오른쪽 비트 이동 연산(양수)



오른쪽 비트 이동 연산(양수)

오른쪽 비트 이동 연산(음수)



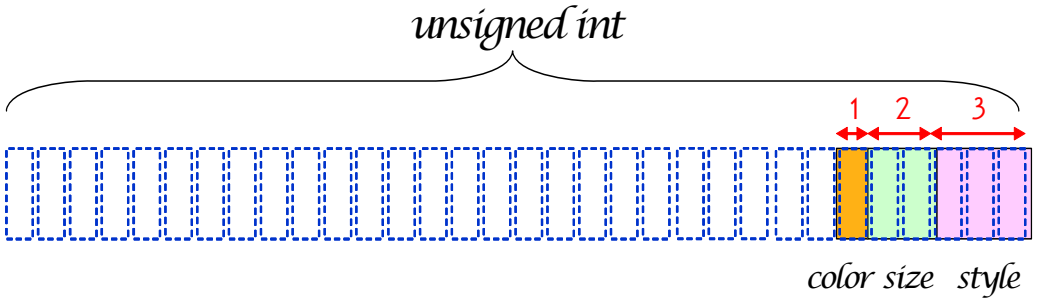
오른쪽 비트 이동 연산(음수)

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

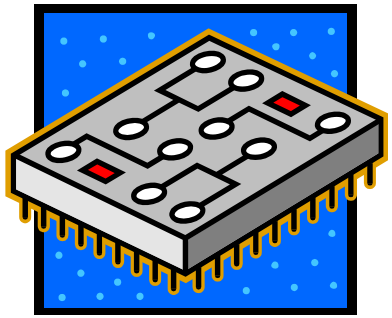
```c
// 비트 필드 구조체
#include <stdio.h>

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



Q & A