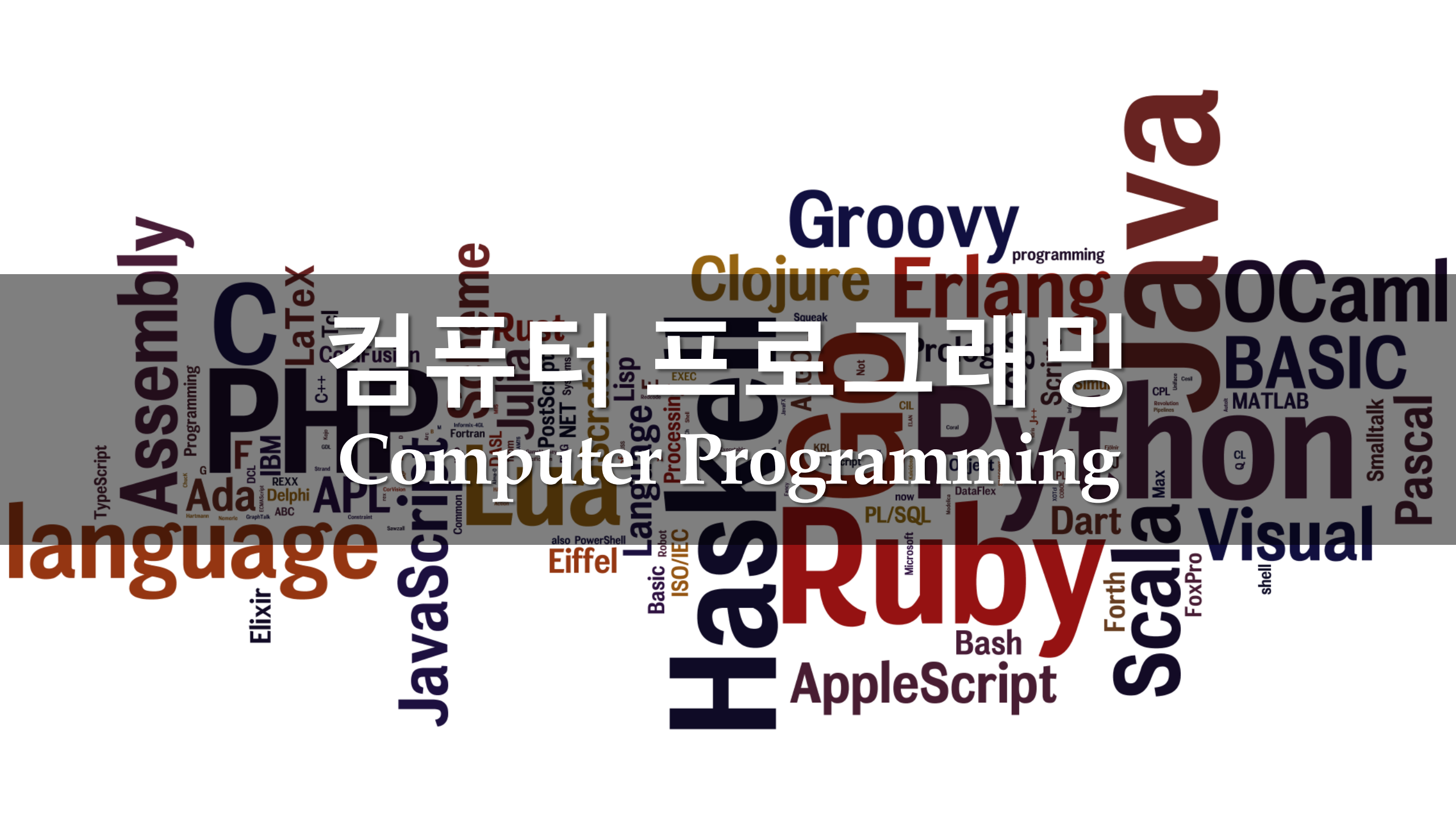
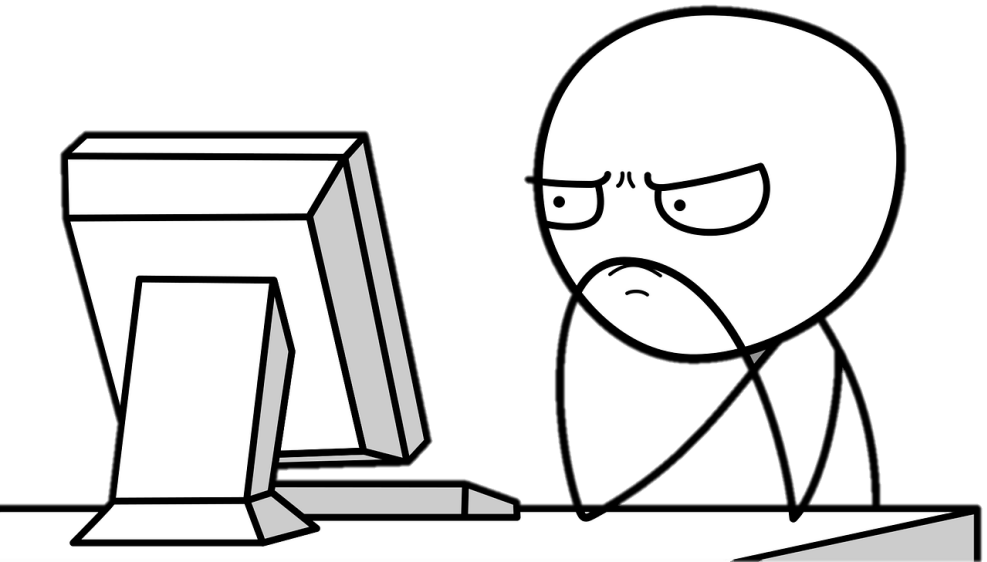

컴퓨터 프로그래밍
Computer Programming


05
연산자

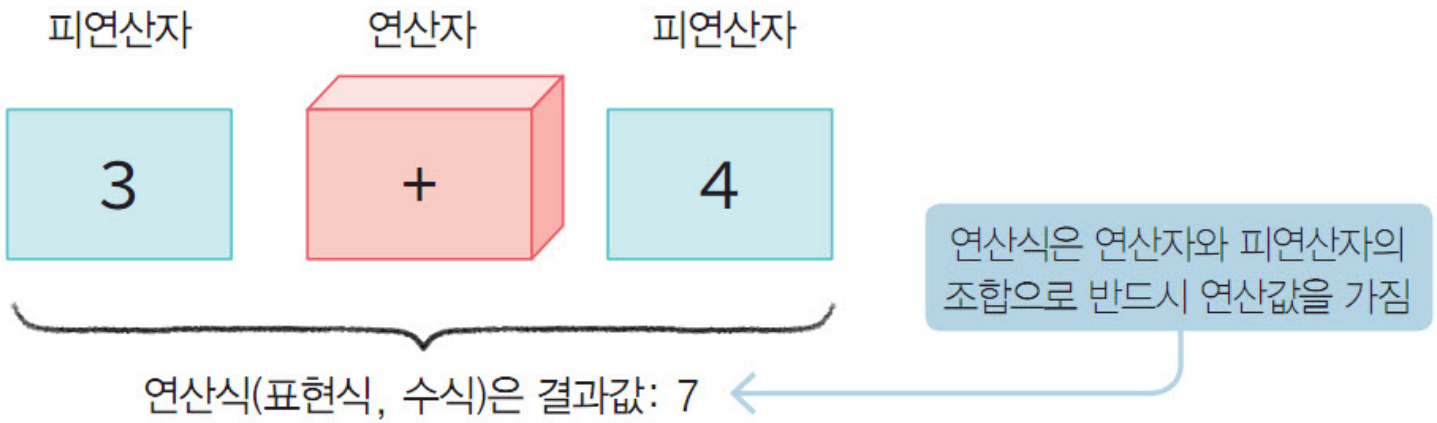
목차

1. 연산식과 다양한 연산자



연산자와 피연산자, 연산식과 연산값

- 연산식
  - 일상 생활에서 사용하는 (3 + 4 * 5)와 같은 간단한 식을 수식
  - 변수와 다양한 리터럴 상수, 함수의 호출 등으로 구성되는 식을 연산식
  - 반드시 하나의 결과값인 연산값
- 연산자operator와 피연산자operand
  - 산술연산자 +, -, * 기호와 같이 이미 정의된 연산을 수행하는 문자 또는 문자조합 기호
  - 연산operation에 참여하는 변수나 상수
- 연산식 3 + 4
  - '+'는 연산자이고, 3과 4는 피연산자
  - 항상 하나의 결과값: 7
    - 연산식의 결과값을 간단히 '연산값



다양한 연산자

- 단(일)항unary, 이항binary, 삼항ternary 연산자
  - 연산자는 연산에 참여하는 피연산자operand의 갯수에 따라 구분
    - 부호를 표시하는 +, -는 단항연산자
    - 덧셈, 뺄셈의 +, -, *, / 등의 연산은 이항연산자
    - 삼항연산자는 조건연산자 '?:'가 유일
- 단항연산자
  - 연산자의 위치에 따라 전위와 후위로 나뉨
    - ++a처럼 연산자가 앞에 있으면 전위prefix 연산자
    - a++와 같이 연산자가 뒤에 있으면 후위postfix 연산자

다양한 연산자

단항연산자

연산자 피연산자 (전위: prefix)

++a: 전위 증가연산자

sizeof (int): sizeof 연산자

(int) 3.46: 자료형 변환연산자

-3: 부호연산자

피연산자 연산자 (후위: postfix)

b--: 후위 감소연산자

a++: 후위 증가연산자

이항연산자

피연산자 1 연산자 피연산자 2

a % 3

3 && 4

5 >= 7

삼항연산자

피연산자 1 ? 피연산자 2 : 피연산자 3

3 ? 4 : 5

(5 >= 7) ? (3+5) : (10/2)

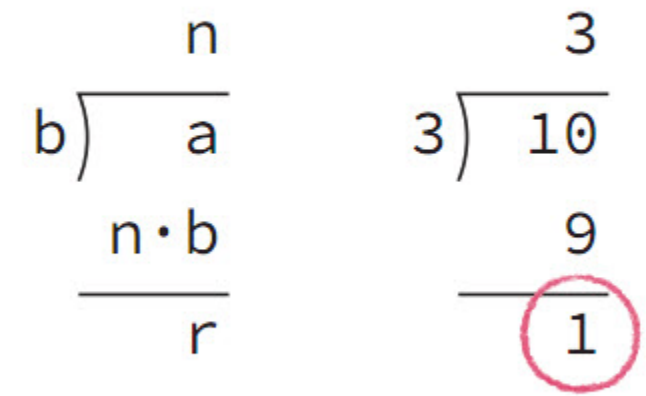
산술연산자는 + , -, *, /, %

- % 나머지remainder, modulus 연산자
  - 피연산자로 정수만 가능
  - 나누기 연산식 10 / 4는 연산값이 2
- 실수끼리의 연산 10.0 / 4.0
  - 결과는 정상적으로 2.5
- 나머지 연산식 a % b
  - 결과는 a를 b로 나눈 나머지 값
  - %의 피연산자는 반드시 정수
  - 실수이면 오류가 발생

