Kumoh National Institute of Technology

C Programming

Ch.2-1 데이터 표현방식의 이해

컴퓨터공학과

C프로그래밍

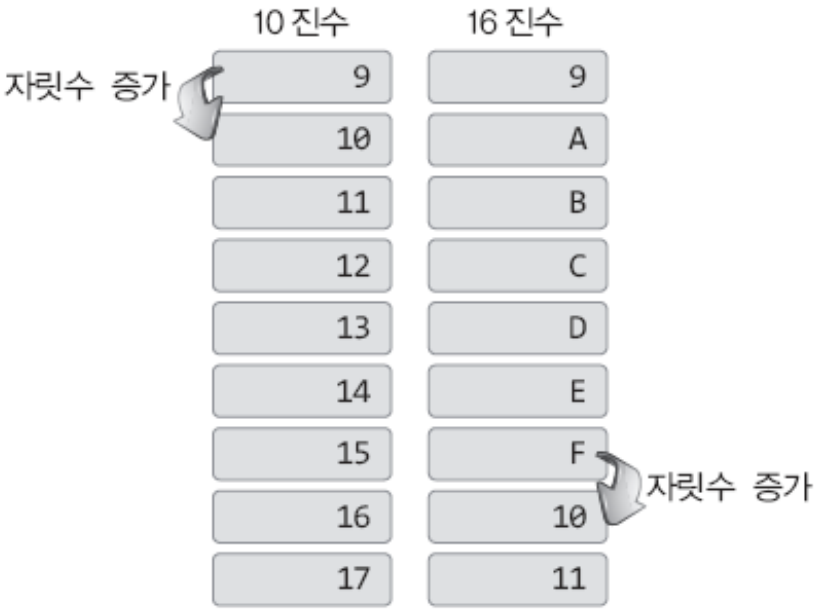
Contents

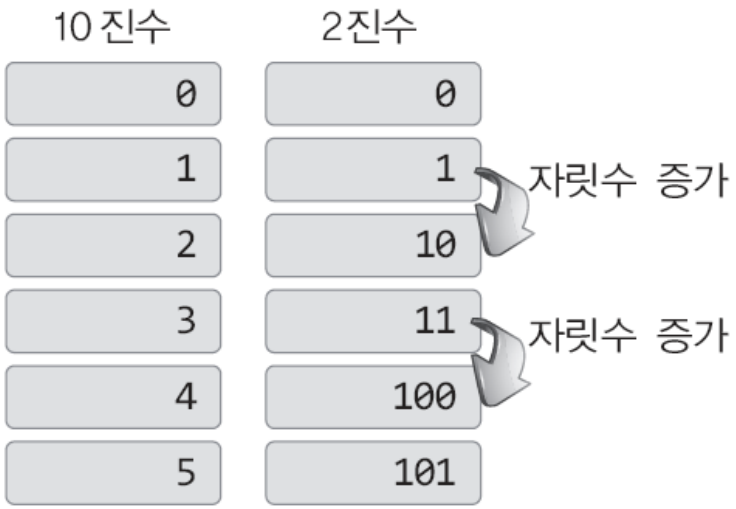
1. 컴퓨터가 데이터를 표현하는 방식
2. 2진수와 10진수 사이의 변환 (추가)
3. 정수와 실수의 표현 방식
4. 비트 연산자

- 추가 : 교재 외 추가 자료

2진수, 10진수, 16진수, 8진수

- 값의 표현 방식
  - 2진수 : 2개의 기호 사용 (0, 1)
  - 10진수 : 10개의 기호 사용 (0, 1, 2, ..., 9)
  - 16진수 : 16개 사용 (0, 1, 2, ..., 9, A, B, C, D, E, F)
  - 8진수 : 8개 사용 (0, 1, 2, ..., 7)



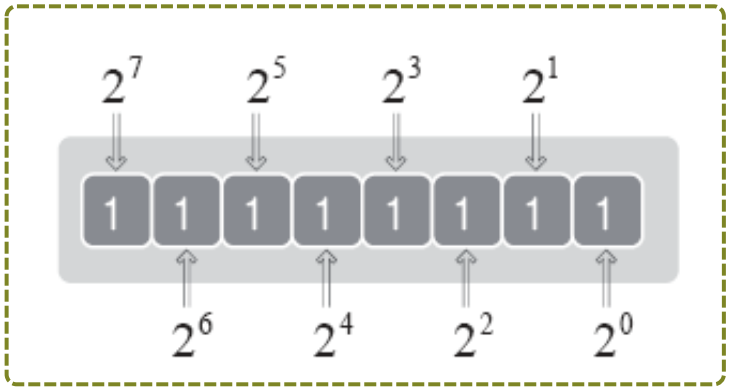


데이터 표현 단위인 비트(Bit)와 바이트(Byte)

