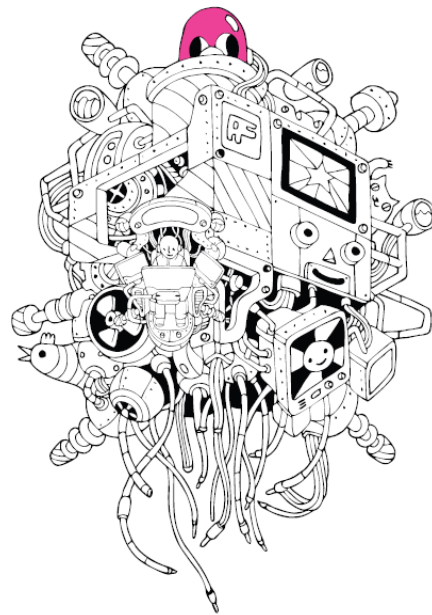


# 윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 04. 데이터 표현방식의 이해

# 윤성우의 열혈 C 프로그래밍



Chapter 04-1. 컴퓨터가 데이터를 표현  
하는 방식

윤성우 저 열혈강의 C 프로그래밍 개정판

# 2진수란 무엇인가?

## 더불어 10진수, 16진수란 무엇인가?



### 2진수

- 두 개의 기호를 이용해서 값(데이터)를 표현하는 방식

### 10진수

- 열 개의 기호를 이용해서 값(데이터)을 표현하는 방식

### N진수

- N개의 기호를 이용해서 값(데이터)을 표현하는 방식

	10 진수	16 진수
자릿수 증가	9	9
	10	A
	11	B
	12	C
	13	D
	14	E
	15	F
	16	10
	17	11

자릿수 증가

10 진수	2진수
0	0
1	1
2	10
3	11
4	100
5	101

자릿수 증가

자릿수 증가

# 데이터의 표현단위인 비트(Bit)와 바이트(Byte)

1비트



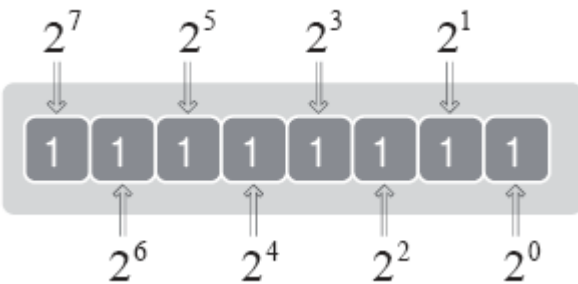
1바이트



2바이트



컴퓨터 메모리의 주소 값은 **1바이트당 하나의 주소가 할당**되어 있다. 따라서 바이트는 컴퓨터에 있어서 상당히 의미가 있는 단위이다.



왼쪽의 의미 있는 정보를 이용하면 2진수를 쉽게 10진수로 변환할 수 있다.

# 8진수와 16진수를 이용한 데이터 표현

```
int num1 = 10;    // 특별한 선언이 없으면 10진수의 표현
int num2 = 0xA;   // 0x로 시작하면 16진수로 인식
int num3 = 012;   // 0으로 시작하면 8진수로 인식
```

```
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);
    return 0;
}
```

## 실행결과

```
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
```

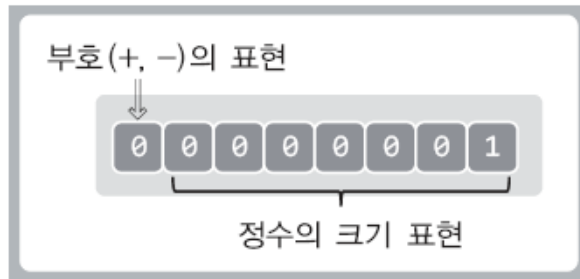
# 윤성우의 열혈 C 프로그래밍



## Chapter 04-2. 정수와 실수의 표현방식

윤성우 저 열혈강의 C 프로그래밍 개정판

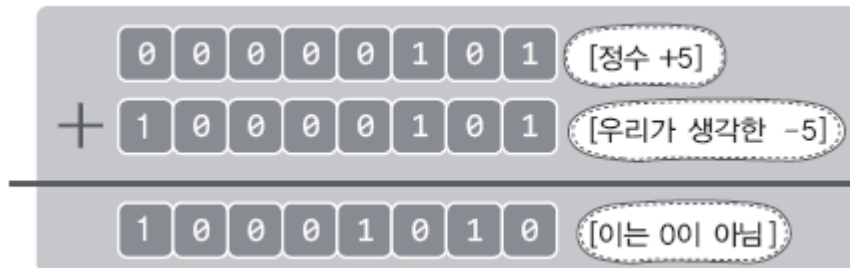
# 정수의 표현방식



- 가장 왼쪽 비트를 MSB(Most Significant Bit)라 한다.
- MSB는 부호를 나타내는 비트이다.
- MSB를 제외한 나머지 비트는 크기를 나타내는데 사용된다.