나누기 연산과 나머지 연산의 이해

a % b 연산값: r

a / b 연산값: n



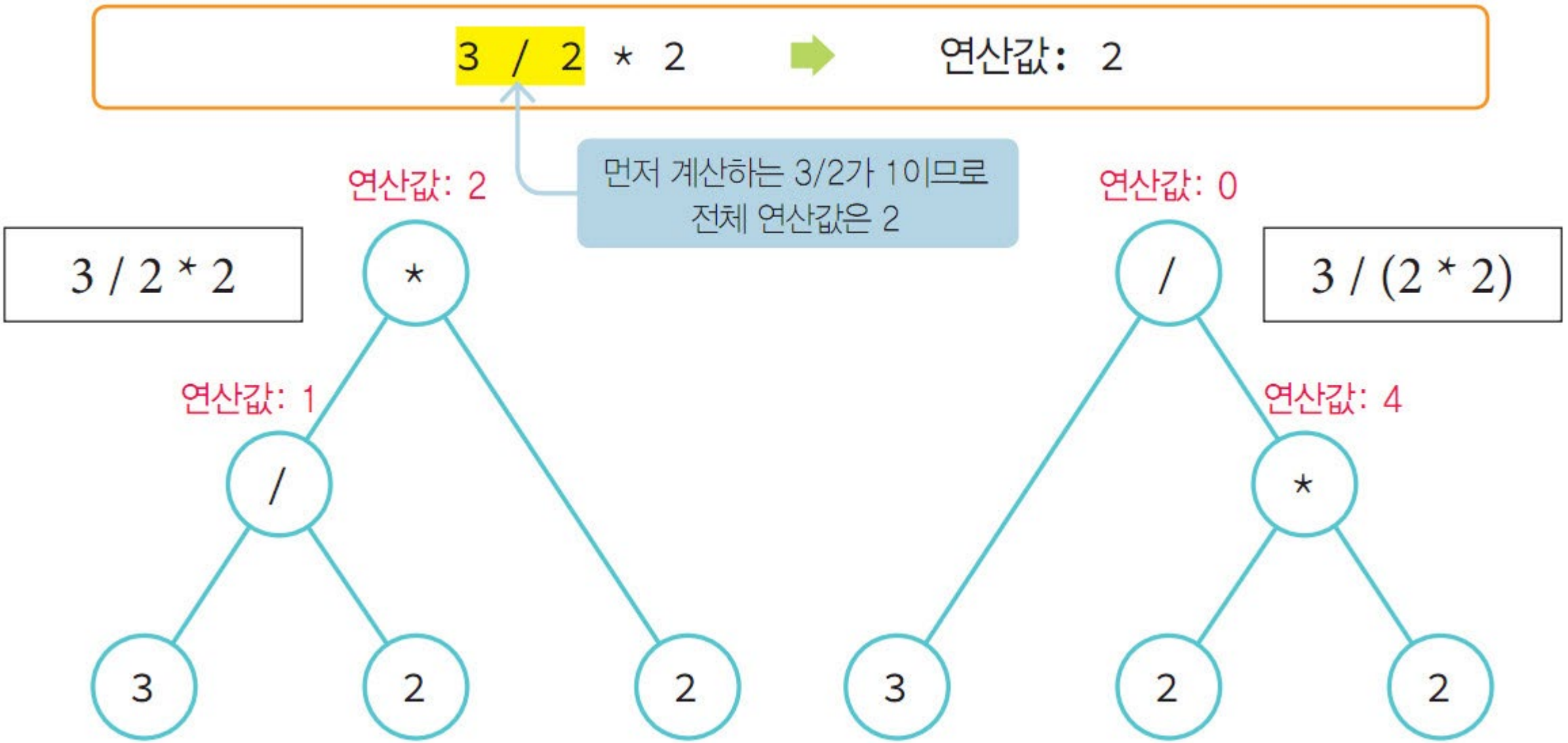
a를 b로 나눈 나머지인 r과 몫인 n

$$a = b * n + r$$

10 % 3 결과: 1

10 / 3 결과: 3

산술연산에서 결합성


3 / 2 * 2
연산값: 2
먼저 계산하는 3/2가 1이므로
전체 연산값은 2
3 / 2 * 2
연산값: 2
*
연산값: 1
/
3
2
2
3 / (2 * 2)
연산값: 0
/
연산값: 4
*
3
2
2

부호 연산자: 단항 연산자

- 연산식 +3, -4.5, -a
- 수, 변수의 부호로 표기하는 연산자

연산식	설명	연산값
-a	부호연산자	10
2 - a	빼기연산자	-3
2 * a + 3	(2 * a) + 3	13
10 - a / 2	10 - (a / 2)	8
10 % a + 10 / a	(10 % a) + (10 / a)	2

연산식	설명	연산값
-b + 2.5	부호연산자	0.0
a / b + b * 2	(a / b) + (b * 2)	5.0
a * 2 / b - a	((a * 2) / b) - a	-1.0
a + b * 2 /a	a + ((b * 2) /a)	6.0
a % b	실수는 %에 오류	오류

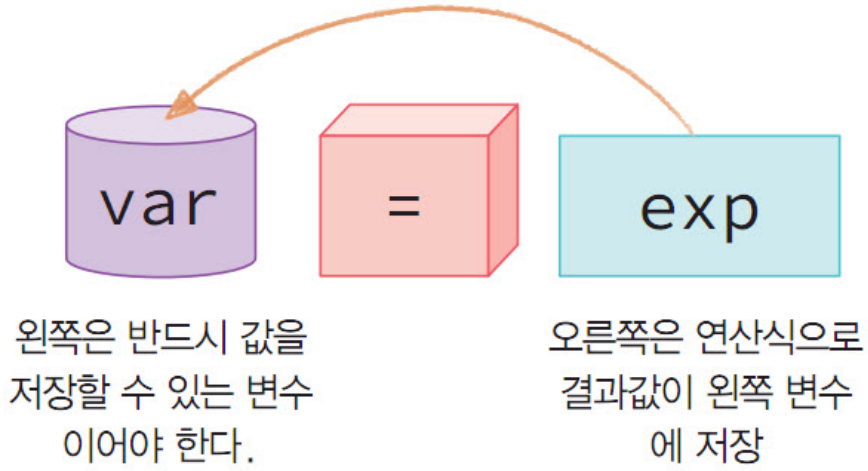
Source Code #01: assignment.c

```c
// file: assignment.c

#include <stdio.h>

int main(void)
{
    int a, b, c;
    a = b = c = 5;  //(a = (b = (c = 5)))

    printf("a = a + 2 ==> %d\n", a = a + 2);
    printf("a ==> %d\n", a);
    printf("a = b + c ==> %d\n", a = b + c);
    printf("a ==> %d\n", a);

    return 0;
}
```

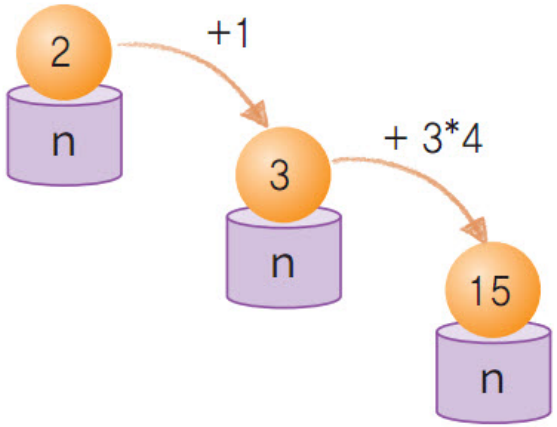
```
a = a + 2 ==> 7
a ==> 7
a = b + c ==> 10
a ==> 10
```

대입연산자 =

- 대입연산자assignment operator =
  - 연산자 오른쪽의 연산값을 변수에 저장하는 연산자
- 연산자 오른쪽에 위치한 연산식 exp를 계산
  - 오른쪽을 의미하는 right 단어에서 r-value
- 그 결과를 왼쪽 변수 var에 저장
  - 대입연산자의 왼쪽 부분에는 반드시 하나의 변수만이 올 수 있음
  - 이 하나의 변수를 왼쪽을 의미하는 left 단어에서 l-value

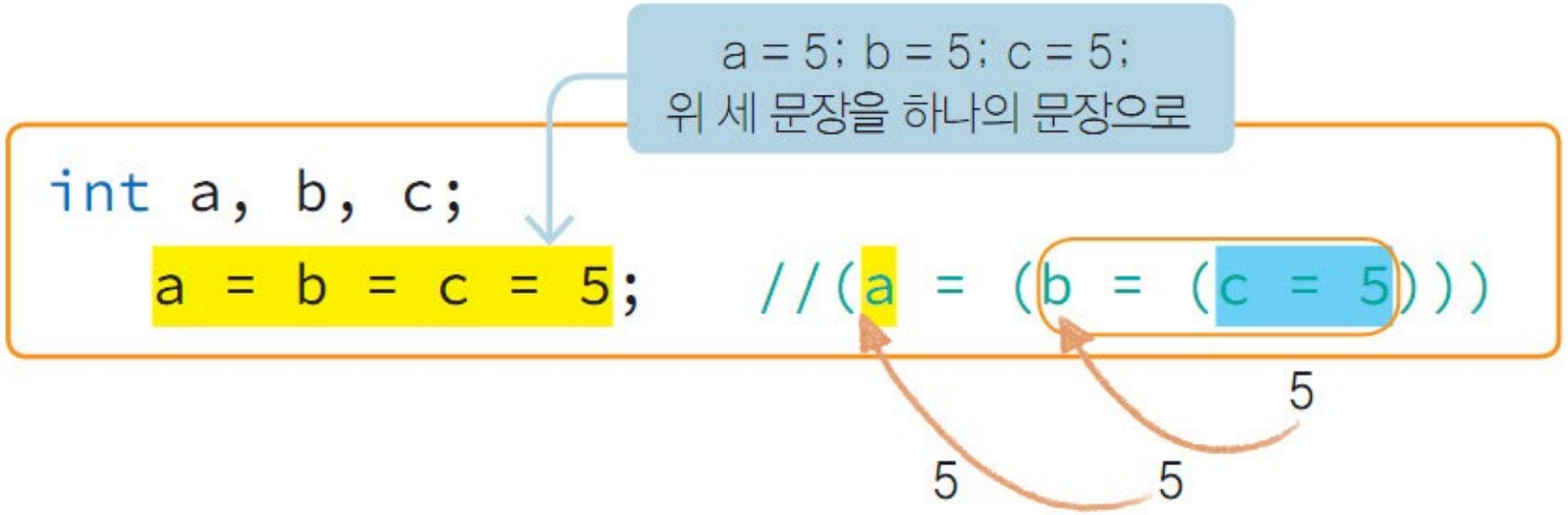



```
n = 2;

n = n + 1;

n = n + 3*4;
```



대입연산자 =

- a = b = c = 5와 같은 중첩된 대입문
  - 대입연산자의 결합성이 '오른쪽에서 왼쪽(←)'이므로 많은 변수에 동일한 값을 한 번에 대입
  - 즉 변수 a, b, c 모두 5가 저장
  - 연산값은 마지막으로 a에 저장된 값인 5