- 모든 데이터는 2진수 개념으로 저장됨
  - 1비트 : 0 또는 1 저장, 1바이트 = 8비트

메모리의 주소 1바이트 단위로 할당됨. 데이터는 주로 바이트 단위로 처리됨

- 2진수의 자리값
  - 2진수 → 10진수 변환
    - (자리값 * 현재값)의 합
    - 00001010 → 10



(참고) 2진수를 8진수 또는 16진수로 변환

- 16진수 : 2진수를 4개씩 끊어서 변환
- 8진수 : 2진수를 3개씩 끊어서 변환
- 예
  - 011010101.....
  - 16진수 : 0110(6) 1010(A) → 6A...
  - 8진수 : 011(3) 010(2) 101(5) → 325...
- 다음 2진수를 16진수와 8진수로 나타내면
  - 011101010000000110100101
  - 16진수 :
  - 8진수 :

8진수와 16진수 데이터의 표현

```
int main(void)
{
    int num1 = 0xA7, num2 = 0x43;
    int num3 = 032, num4 = 024;

    printf("0xA7의 10진수 정수 값: %d \n", num1);
    printf("0x43의 10진수 정수 값: %d \n", num2);
    printf(" 032의 10진수 정수 값: %d \n", num3);
    printf(" 024의 10진수 정수 값: %d \n", num4);

    printf("%d-%d=%d \n", num1, num2, num1 - num2);
    printf("%d+%d=%d \n", num3, num4, num3 + num4);
}
```

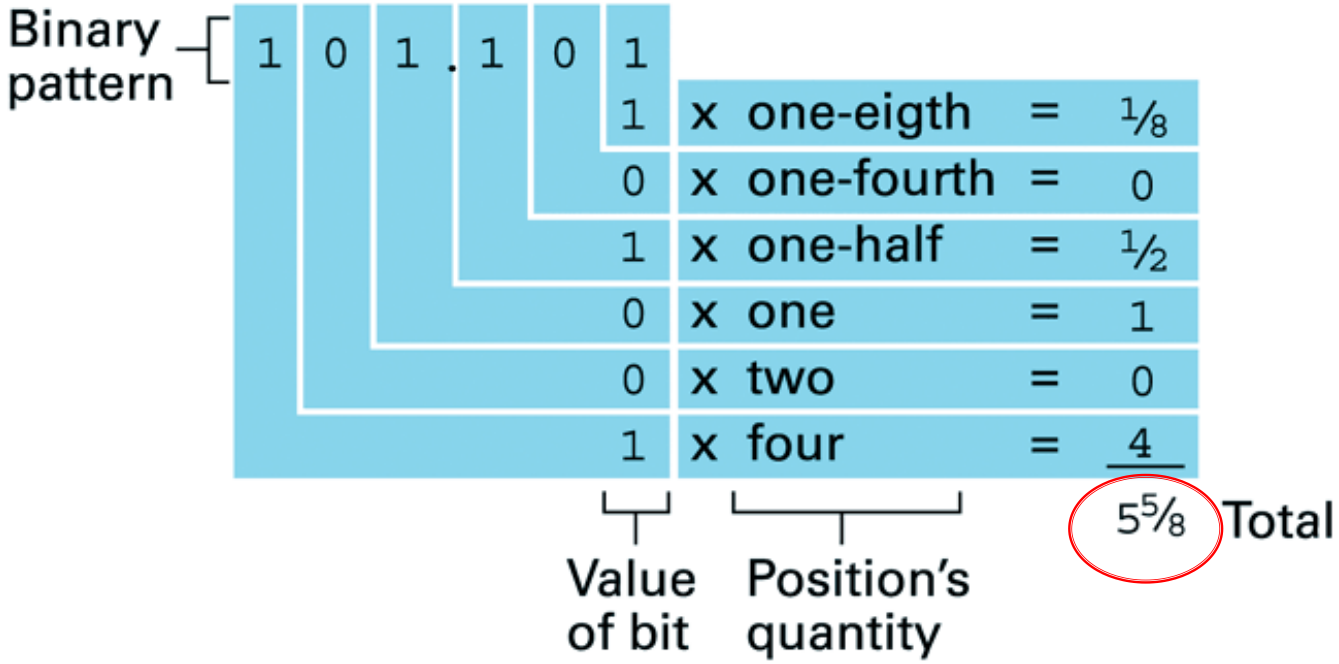
0x로 시작 : 16진수 (0x47)
0으로 시작 : 8진수 (046)