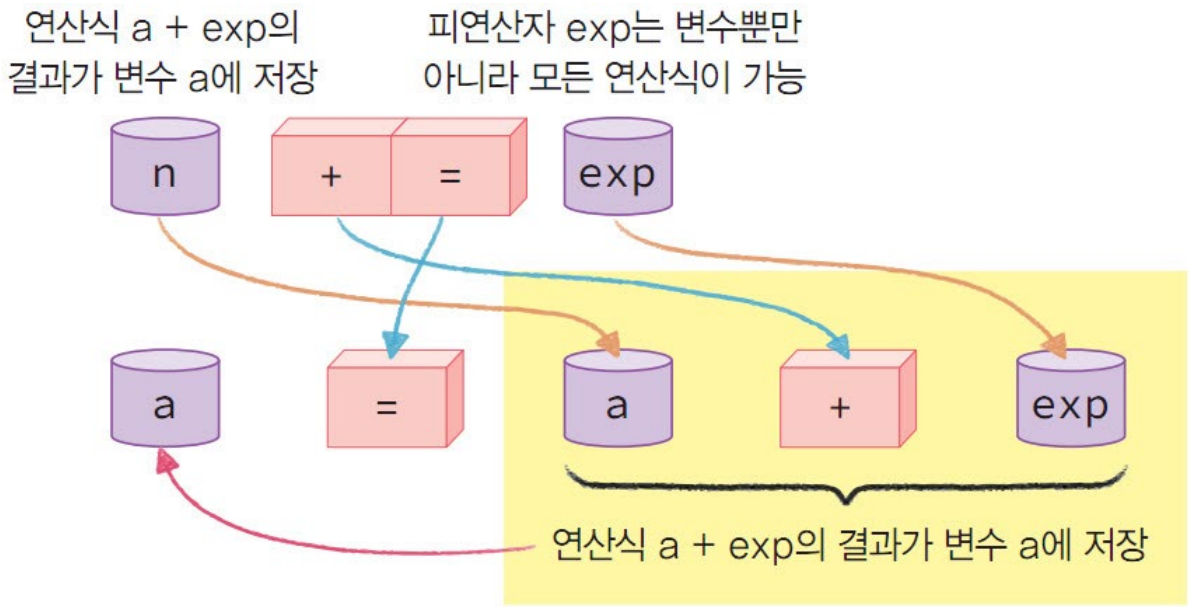
Source Code #02: compoundassign.c

```c
// file: compoundassign.c
#define _CRT_SECURE_NO_WARNINGS //scanf() 오류를 방지하기 위한 상수 정의

#include <stdio.h>

int main(void)
{
    int x = 5, y = 10;

    printf("두 정수를 입력 >> ", &x, &y);
    scanf("%d%d", &x, &y);

    printf("The addition is: %d\n", x += y);
    printf("x = %d, y = %d\n", x, y);
    printf("The subtraction is: %d\n", x -= y);
    printf("x = %d, y = %d\n", x, y);
    printf("The multiplication is: %d\n", x *= y);
    printf("x = %d, y = %d\n", x, y);
    printf("The division is: %d\n", x /= y);
    printf("x = %d, y = %d\n", x, y);
    printf("The remainder is: %d\n", x %= y);
    printf("x = %d, y = %d\n", x, y);
    printf("x *= x + y is: %d\n", x *= x + y);
    printf("x = %d, y = %d\n", x, y);

    return 0;
}
```

```
두 정수를 입력 >> 10 5
The addition is: 15
x = 15, y = 5
The subtraction is: 10
x = 10, y = 5
The multiplication is: 50
x = 50, y = 5
The division is: 10
x = 10, y = 5
The remainder is: 0
x = 0, y = 5
x *= x + y is: 0
x = 0, y = 5
```

축약 대입연산자 =

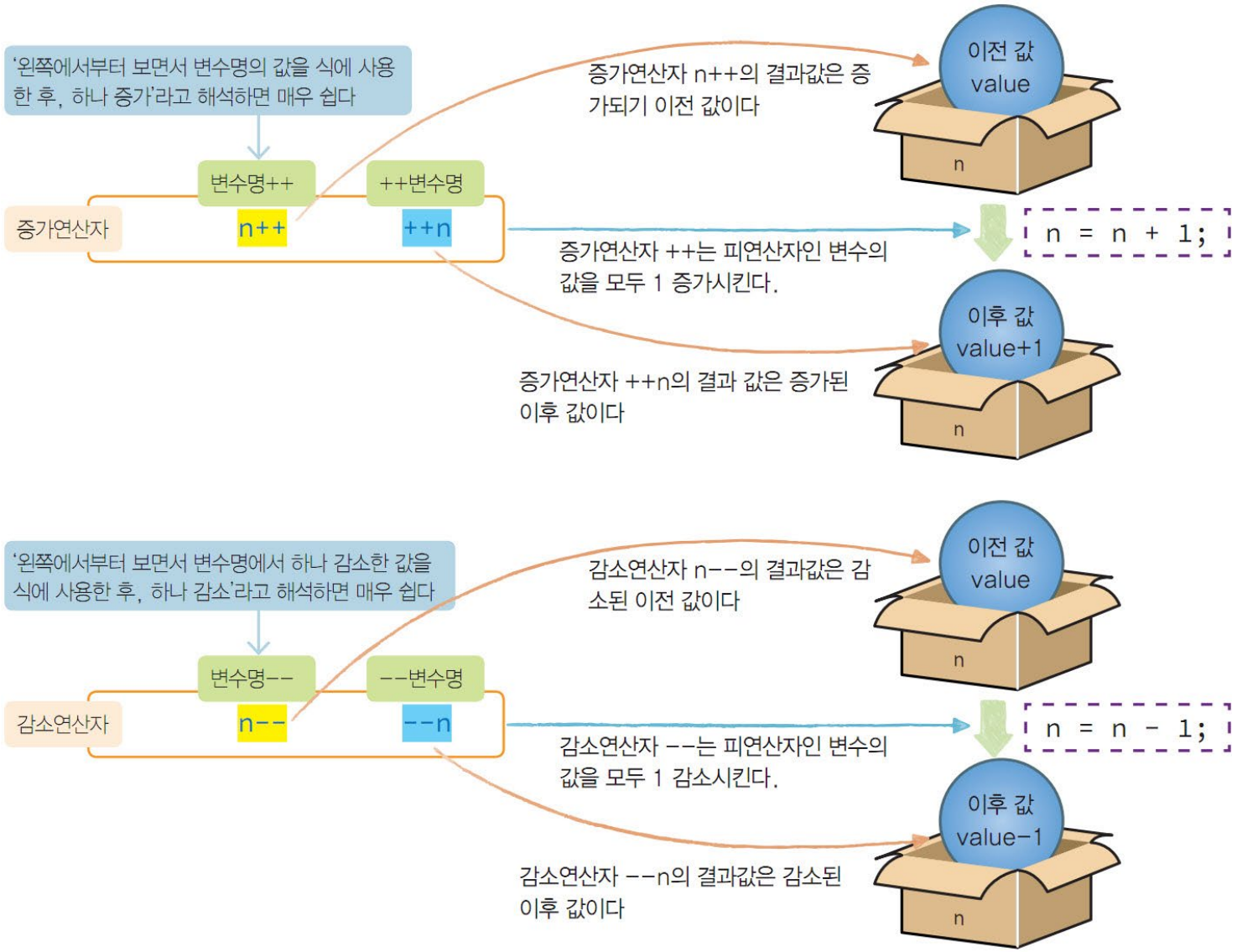
- 대입연산식 a = a+b
  - 중복된 a를 생략하고 간결하게 a += b
- -=, *=, /=, %=을 축약 대입연산자
  - 산술연산자와 대입연산자를 이어 붙인 연산자 +=
  - 즉 a += 2는 a = a+2의 대입연산을 의미



연산자 ++, --

- 증가연산자 ++
  - 변수값을 각각 1 증가시키고
- 감소연산자 –
  - 1 감소
- n++와 ++n
  - 모두 n=n+1의 기능을 수행
- n--와 --n
  - n=n-1의 기능을 수행

증가 감소 연산자


'왼쪽에서부터 보면서 변수명의 값을 식에 사용한 후, 하나 증가'라고 해석하면 매우 쉽다
변수명++
++변수명
증가연산자
n++
++n
증가연산자 n++의 결과값은 증가되기 이전 값이다
이전 값
value
n
n = n + 1;
증가연산자 ++는 피연산자인 변수의 값을 모두 1 증가시킨다.
이후 값
value+1
n
증가연산자 ++n의 결과 값은 증가된 이후 값이다
'왼쪽에서부터 보면서 변수명에서 하나 감소한 값을 식에 사용한 후, 하나 감소'라고 해석하면 매우 쉽다
변수명--
--변수명
감소연산자
n--
--n
감소연산자 n--의 결과값은 감소된 이전 값이다
이전 값
value
n
n = n - 1;
감소연산자 --는 피연산자인 변수의 값을 모두 1 감소시킨다.
이후 값
value-1
n
감소연산자 --n의 결과값은 감소된 이후 값이다

증가 감소 연산자

- n++: 후위|postfix
  - 1 증가되기 전 값이 연산 결과값
- ++n: 전위|prefix
  - 1 증가된 값이 연산 결과값
- 주의
  - 연산자 기호 중간에 공백이 들어갈 수 없으며
  - 증감연산자는 변수만을 피연산자로 사용할 수 있으며
  - 상수나 일반 수식을 피연산자로 사용할 수 없음

증가 감소 연산자

```
int n = 10;

printf("%d\n", n++);
printf("%d\n", n);
```

출력

```
10
11
```

```
int n = 10;

printf("%d\n", ++n);
printf("%d\n", n);
```

출력

```
10
11
```

```
int n = 10;

printf("%d\n", n--);
printf("%d\n", n);
```

출력

```
10
9
```

```
int n = 10;

printf("%d\n", --n);
printf("%d\n", n);
```

출력

```
9
9
```

잘못된 사용 예

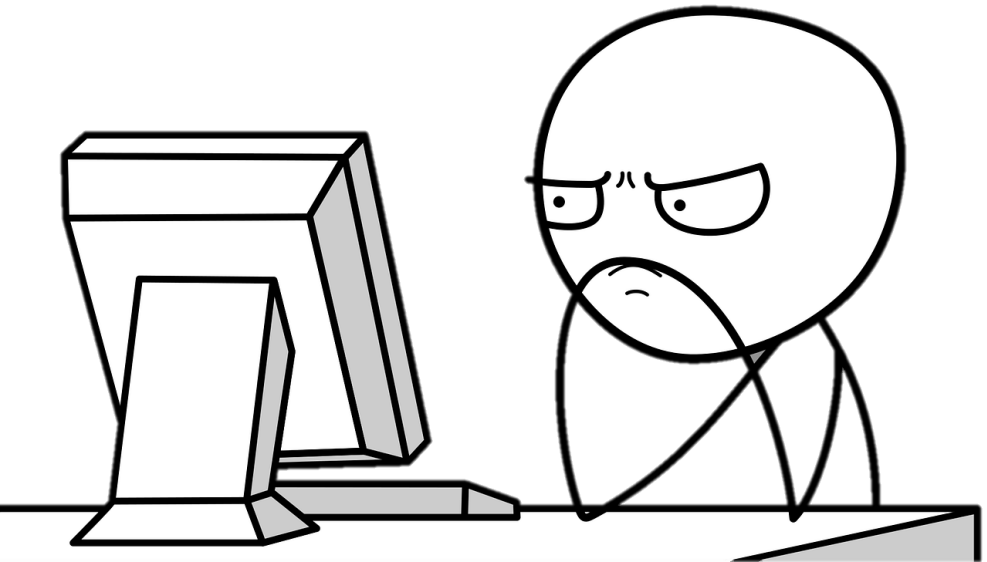
```
int a = 10;
++300;          //상수에는 증감연산자를 사용할 수 없다.
(a+1)--;        //일반 수식에는 증감연산자를 사용할 수 없다.

//하나의 연산식에 동일한 변수의 증감연산자는 사용하지 말자.
a = ++a * a--;
```

Lab #01: 표준입력된 두 실수의 산술연산 출력

- 다음 정보를 이용하여 두 실수의 합, 차, 곱, 나누기의 과정과 결과를 출력하는 프로그램
  - 자료형 double의 변수 a와 b 에 표준입력으로 받아 저장
  - 다음 결과 창과 같은 서식(실수는 모두 폭 8, 정밀도는 2로)으로 출력
- 실행결과
  - 산술연산을 수행할 두 실수를 입력하세요
  - 54.987, 4.87654
  - 54.99 + 4.88 ==> 59.86
  - 00054.99 - 00004.88 ==> 50.11
  - +54.99 * +4.88 ==> 268.15
  - 54.99 / 4.88 ==> 11.28

```c
// file: tworealop.c
#define _CRT_SECURE_NO_WARNINGS //scanf() 오류를 방지하기 위한 상수 정의

#include <stdio.h>

int main(void)
{
    double a = 0, b = 0;

    printf("산술연산을 수행할 두 실수를 입력하세요\n");
    scanf("        ", &a, &b);

    printf("%8.2f + %8.2f ==> %8.2f\n", a, b, a + b);
    printf("%08.2f - %08.2f ==> %8.2f\n", a, b, a - b);
    printf("                ==> %8.2f\n",          );
    printf("                ==> %8.2f\n",          );

    return 0;
}
```

2. 관계와 논리, 조건과 비트연산자



Source Code #03: relation.c

```c
// file: relation.c

#include <stdio.h>

int main(void)
{
    printf("(3 > 4) 결과값: %d\n", (3 > 4));
    printf("(3 < 4.0) 결과값: %d\n", (3 < 4.0));
    printf("('a' <= 'b') 결과값: %d\n", ('a' <= 'b'));
    printf("('Z' <= 'a') 결과값: %d\n", ('Z' <= 'a'));
    printf("(4.27 >= 4.35) 결과값: %d\n", (4.27 >= 4.35));
    printf("(4 != 4.0) 결과값: %d\n", (4 != 4.0));
    printf("(4.0F == 4.0) 결과값: %d\n", (4.0F == 4.0));

    return 0;
}
```

```
(3 > 4) 결과값: 0
(3 < 4.0) 결과값: 1
('a' <= 'b') 결과값: 1
('Z' <= 'a') 결과값: 1
(4.27 >= 4.35) 결과값: 0
(4 != 4.0) 결과값: 0
(4.0F == 4.0) 결과값: 1
```

두 피연산자의 크기 비교

- 두 피연산자의 크기를 비교하기 위한 연산자: 관계연산자
  - 비교 결과가 참이면 0, 거짓이면 1
- 관계연산자 !=, >=, <=는 연산 기호의 순서가 명확
  - 관계연산자 ==는 대입연산자 =와 혼동하지 않도록 주의
  - 정수형, 실수형, 문자형 등이 피연산자가 될 수 있음
- 피연산자가 문자인 경우, 문자 코드값에 대한 비교의 결과
  - 문자 'a'는 코드값이 97이고 문자 'Z'는 코드값이 90이므로 연산식 ('Z' < 'a')는 1인 참을 의미
  - 즉 문자 '0' < '1' < '2' < '3' < … < '9'인 관계가 있으며
  - 'a' < 'b' < 'c' < … < 'x' < 'y' < 'z' 관계가 있고,
  - 대문자도 마찬가지이며, 'Z' < 'a'로 소문자는 대문자보다 모두 큼

관계연산자의 종류와 사용

연산자	연산식	의미	예제	연산(결과)값
>	x > y	x가 y보다 큰가?	3 > 5	0(거짓이면)
>=	x >= y	x가 y보다 큰거나 같은가?	5-4 >= 0	1(참이면)
<	x < y	x가 y보다 작은가?	'a' < 'b'	1(참이면)
<=	x <= y	x가 y보다 작거나 같은가?	3.43 <= 5.862	1(참이면)
!=	x != y	x와 y가 다른가?	5-4 != 3/2	0(거짓이면)
==	x == y	x가 y가 같은가?	'%' == 'A'	0(거짓이면)

Source Code #04: logic.c

```c
// file: logic.c

#include <stdio.h>

int main(void)
{
    char null = '\0', a = 'a';
    int zero = 0, n = 10;
    double dzero = 0.0, x = 3.56;

    printf("%d ", !zero);
    printf("%d ", zero && x);
    printf("%d\n", dzero || null);
    printf("%d ", n && x);
    printf("%d ", a || null);
    printf("%d\n", "java" && "C Lang");

    return 0;
}
```

```
1 0 0
1 1 1
```

논리연산자

- 세 가지의 논리연산자 &&, ||, !을 제공
  - 논리연산자 &&, ||, !은 각각 and, or, not 의미
  - 결과가 참이면 1 거짓이면 0을 반환
- C 언어에서 참과 거짓의 논리형은 따로 없음
  - 0, 0.0, \0은 거짓을 의미
  - 0이 아닌 모든 정수와 실수, 그리고 널(null) 문자 '\0'가 아닌 모든 문자와 문자열은 모두 참을 의미
- 논리연산자 &&
  - 두 피연산자가 모두 참(0이 아니어야)이면 결과가 1(참)이며
  - 나머지 경우는 모두 0
- 논리연산자 ||
  - 두 피연산자 중에서 하나만 참(0이 아니어야)이면 1이고,
  - 모두 0(거짓)이면 0
- 논리연산자 !
  - 단항연산자로 피연산자가 0이면 결과는 1이고
  - 참(0이 아닌 값)이면 결과는 0

논리연산자의 연산 결과

x	y	x && y	x || y	!x
0(거짓)	0(거짓)	0	0	1
0(거짓)	Ø(0이 아닌 값)	0	1	1
Ø(0이 아닌 값)	0(거짓)	0	1	0
Ø(0이 아닌 값)	Ø(0이 아닌 값)	1	1	0

```
21 && 3                1
!2 && 'a'              0
3>4 && 4>=2            0
1 || '\0'              1
2>=1 || 3 <=0          1
0.0 || 2-2             0
!0                     1
```

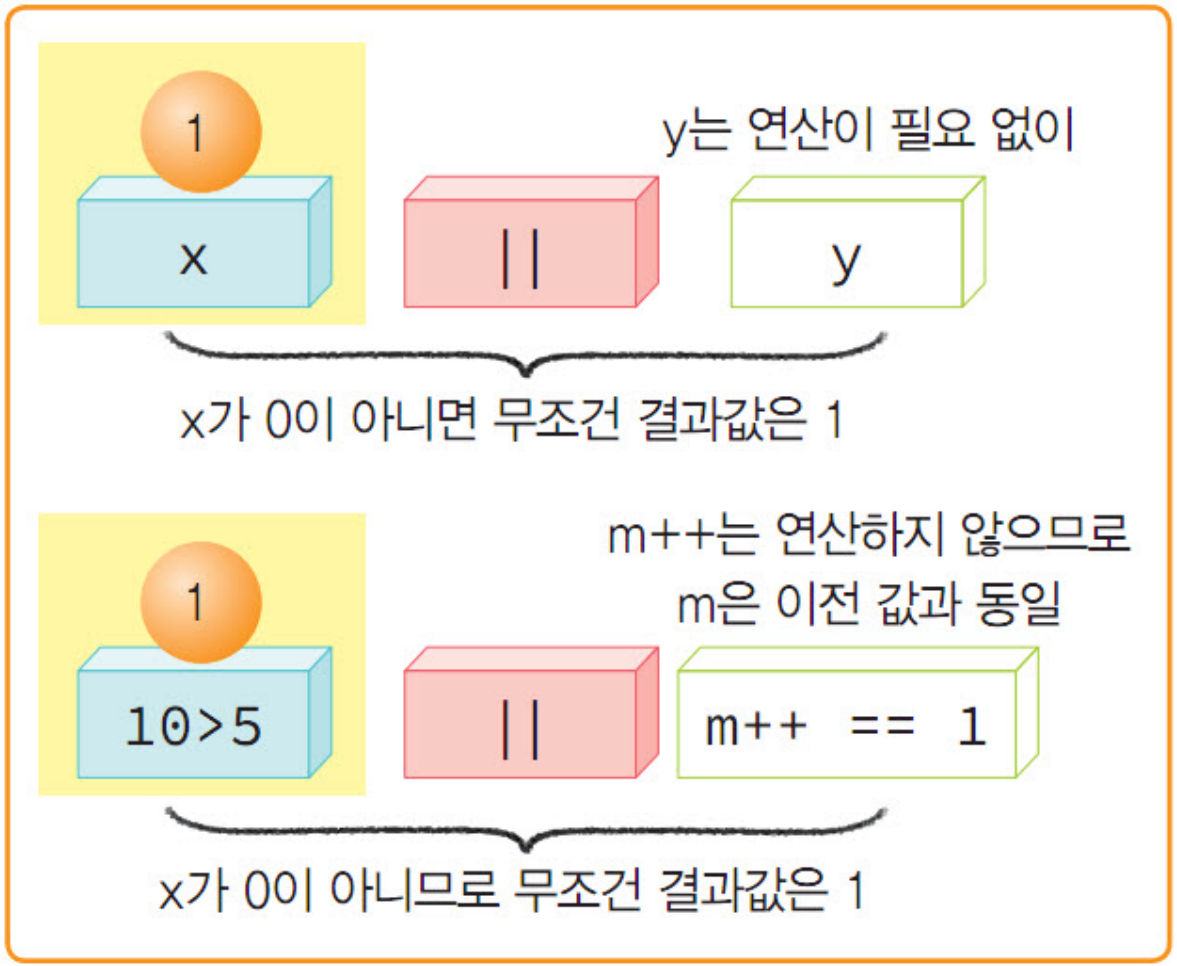
Source Code #05: shorteval.c

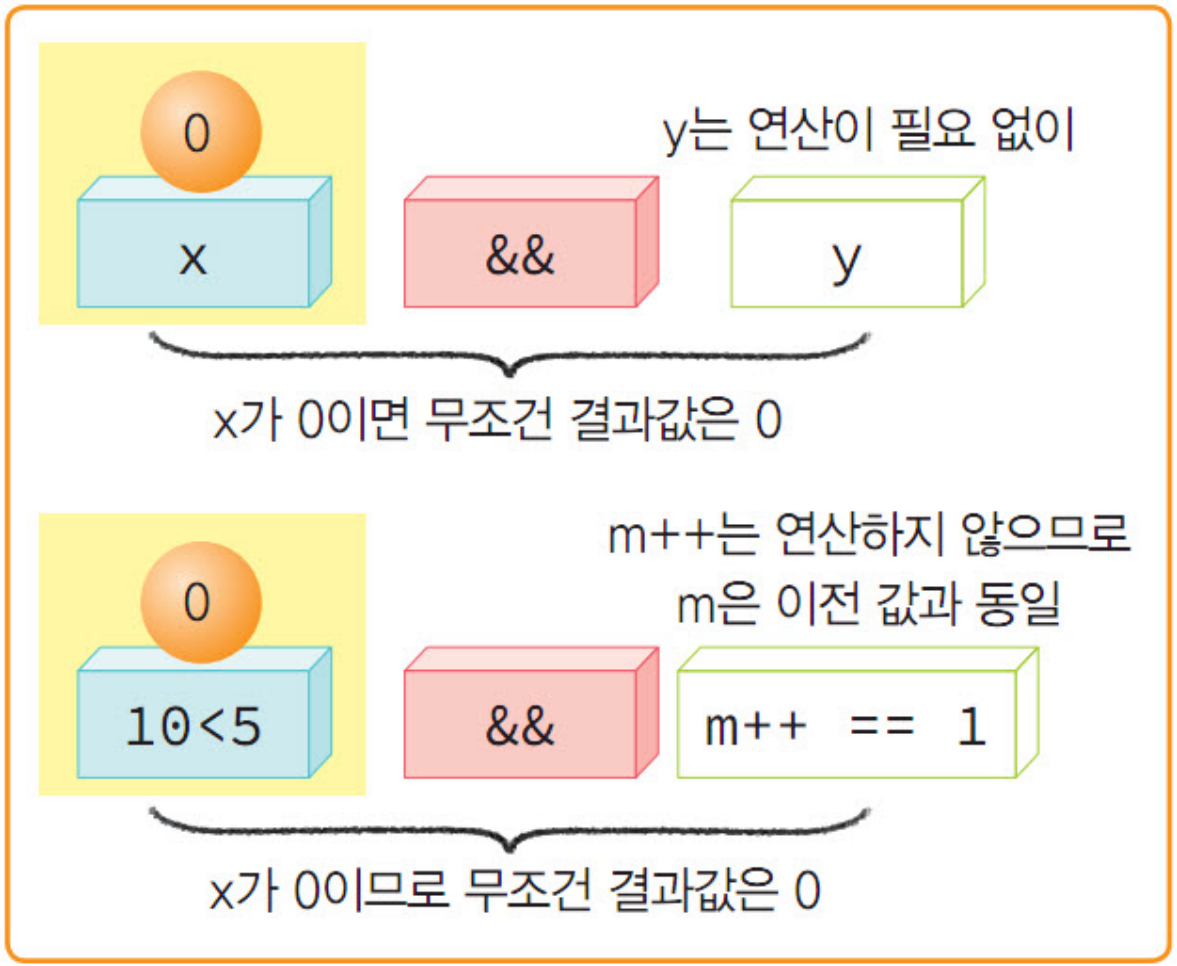
```c
// file: shorteval.c

#include <stdio.h>

int main(void)
{
    int a = 10, b = 5, m = 1;
    int result;

    result = (a < b) && (m++ == 1);          m=1 result=0
    printf("m=%d result=%d\n", m, result);   m=1 result=1

    result = (a > b) || (--m == 0);
    printf("m=%d result=%d\n", m, result);

    return 0;
}
```