printf 함수
16진수로 출력 : %x
8진수로 출력 : %o
→ 양의 정수만 표현
→ 음수 표현에 잘 사용되지 않음

```
0xA7의 10진수 정수 값: 167
0x43의 10진수 정수 값: 67
 032의 10진수 정수 값: 26
 024의 10진수 정수 값: 20
167-67=100
26+20=46
```

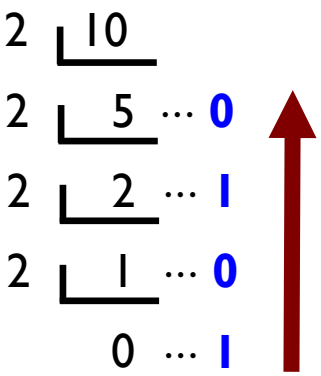
2진수 → 10진수

- 2진수 101.101 → 10진수 ?

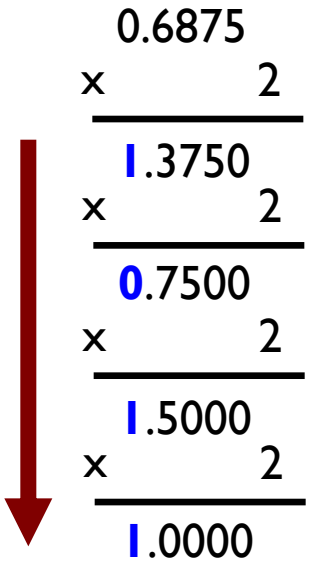


10진수 → 2진수

- 10진수 10.6875 → 2진수 ?



1010_2



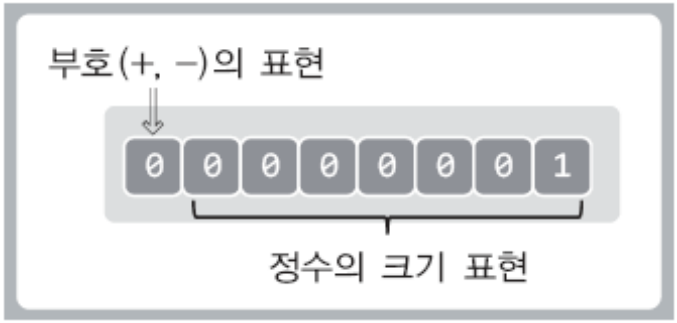
0.1011_2

1010.1011_2

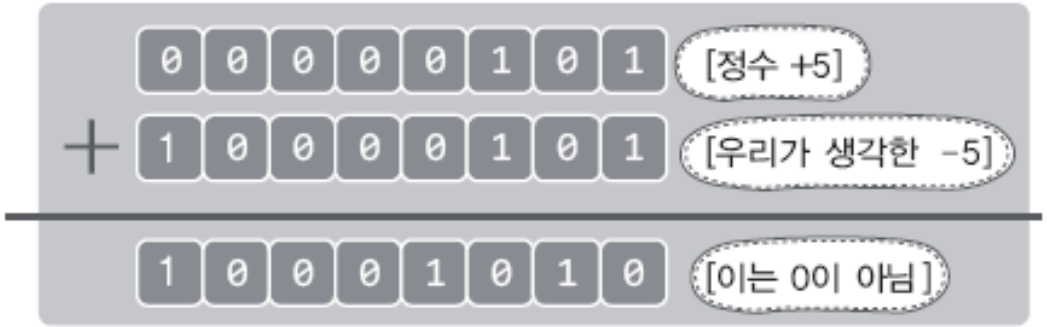
정수의 표현 방식 (1)