단축 평가

- 단축 평가short circuit evaluation
  - 논리연산자 &&와 ||는 피연산자 두 개 중에서 왼쪽 피연산자만으로 논리연산 결과가 결정
  - 오른쪽 피연산자는 평가하지 않음
- (x && y) 연산식
  - x의 값이 0(거짓)이라면 y의 값을 평가하지 않고 연산 (x && y) 결과는 0
- (x || y) 수식
  - x가 0이 아니(참)라면 더 이상 y의 값을 평가하지 않고 연산식 (x || y)는 1이라고 평가

단축 평가


1
x
||
y
y는 연산이 필요 없이
x가 0이 아니면 무조건 결과값은 1
1
10>5
||
m++ == 1
m++는 연산하지 않으므로
m은 이전 값과 동일
x가 0이 아니므로 무조건 결과값은 1


0
x
&&
y
y는 연산이 필요 없이
x가 0이면 무조건 결과값은 0
0
10<5
&&
m++ == 1
m++는 연산하지 않으므로
m은 이전 값과 동일
x가 0이므로 무조건 결과값은 0

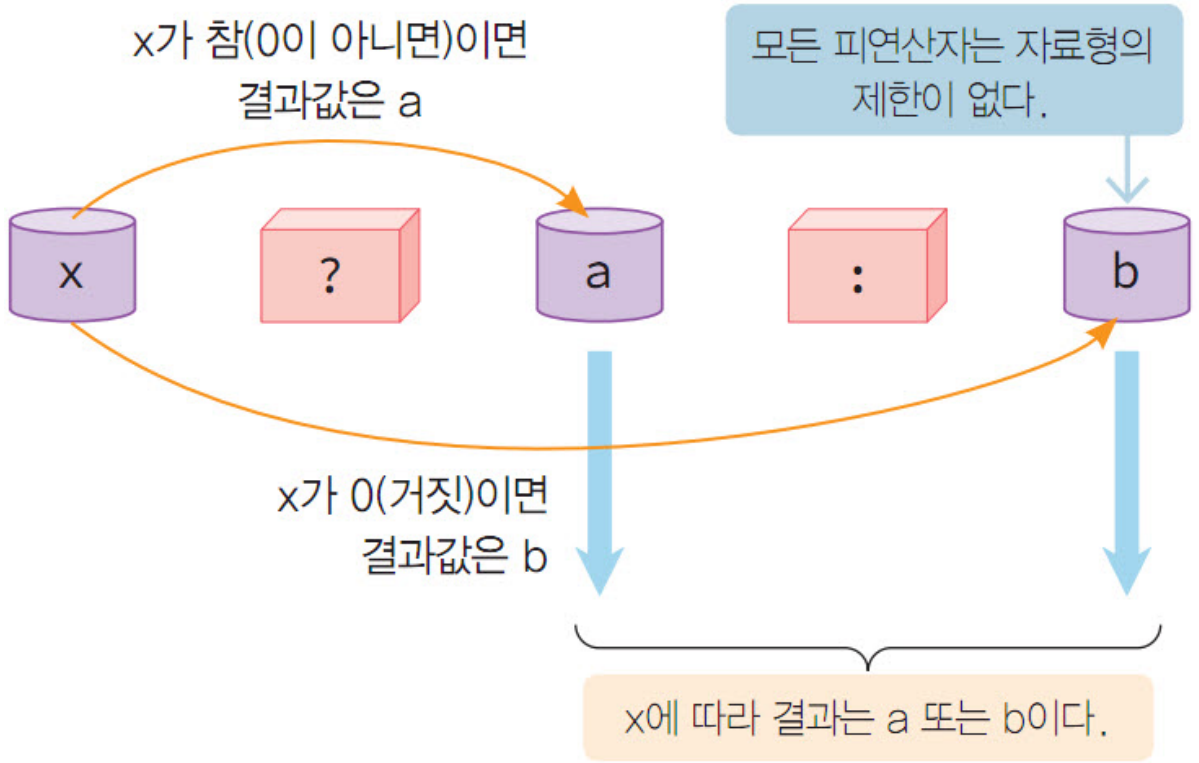
Source Code #06: condition.c

```c
// file: condition.c
#define _CRT_SECURE_NO_WARNINGS //scanf() 오류를 방지하기 위한 상수 정의

#include <stdio.h>

int main(void)
{
    int a = 0, b = 0;
    printf("두 정수를 입력 >> ");
    scanf("%d%d", &a, &b);

    printf("최대값: %d ", (a > b) ? a : b);
    printf("최소값: %d\n", (a < b) ? a : b);
    printf("절대값: %d ", (a > 0) ? a : -a);
    printf("절대값: %d\n", (b > 0) ? b : -b);

    ((a % 2) == 0) ? printf("짝수 ") : printf("홀수 ");;
    printf("%s\n", ((b % 2) == 0) ? "짝수" : "홀수");

    return 0;
}
```

```
두 정수를 입력 >> 8 -9
최대값: 8 최소값: -9
절대값: 8 절대값: 9
짝수 홀수
```

조건 연산자

- 연산자 ? :
  - 조건연산자는 조건에 따라 주어진 피연산자가 결과값이 되는 삼항연산자
- 즉 연산식 (x ? a : b)에서 피연산자는 x, a, b 세 개
  - 피연산자인 x가 참이면(0이 아니면) 결과는 a 이며, x가 0이면(거짓) 결과는 b

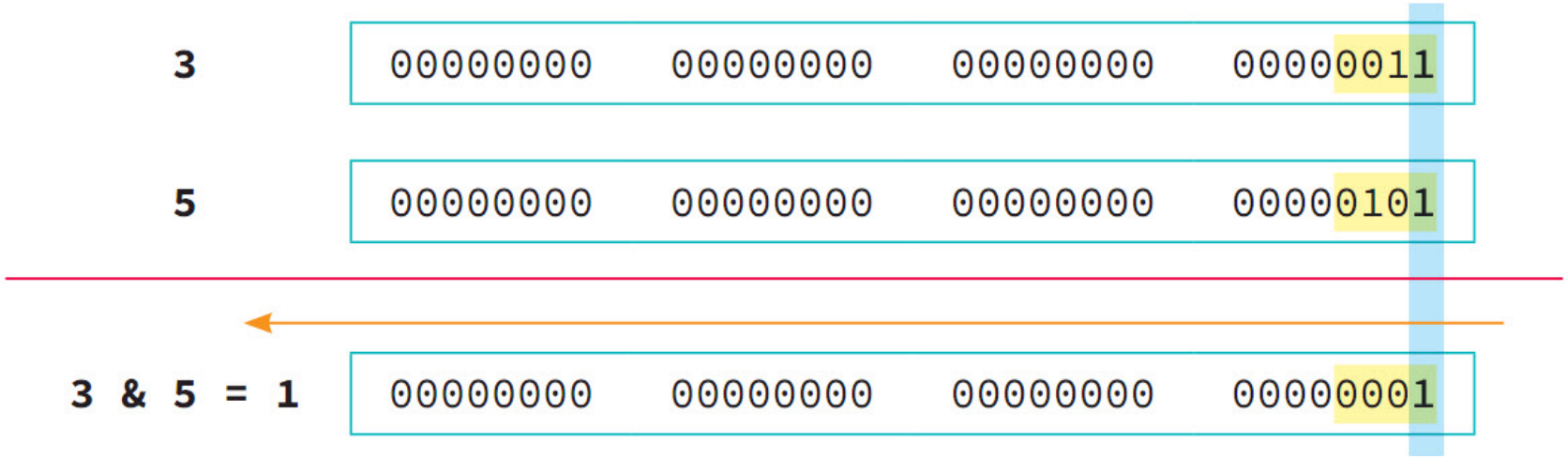


조건 연산자

```
max = (a > b) ? a : b;          //최대값 반환 조건연산
max = (a < b) ? b : a;          //최대값 반환 조건연산
min = (a > b) ? b : a;          //최소값 반환 조건연산
min = (a < b) ? a : b;          //최소값 반환 조건연산
absolute = (a > 0) ? a : -a;    //절대값 반환 조건연산
absolute = (a < 0) ? -a : a;    //절대값 반환 조건연산
```

비트 연산자

- 정수의 비트 중심bitwise 연산자를 제공
  - 비트 논리연산자와 이동연산자가 제공
- 비트 논리 연산자
  - 피연산자 정수값을 비트 단위로 논리 연산을 수행하는 연산자



&, |, ^, ~

- 연산자 ~
  - ~5와 같이 연산자 ~가 피연산자인 4 앞에 위치하는 전위인 단항 연산자
- 나머지는 모두 (3 | 4)처럼 피연산자가 두 개인 이항 연산자
  - 피연산자의 자료형은 정수형에 해당하는 char, int, long, long long
  - 각 피연산자를 int 형으로 변환하여 연산하며 결과도 int 형

x(비트1)	y(비트2)	x & y	x | y	x ^ y	~x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

보수 연산 예제

피연산자		보수 연산	
수	비트표현(이진수)	보수 연산 결과	십진수
1	000000000 000000000 000000000 000000001	11111111 11111111 11111111 11111110	~1 = -2
4	000000000 000000000 000000000 000000100	11111111 11111111 11111111 11111011	~4 = -5

Source Code #07: bitlogic.c

```c
// bitlogic.c

#include <stdio.h>

int main(void)
{
    int x = 127;

    printf("%5d -> %08x\n", x, x);
    printf("x & 1 -> %08x\n", x & 1);
    printf("x | 1 -> %08x\n", x | 1);
    printf("x ^ 1 -> %08x\n", x ^ 1);
    printf("~(-1) -> %08x\n", ~(-1));
    printf(" ~1   -> %08x\n", ~1);

    return 0;
}
```

```
  127 -> 0000007f
x & 1 -> 00000001
x | 1 -> 0000007f
x ^ 1 -> 0000007e
~(-1) -> 00000000
 ~1   -> fffffffe
```

비트 연산자

- 컴퓨터에서 정수의 음수 표현 방법
  - 보수를 이용하는 방법을 사용
  - 즉 양수 정수 a에서 음수인 -a의 비트 표현은 2의 보수 표현인 (~a + 1)
  - 즉 -1은 ((~1)+1)로, 정수 -1을 비트로 표현하면 32비트가 모두 1인 정수
- 비트 논리 연산식 x&-1
  - 정수 x를 -1로 논리 and연산을 수행하는 식으로 결과는 x
- 비트 논리 연산식 x | 0
  - 정수 x를 0으로 논리 or연산을 수행하는 식으로 결과는 x