- 다음과 같이 일반적인 방식으로 표현한다면? (8비트 가정)
  - 가장 왼쪽 비트로 부호 표현
  - 나머지는 크기

MSB(Most Significant Bit)
– 가장 왼쪽 비트
LSB(Least Significant Bit)
– 가장 오른쪽 비트

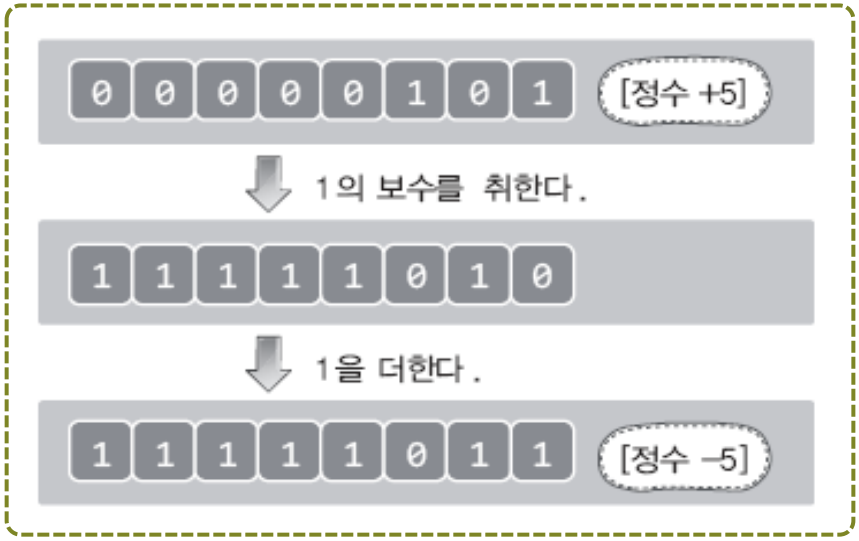


- 10000000, 00000000 모두 0을 의미
- (+5) + (–5) = ?
  - 2진수 덧셈?

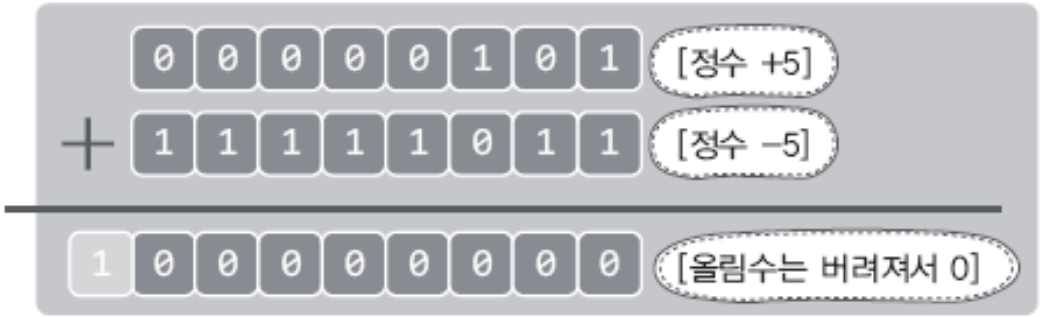


정수의 표현 방식 (2)

- 2의 보수 표기법
  - 음의 정수 표현 방법
  - 최상위 비트는 여전히 부호를 나타냄



- 2의 보수 표기법에서의 덧셈
  - (+5) + (–5) = 0!



2의 보수 표기법의 특징 (참고)

- 절대값이 동일한 양수와 음수 간에는 일정한 관계가 성립
  - 이 두 값을 오른쪽에서 왼쪽으로 읽어나갈 때 첫 번째 1을 만나는 위치까지는 서로 동일하게 코딩되어 있음
  - 그 다음 위치부터는 서로 보수 형태의 패턴으로 구성

2의 보수 표기법의 특징 (참고)

- 3비트와 4비트로 나타낼 수 있는 값의 범위

a. Using patterns of length three

Bit pattern	Value represented
011	3
010	2
001	1
000	0
111	−1
110	−2
101	−3
100	−4

b. Using patterns of length four

Bit pattern	Value represented
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	−1
1110	−2
1101	−3
1100	−4
1011	−5
1010	−6
1001	−7
1000	−8

- 8비트로 나타낼 수 있는 값의 범위는?