다양한 비트 논리연산식

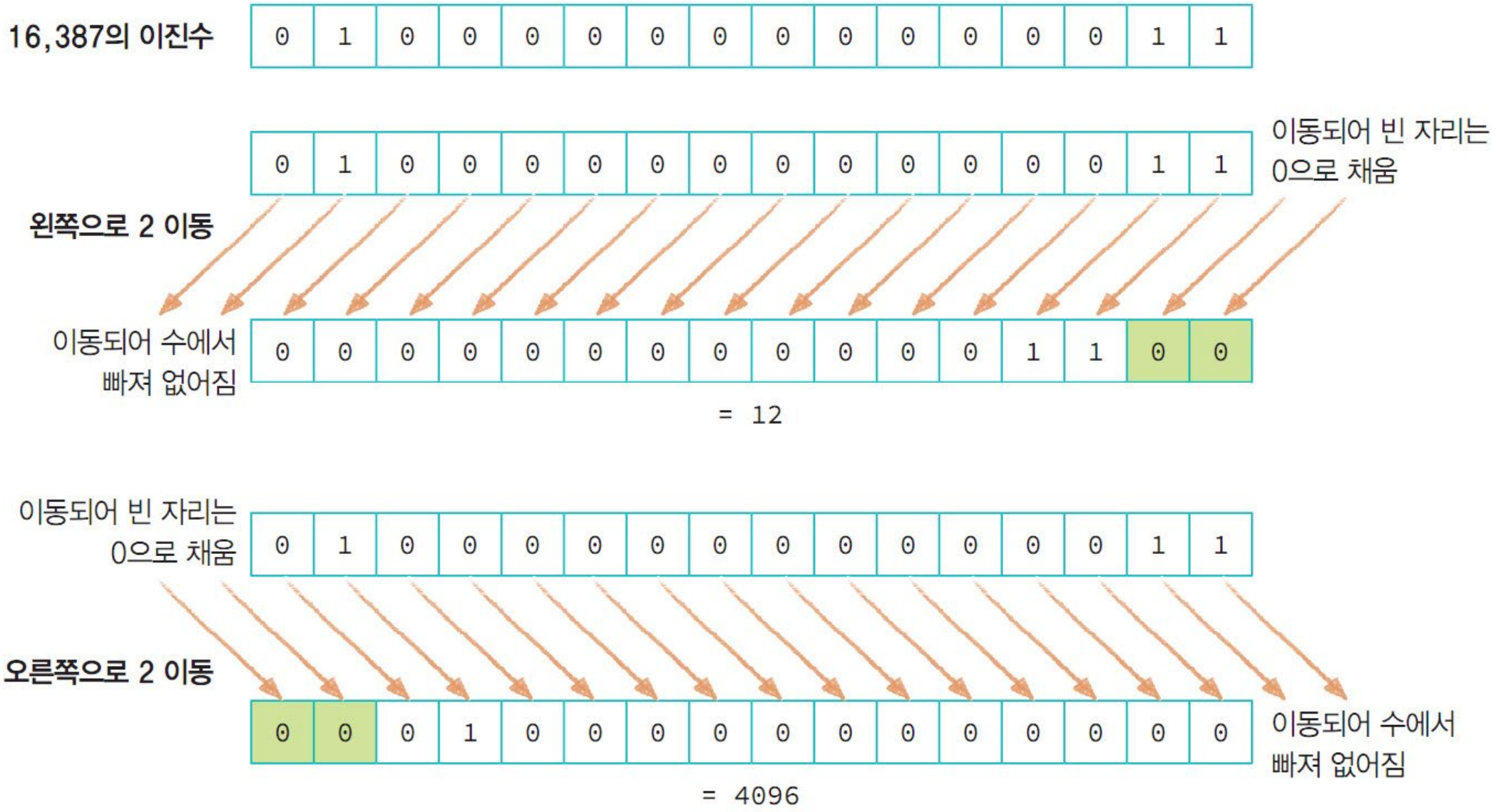
연산식	설명	연산값
1 & 2	0001 & 0010	0
3 | 4	0011 | 0100	7
3 ^ 4	0011 ^ 0100	7
~2	-2 == ~2 + 1	-3

연산식	설명	연산값
1 & 3	0001 & 0011	1
3 & 5	0011 | 0101	7
3 ^ 5	0011 | 0101	6
~3	-3 == ~3 + 1	-4

비트 이동연산자bit shift operators >>, <<

- 연산자의 방향인 왼쪽이나 오른쪽으로, 비트 단위로 줄줄이 이동시키는 연산자
- <<: 오른쪽LSB: Least Significant Bit 빈 자리가 생겨 모두 0으로 채워짐
- >>: 왼쪽MSB: Most Significant Bit 빈 자리는 원래의 부호비트에 따라 0또는 1이 채워짐
- 실제 연산에서는 int 형의 크기인 32비트의 비트에서 연산을 수행

비트 논리 연산 16387 << 2와 16387 >> 2



비트 이동 연산자

연산자	이름	사용	연산 방법	새로 채워지는 비트
>>	right shift	op1 >> op2	op1을 오른쪽으로 op2 비트만큼 이동	가장 왼쪽 비트인 부호 비트는 원래의 부호 비트로 채움
<<	left shift	op1 << op2	op1을 왼쪽으로 op2 비트만큼 이동	가장 오른쪽 비트를 모두 0으로 채움

Source Code #08: shift.c

```c
// shift.c

#include <stdio.h>

int main(void)
{
    int x = 16391;

    printf("%6d --> %08x\n", x, x);
    printf("x >> 1 --> %d, %08x\n", x >> 1, x >> 1);
    printf("x >> 2 --> %d, %08x\n", x >> 2, x >> 2);
    printf("x >> 2 --> %d, %08x\n", x >> 3, x >> 3);
    printf("x << 2 --> %d, %08x\n", x << 2, x << 2);
    printf("x << 2 --> %d, %08x\n", x << 3, x << 3);

    return;
}
```

```
 16391 --> 00004007
x >> 1 --> 8195, 00002003
x >> 2 --> 4097, 00001001
x >> 2 --> 2048, 00000800
x << 2 --> 65564, 0001001c
x << 2 --> 131128, 00020038
```

비트 이동 연산자

- 왼쪽 이동연산자에 의해 최상위 부호 비트가 0에서 1로, 1에서 0으로 바뀔 수 있으므로 왼쪽 이동연산자에 의해 항상 2배씩 커지지는 않음
- 음수 홀수에 대한 오른쪽 이동연산자 >>도 -1을 한 짝수를 2로 나눈 결과

연산식	설명	연산값
15 << 1	0000 1111 << 1 0001 1110	30
0x30000000 << 2	최상위 4비트: 0011 0… << 2 1100 0…	-1073741824 0XC0000000
-30 >> 1	짝수이면 2로 나누기	-15

연산식	설명	연산값
15 << 2	0000 1111 << 2 0011 1100	60
-30 << 2	음수로 4배 커짐	-120
-30 >> 2	한 비트 이동마다. 음수 홀수는 -(a+1)한 수를 2로 나누기	-8

Lab #02: 표준입력으로 받은 두 정수의 비트 연산 수행 출력

- 표준입력으로 받은 두 정수의 6개 비트 연산을 수행하여 결과를 출력하는 프로그램
- 비트 연산은 &, |, ^, ~, >>, << 연산을 수행
- 단항 연산은 첫 번째 변수에 대한 연산 수행

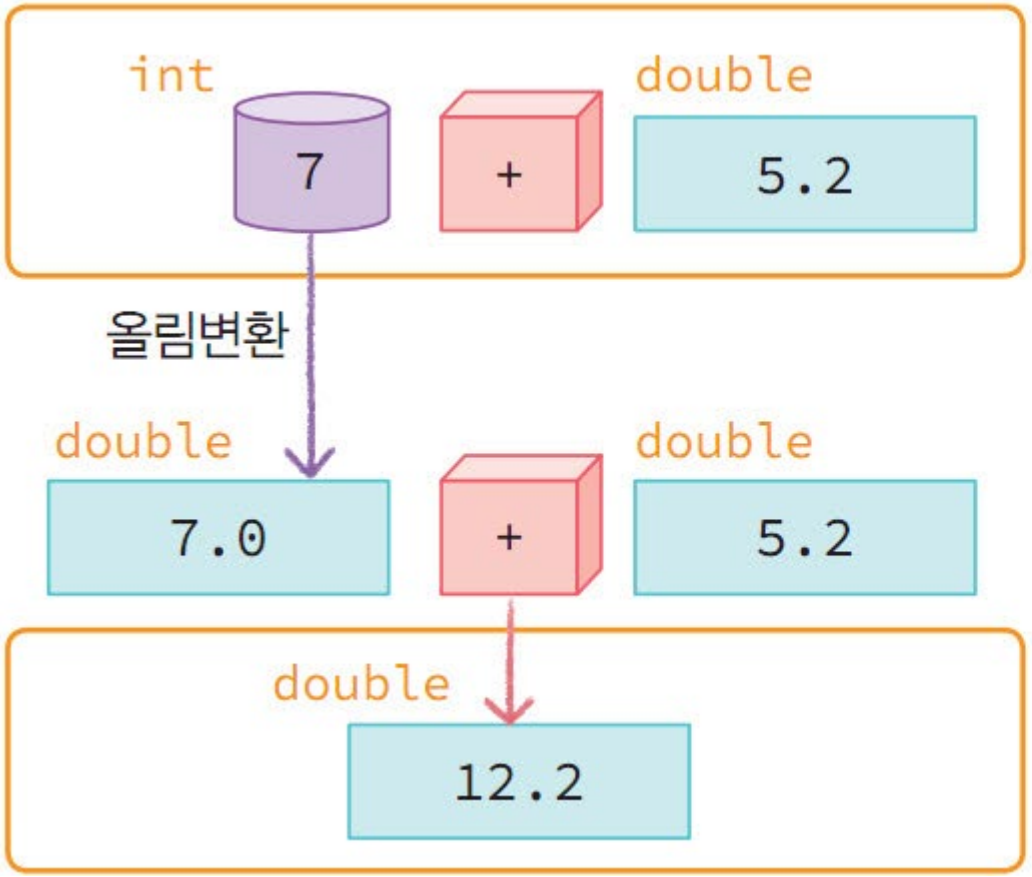
```
비트 연산이 가능한 두 정수를 입력하세요
40 3
 40 &   3 ==>   0
 40 |   3 ==>  43
 40 ^   3 ==>  43
 ~ 40 ==> -41
 40 >>   3 ==>   5
 40 <<   3 ==> 320
```

```c
// twobitop.c:
#define _CRT_SECURE_NO_WARNINGS //scanf() 오류를 방지하기 위한 상수 정의

#include <stdio.h>

int main(void)
{
    int a = 0, b = 0;

    printf("비트 연산이 가능한 두 정수를 입력하세요\n");
    scanf("%d %d", &a, &b);

    printf("%3d & %3d ==> %3d\n", a, b, a & b);
    printf("%3d | %3d ==> %3d\n", a, b, a | b);
    printf("%3d ^ %3d ==> %3d\n", a, b, a ^ b);
    printf(" ~%3d ==> %3d\n", a, ~a);
    printf("%3d >> %3d ==> %3d\n", a, b, a >> b);
    printf("%3d << %3d ==> %3d\n", a, b, a << b);

    return 0;
}
```

내림변환과 올림변환

- 자료형 변환은 크기 자료형의 범주 변화에 따른 구분
- 올림변환
  - 작은 범주의 자료형int에서 보다 큰 범주인 형double으로의 형변환
  - 올림변환은 형 넓히기라고도 부름
  - 올림변환은 정보의 손실이 없으므로 컴파일러에 의해 자동으로 수행 가능
  - 컴파일러가 자동으로 수행하는 형변환: 묵시적 형변환implicit type conversion