정수 연산 시 오버플로우 문제 (참고)

- 오버플로우
  - 표현할 수 있는 값의 범위를 벗어나는 값을 표현하고자 할 때 발생
  - char 변수(8비트)의 값 127 + 1 = 128?

```
int main(void)
{
    char num = 127;
    num = num + 1;

    printf("%d\n", num);
}
```

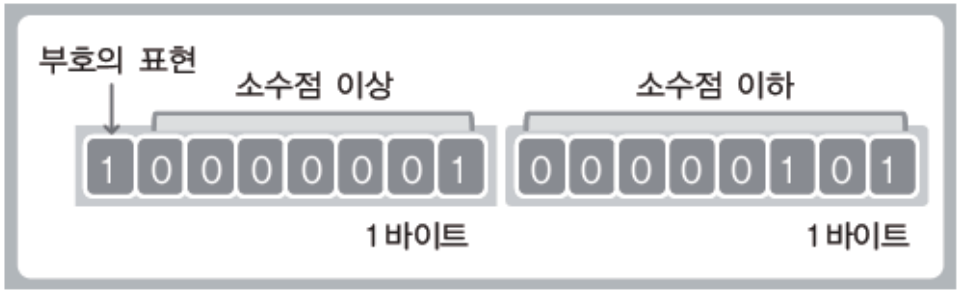
-128

최대값 + 1 = 최소값
최소값 - 1 = 최대값

  - 부호 비트를 조사하여 오버플로우의 발생 여부 확인 가능
    - 양수끼리 더했을 때 부호가 음수가 되는 경우
    - 음수끼리 더했을 때 부호가 양수가 되는 경우

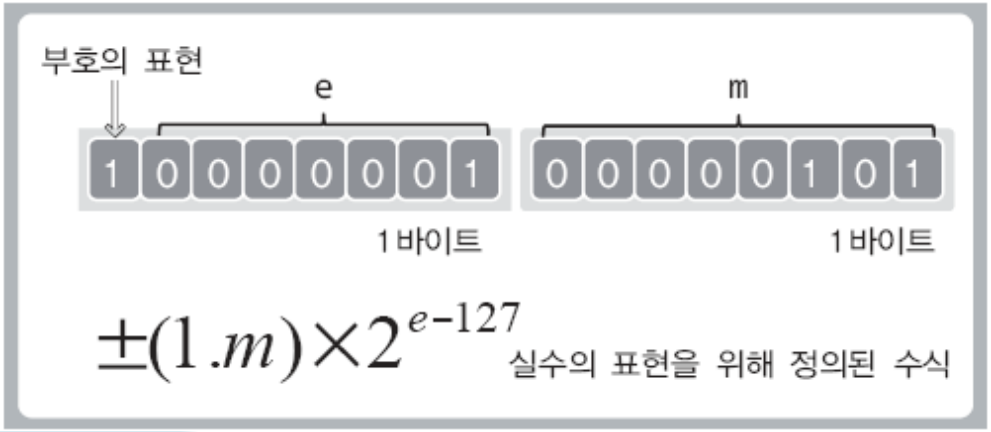
실수의 표현 방식 (1)

- -1.5를 다음과 같이 표현한다면?
  - 큰 숫자의 표현 방법? 100000000?



- 실수의 표현 방식 : 부동 소수점 방식
  - 부호 비트, 지수, 가수로 나누어 표현
  - -2.5 → -10.1_2
    → $-1.01 * 2^{+1}$

1	+1	010000...



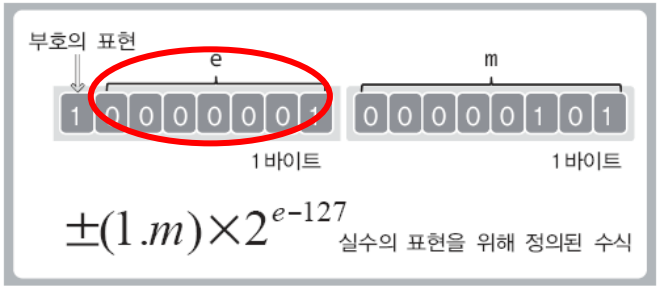
실수의 표현 방식 (2)

- 실제 float 타입과 double 타입의 실수 표현 방식
  - float : 32bit
  - double : 64bit

실수의 표현 방식 (3)