피연산자의 자동 올림변환



문자 'a'의 아스키 코드값의 int로 변환

다양한 올림변환의 예

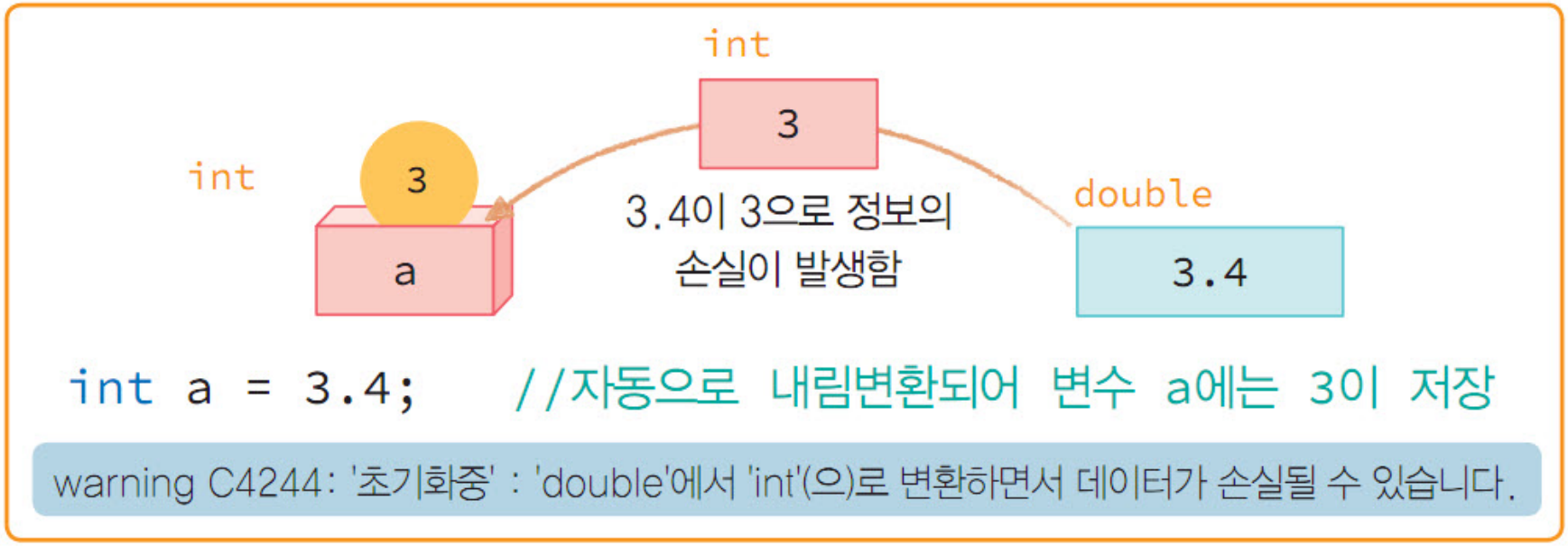
```
'a' + 2
3 * 4.1F
4.45F / 3.81
9.34 - 2
3 * 4.28
```

```
int + int
float * float
double / double
double - double
double * double
```

내림변환

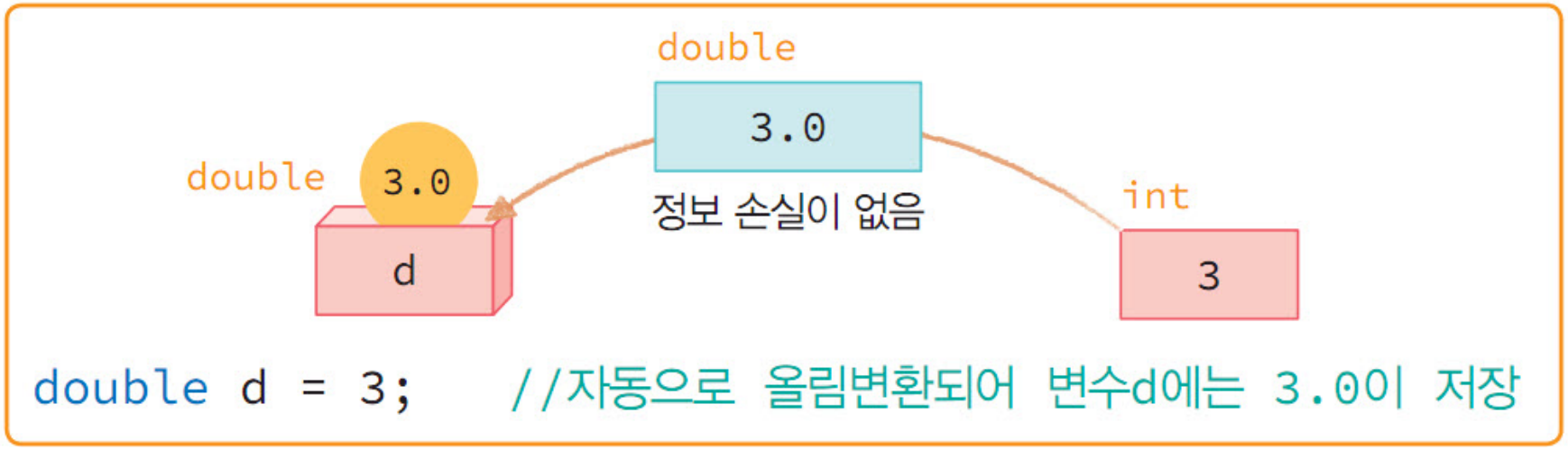
- 큰 범주의 자료형double에서 보다 작은 범주인 형int으로의 형변환
- 대입연산 int a = 3.4에서 내림변환이 필요
  - 컴파일러가 스스로 시행하는 묵시적 내림변환의 경우 정보의 손실이 일어날 수 있으므로 경고를 발생
  - 프로그래머의 명시적 형변환이 필요

대입연산에서의 내림변환과 올림변환




```
int a = 3.4;    //자동으로 내림변환되어 변수 a에는 3이 저장
```

warning C4244: '초기화중' : 'double'에서 'int'(으)로 변환하면서 데이터가 손실될 수 있습니다.




```
double d = 3;    //자동으로 올림변환되어 변수d에는 3.0이 저장
```

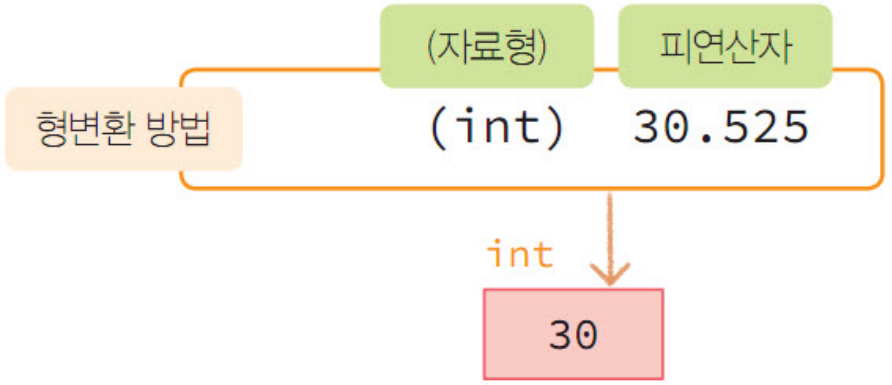
Source Code #09: typecast.c

```c
// file: typecast.c

#include <stdio.h>

int main(void)
{
    int a = 3.4;      //자동으로 내림변환되어 변수 a에는 3이 저장
    double d = 3;     //자동으로 올림변환되어 변수 d에는 3.0이 저장

    printf("%5d %10f ", a, d);
    printf("%10f\n", 3 + 4.5);

    printf("%5d ", 10 / 4);
    printf("%10f ", (double)10 / 4);
    printf("%10f ", 10 / (double)4);
    printf("%10f\n", (double)(10 / 4));

    printf("%5d ", (int)(3.4 + 7.8));
    printf("%10d ", (int) 3.4 + (int) 7.8);
    printf("%10f ", (int) 3.4 + 7.8);
    printf("%10f\n", 3.4 + (int) 7.8);

    return 0;
}
```

```
    3   3.000000   7.500000
    2   2.500000   2.500000   2.000000
   11         10  10.800000  10.400000
```

명시적 형변환explicit type conversion

- 형변환 연산자 '(type) 피연산자'는 뒤에 나오는 피연산자의 값을 괄호에서 지정한 자료형으로 변환하는 연산자
- 내림변환에서는 형변환 연산자type cast를 사용하여 내림변환을 직접 수행



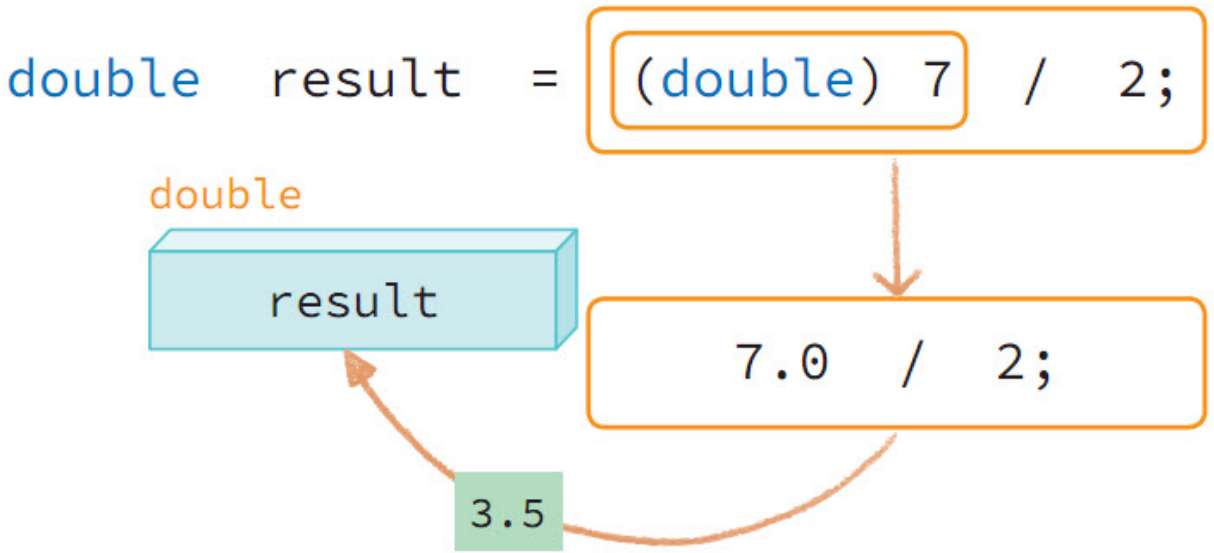
다양한 형변환연산 예

```
(int) 'A'        65
(int) 3.14       3
(double) 9       9.0
(double) 3.4F    3.4
(double) 7/2     3.5
```

형변환 연산자

- 올림변환
  - 상수나 변수의 정수값을 실수로 변환하려면 올림변환을 사용
  - 연산식 (double) 7의 결과는 7.0
- 내림변환
  - 실수의 소수부분을 없애고 정수로 사용하려면 내림변환을 사용
  - (int) 3.8의 결과는 3
  - 단항연산자인 형변환 연산자는 모든 이항연산자보다 먼저 계산

형변환연산자와 산술연산자의 사용



다양한 형변환 연산 예

(int) 3.8 + 5.7	8.7
3.8 + (int) 5.7	8.8
(int) 3.8 + (int) 5.7	8
(int) (3.8 + 5.7)	9
7 / 2	3
7.0 / 2	3.5
7 / (double) 2	3.5
(double) (7/2)	3.0

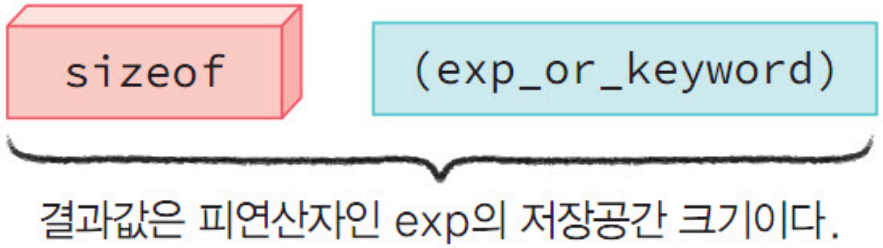
Source Code #10: comma.c

```c
// file: comma.c

#include <stdio.h>

int main(void)
{
    int a = 100, b = 50, c;

    printf("%d ", sizeof(short));
    printf("%d ", sizeof a);
    printf("%d ", sizeof 3.5F);
    printf("%d\n", sizeof 3.14);

    c = ++a, b++;
    printf("%d %d %d\n", a, b, c);
    c = (3 + a, b * 2);
    printf("%d %d %d\n", a, b, c);

    return 0;
}
```

```
2 4 4 8
101 51 101
101 51 102
```