- 지수 : 정수값 – 127 초과 표기법 사용 (8비트의 경우)
  - 원래 10진수값에서 127을 뺀 수

비트열	10진수	실제값
11111111	255	128
.....	.....	.....
10000010	130	3
10000001	129	2
10000000	128	1
01111111	127	0
01111110	126	-1
01111101	125	-2
01111100	124	-3
01111011	123	-4
.....	.....	.....
00000000	0	-127



실수의 표현 방식 (4)

- float 변수(총 32비트)의 2진 표현 예
  - 부호(1비트), 지수(8비트), 가수(23비트)

```
#include <stdio.h>
#include <math.h>    // INFINITY, NAN

void PrintBits(float f_value)  // 신경쓸 필요 없음
{
    int *i_ptr = (int *)&f_value;
    int mask = 1;

    printf("%20f : ", f_value);

    for (int shift_cnt = 31; shift_cnt >= 0; shift_cnt--)
    {
        int tmp = (*i_ptr) >> shift_cnt;
        tmp = tmp & mask;
        printf("%d", tmp);
    }

    printf("\n");
}

int main(void)
{
    PrintBits(1.0f);        // => 1.0 * 2^0
    PrintBits(10.6875f);    // 1010.1011 => 1.0101011 * 2^3
    PrintBits(-10.6875f);
    PrintBits(0.1f);        // 0.0001100110011... => 1.1001100... * 2^(-4)
    PrintBits(0.0f);        // 특수값
    float a = INFINITY; float b = NAN;
    PrintBits(a); PrintBits(b);
}
```

```
   1.000000 : 00111111100000000000000000000000
  10.687500 : 01000001010101100000000000000000
 -10.687500 : 11000001010101100000000000000000
   0.100000 : 00111101110011001100110011001101
   0.000000 : 00000000000000000000000000000000
   1.#INF00 : 01111111100000000000000000000000
  -1.#IND00 : 11111111110000000000000000000000
```

특수값
- 0.0 : 지수 모두 0, 가수 모두 0
- 무한대 : 지수 모두 1, 가수 모두 0
- 잘못된 값(NaN) : 지수 모두 1, not 모두 0

실수 표현 시 오차

- 0.1조차도 정확하게 표현할 수 없었다!
  - 어쩔 수 없다. 실수 연산 시 오차로 인한 이상 동작에 유의하자!

```
void main(void)
{
    int i;
    float num = 0.0;

    for (i = 0; i < 100; i++)
        num += 0.1;

    printf("0.1을 100번 더한 결과: %f \n", num);
}
```

```
0.1을 100번 더한 결과: 10.000002
```

비트 연산자

- 비트 단위로의 연산 수행

연산자	연산자의 기능	결합방향
&	비트단위로 AND 연산을 한다. 예) num1 & num2;	→
|	비트단위로 OR 연산을 한다. 예) num1 | num2;	→
^	비트단위로 XOR 연산을 한다. 예) num1 ^ num2;	→
~	단항 연산자로서 피연산자의 모든 비트를 반전시킨다. 예) ~num; // num은 변화 없음, 반전 결과만 반환	←
<<	피연산자의 비트 열을 왼쪽으로 이동시킨다. 예) num<<2; // num은 변화 없음, 두 칸 왼쪽 이동 결과만 반환	→
>>	피연산자의 비트 열을 오른쪽으로 이동시킨다. 예) num>>2; // num은 변화 없음, 두 칸 오른쪽 이동 결과만 반환	→

& 연산자 : 비트 단위 AND

```
int main(void)
{
    int num1 = 15;  // 00000000 00000000 00000000 00001111
    int num2 = 20;  // 00000000 00000000 00000000 00010100
    int num3 = num1 & num2;
    printf("AND 연산의 결과: %d \n", num3);
}
```

```
AND 연산의 결과: 4
```

```
        00000000 00000000 00000000 00001111
& 연산  00000000 00000000 00000000 00010100
-------------------------------------------
        00000000 00000000 00000000 00000100
```

- 0 & 0 → 0을 반환
- 0 & 1 → 0을 반환
- 1 & 0 → 0을 반환
- 1 & 1 → 1을 반환

| 연산자 : 비트 단위 OR

```
int main(void)
{
    int num1 = 15;  // 00000000 00000000 00000000 00001111
    int num2 = 20;  // 00000000 00000000 00000000 00010100
    int num3 = num1 | num2;
    printf("OR 연산의 결과: %d \n", num3);
}
```

```
OR 연산의 결과: 31
```

```
     00000000 00000000 00000000 00001111
| 연산 00000000 00000000 00000000 00010100
-------------------------------------------
     00000000 00000000 00000000 00011111
```