연산자 sizeof

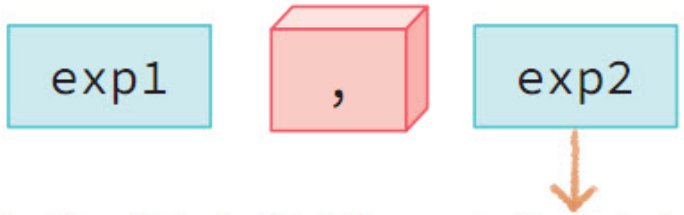
- 연산값 또는 자료형의 저장장소의 크기를 구하는 연산자
  - 결과값은 바이트 단위의 정수
  - 피연산자가 int와 같은 자료형인 경우 반드시 괄호를 사용
  - 피연산자가 상수나 변수 또는 연산식이면 괄호는 생략 가능




```
sizeof (int)      결과는 4
sizeof (3.14)     결과는 8
sizeof a          결과는 2 (short a;)
```

콤마연산자

- 콤마연산자 ,는 왼쪽과 오른쪽 연산식을 각각 순차적으로 계산
  - 결과값은 가장 오른쪽에서 수행한 연산의 결과
  - 연산식 2, 4의 결과값은 4
  - 또한 3+4, 2*5의 결과값은 10
  - 콤마연산자가 연속으로 나열된 식에서는 마지막에 수행된 가장 오른쪽 연산식의 결과가 전체 식의 결과값



결과값은 나중에 계산된 exp2의 결과값이다.

```
3 + 4, 5 - 10                    결과는 -5
3 + 4, 5 - 10, 2 * 3             결과는 6
```

복잡한 표현식의 계산

- 연산자 우선순위와 결합성
- 첫 번째 규칙은 괄호가 있으면 먼저 계산
  - 괄호 우선 규칙: '괄호가 있으면 먼저 계산한다'라는 규칙
- 두 번째 규칙으로 연산의 우선순위(priority)
  - 즉 '곱하기와 나누기는 더하기와 빼기보다 먼저 계산한다.'라는 규칙
- 세 번째 규칙은 동일한 우선순위인 경우
  - 연산을 결합하는 방법인 결합성(또는 결합규칙)
  - 연산자 결합성: '괄호가 없고 동일한 우선 순위라면, 덧셈과 뺄셈, 곱셈과 나눗셈과 같은 일반적인 연속된 연산은 왼쪽부터 오른쪽으로 차례로 계산
  - 다만 제곱승과 같은 정해진 연산은 오른쪽에서 왼쪽으로 차례로 계산한다'라는 규칙

다양한 연산자가 있는 연산 방식

```
3 + 8 / 2     * 4
```

다음 수식 연산 절차와 방법
1. 3 + ① .
2. ① = ② * 4, ② = 8 / 2 = 4
3. ① = 4 * 4 = 16
4. 3 + ① = 3 + 16 => 1

다음 수식 연산 절차와 방법
1. 8 / 2 * 4를 먼저 계산해야 하므로 3 + ① .
2. ①은 결합성에 따라 8/2를 먼저 계산하여 4
3. 다시 결합성에 따라 위 2의 결과인 4와 다음 4를 곱하면 결과는 16
4. 그러므로 최종 3 + 16이므로 결과는 19

```
3 + (   ( 8 / 2 ) * 4   )
```

연산자의 우선순위priority

- 우선순위가 1위
  - 함수호출과 괄호로 사용되는 ()
  - 후위 증감연산자 a++와 a-- 등의 단항연산자
  - 여러 개 있으면 결합성에 따라 왼쪽에서 오른쪽 순으로 계산
- 우선순위가 2위
  - 전위 증감연산자 ++a와 –a
  - 주소연산자 &
  - 오른쪽에서 왼쪽으로 차례로 계산
  - 모든 단항연산자는 우선순위가 1, 2위이며
- 우선순위가 16위
  - 콤마연산자

연산자의 우선순위priority

우선 순위	연산자	설명	분류	결합성(계산방향)
1	( ) [ ] . -> a++ a--	함수 호출 및 우선 지정 인덱스 필드(유니온) 멤버 지정 필드(유니온) 포인터 멤버 지정 후위 증가, 후위 감소	단항	-> (좌에서 우로)
2	++a --a ! ~ sizeof - + & *	전위 증가, 전위 감소 논리 NOT, 비트 NOT(보수) 변수, 자료형, 상수의 바이트 단위 크기 음수 부호, 양수 부호 주소 간접, 역참조		<- (우에서 좌로)
3	(형변환)	형변환		
4	* / %	곱하기 나누기 나머지	산술	-> (좌에서 우로)
5	+ -	더하기 빼기		-> (좌에서 우로)
6	<< >>	비트 이동	이동	-> (좌에서 우로)
7	< > <= >=	대소 비교	관계	-> (좌에서 우로)
8	== !=	동등 비교		-> (좌에서 우로)

연산자의 우선순위priority

우선 순위	연산자	설명	분류	결합성(계산방향)
9	&	비트 AND 또는 논리 AND	비트	-> (좌에서 우로)
10	^	비트 XOR 또는 논리 XOR		-> (좌에서 우로)
11	|	비트 OR 또는 논리 OR		-> (좌에서 우로)
12	&&	논리 AND(단락 계산)	논리	-> (좌에서 우로)
13	||	논리 OR(단락 계산)		-> (좌에서 우로)
14	? :	조건	조건	<- (우에서 좌로)
15	= += -= *= /= %= <<= >>= &= |= ^=	대입	대입	<- (우에서 좌로)
16	,	콤마	콤마	-> (좌에서 우로)

Source Code #11: priority.c

```c
// file: priority.c

#include <stdio.h>

int main(void)
{
    int a = 4, b = 6;
    double x = 3.3, y = 4.7;

    printf("%d ", a + b > y && x < y);  //산술 > 관계 > 논리
    printf("%d ", a++ - --b * 2);       //단항 > 곱셈 > 뺄셈
    printf("%f ", a > b ? x + 1 : y * 2);   //산술 > 관계 > 조건
    printf("%f ", x += 3 && y + 2); //산술 > 논리 > 대입
    printf("%f\n", (x = x + 1, y = y + 1));//괄호 > 산술 > 대입 > 콤마

    return 0;
}
```

```
1 -6 9.400000 4.300000 5.700000
```

우선순위 요약

이항연산자

콤마 < 대입 < 조건(삼항) < 논리 < 관계 < 산술 < 단항 < 괄호와 대괄호

- 괄호와 대괄호는 무엇보다도 가장 먼저 계산한다.
- 모든 단항연산자는 어느 이항연산자보다 먼저 계산한다.
- 산술연산자 *, /, %는 +, -보다 먼저 계산한다.
- 산술연산자는 이항연산자 중에서 가장 먼저 계산한다.
- 관계연산자는 논리연산자보다 먼저 계산한다.
- 조건 삼항연산자는 대입연산자보다 먼저 계산하나, 다른 대부분의 연산보다는 늦게 계산한다.
- 조건 > 대입> 콤마연산자 순으로 나중에 계산한다.

연산 우선순위 예

변수값	표현식	설명	해석	결과
x = 3 y = 3	x >> 1 + 1 > 1 & y	산술 > 이동 > 관계 > 비트	((x >> (1 + 1)) > 1) & y	0
	x - 3 || y & 2	산술 > 비트 > 논리	(x - 3) || (y & 2)	1
	x & y && y >= 4	관계 > 비트 > 논리	(x & y) && (y >= 4)	0
	x && x | y++	증가 > 비트 > 논리	x && (x | (y++))	1

Source Code #12: association.c

```c
// file: association.c

#include <stdio.h>

int main(void)
{
    int m = 5, n = 10;

    //우측에서 좌측으로 결합
    printf("%d ", n += m /= 3);
    m = 5; n = 10;
    printf("%d\n", (n += (m /= 3)));

    printf("%d ", 10 * 3 / 2);  //좌측에서 우측으로 결합
    printf("%d\n", 10 * (3 / 2)); //우측에서 좌측으로 결합

    //우측에서 좌측으로 결합
    printf("%d ", 3 > 4 ? 3 - 4 : 3 > 4 ? 3 + 4 : 3 * 4);
    printf("%d\n", 3 > 4 ? 3 - 4 : (3 > 4 ? 3 + 4 : 3 * 4));

    return 0;
}
```

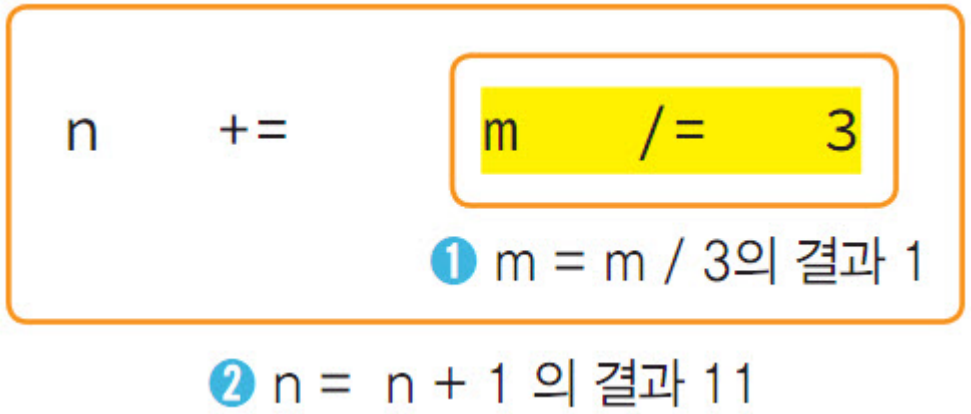
```
11 11
15 10
12 12
```

결합법칙

- 대부분 좌에서 우로 수행
- 우에서 좌(d)로 수행
  - 우선 순위가 2위인 전위의 단항 연산자, 우선 순위 14위인 조건연산자 그리고 우선순위 15위인 대입연산자
- 산술연산식 10 * 3 / 2는 ((10 * 3) / 2)
  - 결과는 15
- 축약 대입연산자로 구성
  - 연산식 n += m /= 3
  - 우에서 좌(d)로 먼저 결합하여 식 (n += (m /= 3))을 수행

우에서 좌로 먼저 결합하는 축약 대입연산자

```
int n = 10, m = 5;
n += m /= 3;
```



다양한 연산식

- 수학이나 공학에서의 다양한 수식을 표현
  - 연산의 우선순위를 고려하여 괄호의 사용이 필요
  - 제곱근을 구하는 함수 sqrt()를 활용

다양한 연산식 표현

수학식	C 연산식
$(a+b)(x+y)^2$	`(a + b) * (x + y) * (x + y)`
$4x^3+3x^2-5x+10$	`4*x*x*x + 3*x*x - 5*x + 10`
$\frac{a+b}{a-b}$	`(a + b) / (a - b)`
$\frac{5}{9}(F-32)$	`(5.0 / 9) * (F - 32)`
$\frac{9}{5}C+32$	`9.0 / 5 * C + 32`
$\frac{1-x}{x^2}$	`(1 - x) / (x * x)`
$\frac{-b+\sqrt{b^2-4ac}}{2a}$	`(-b + sqrt(b*b - 4*a*c)) / 2*a` sqrt(x)는 x의 제곱근을 구하는 함수

Lab #03: 온도celsius를 화씨fahrenheit 온도로 출력

- 표준입력으로 받은 섭씨celsius 온도를 화씨fahrenheit 온도로 출력하는 프로그램
  - 다음과 같은 입력과 출력
  - 섭씨(C) 온도를 화씨 온도(F)로 변환하는 식: F = 9/5*C + 32

```c
// celtofar.c
#define _CRT_SECURE_NO_WARNINGS //scanf() 오류를 방지하기 위한 상수 정의

#include <stdio.h>

int main(void)
{
    double fahrenheit, celsius;
    printf("변환할 섭씨온도를 입력하세요. >> ");
    scanf("%lf", &celsius);

    fahrenheit = 

    printf("\n입력된 %.2f도는 화씨온도로 %.2f도 입니다.\n", );

    return 0;
}
```

```
변환할 섭씨온도를 입력하세요. >> 34.765879

입력된 34.77도는 화씨온도로 94.58도 입니다.
```


I
PROGRAMMING