- 0 & 0 0을 반환
- 0 & 1 1을 반환
- 1 & 0 1을 반환
- 1 & 1 1을 반환

^ 연산자 : 비트 단위 XOR

```
int main(void)
{
    int num1 = 15;  // 00000000 00000000 00000000 00001111
    int num2 = 20;  // 00000000 00000000 00000000 00010100
    int num3 = num1 ^ num2;
    printf("XOR 연산의 결과: %d \n", num3);
}
```

```
XOR 연산의 결과: 27
```

```
      00000000 00000000 00000000 00001111
^ 연산 00000000 00000000 00000000 00010100
-------------------------------------------
      00000000 00000000 00000000 00011011
```

- 0 & 0 — 0을 반환
- 0 & 1 — 1을 반환
- 1 & 0 — 1을 반환
- 1 & 1 — 0을 반환

~ 연산자 : 비트 단위 NOT

```
int main(void)
{
    int num1 = 15;  // 00000000 00000000 00000000 00001111
    int num2 = ~num1;
    printf("NOT 연산의 결과: %d \n", num2);
}
```

```
NOT 연산의 결과: -16
```

~ 연산 전 00000000 00000000 00000000 00001111

~ 연산 후 11111111 11111111 11111111 11110000

- ~ 0 1을 반환
- ~ 1 0을 반환

<< 연산자 : 비트 단위 왼쪽 이동(Shift)

- num1 << num2
  - num1의 비트들을 num2칸만큼 왼쪽으로 자리 이동
  - 빈 자리는 0으로 채워짐

```
1칸 이동 결과: 30
2칸 이동 결과: 60
3칸 이동 결과: 120
```

```
int main(void)
{
    int num = 15;      // 00000000 00000000 00000000 00001111
    int result1 = num << 1; // num의 비트 열을 왼쪽으로 1칸씩 이동
    int result2 = num << 2; // num의 비트 열을 왼쪽으로 2칸씩 이동
    int result3 = num << 3; // num의 비트 열을 왼쪽으로 3칸씩 이동
    printf("1칸 이동 결과: %d \n", result1);
    printf("2칸 이동 결과: %d \n", result2);
    printf("3칸 이동 결과: %d \n", result3);
}
```

```
00000000 00000000 00000000 00011110    // 10진수로 30
00000000 00000000 00000000 00111100    // 10진수로 60
00000000 00000000 00000000 01111000    // 10진수로 120
```

왼쪽으로 한 칸씩 이동할 때마다 정수의 값은 두 배씩 증가
반대로 오른쪽으로 한 칸씩 이동할 때마다 정수의 값은 반으로 감소

>> 연산자 : 비트 단위 오른쪽 이동

- 8 >> 2 : 정수 8의 비트들을 오른쪽으로 2회 산술적 자리 이동 또는 논리적 자리 이동 (CPU마다 다를 수 있음)

```
11111111 11111111 11111111 11110000      // -16
```

(-16) >> 2 : CPU에 따라서 달라지는 연산의 결과

```
00111111 11111111 11111111 11111100      // 0이 채워진 경우

11111111 11111111 11111111 11111100      // 1이 채워진 경우
```

논리적 이동

산술적 이동
(부호 동일)

```
int main(void)
{
    int num = -16;      // 11111111 11111111 11111111 11110000
    printf("2칸 오른쪽 이동의 결과: %d \n", num >> 2);
    printf("3칸 오른쪽 이동의 결과: %d \n", num >> 3);
}
```

```
2칸 오른쪽 이동의 결과: -4
3칸 오른쪽 이동의 결과: -2
```

이번 장에서 배운 것

- 컴퓨터에 저장되는 값들은 2진수 형태로 저장되고, 보통 프로그램을 통해 10진수, 16진수, 8진수 형태로 다룬다.
- 2진수와 10진수 사이에는 쉽게 변환이 가능하다.
- 정수는 2의 보수 표기법으로 저장된다.
- 실수는 부동 소수점 방식으로 저장된다. 즉, 부호, 지수, 가수로 나뉘어 저장된다.
- 비트 단위 연산자로는 AND, OR, NOT, XOR, Left Shift, Right Shift 연산자가 있다.