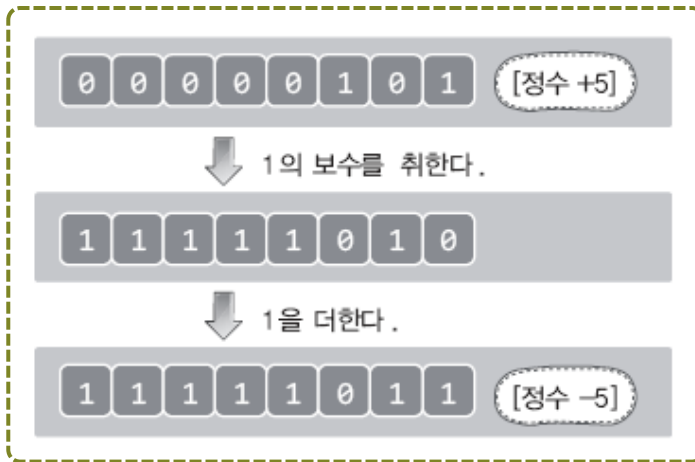
정수의 표현방식은 바이트의 크기와는 상관 없다.

바이트의 크기가 크면 그만큼 넓은 범위의 정수를 표현할 수 있을 뿐이다.



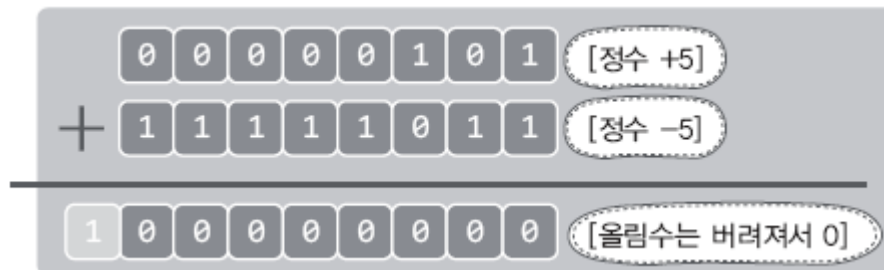
왼쪽에서는 양의 정수를 표현하는 방식으로 음의 정수를 표현하는 것이 적절치 않은 이유를 설명한다.

# 음의 정수 표현방식



음의 정수를 표현하는 방식

2의 보수를 취하여 음의 정수를 표현한다.



2의 보수 표현법이

음수를 표현하는데에 있어서 타당한지를 확인



# 실수의 표현방식

다음의 방식과 같이 정수를 표현하는 방식과 유사하게 실수를 표현하면 다음의 문제가 따른다.

- 표현할 수 있는 실수의 수가 몇 개 되지 않는다.
- 3.12456과 3.12457 사이에 있는 무수히 많은 실수조차 제대로 표현하지 못한다.

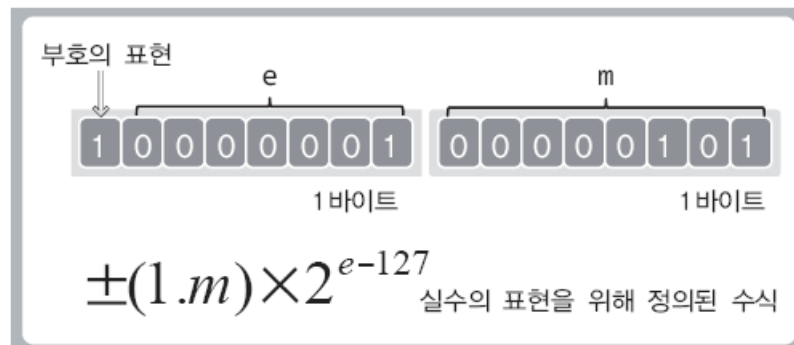
적절하지 못한 실수의  
표현방법



따라서 실수의 표현방식은 정수의 표현방식과 다르다.

- 실수의 표현방식에서는 정밀도를 포기하는 대신에 표현할 수 있는 값의 범위를 넓힌다.
- 따라서 컴퓨터는 완벽하게 정밀한 실수를 표현하지 못한다.

오차가 존재하지만  
적절한 실수의 표현방법



# 실수 표현의 오차 확인하기

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

    for(i=0; i<100; i++)
        num+=0.1;    // 이 연산을 총 100회 진행하게 됩니다.

    printf("0.1을 100번 더한 결과: %f \n", num);
    return 0;
}
```

실행결과

0.1을 100번 더한 결과: 10.000002

이론적으로 오차 없이 모든 실수를 완벽하게 표현할 능력이 있는 컴퓨팅 환경은 존재하지 않는다.  
즉, 실수 표현에 있어서의 오차 발생은 C언어의 특성이 아닌 컴퓨터의 기본적인 특성이다.



# 윤성우의 열혈 C 프로그래밍



## Chapter 04-3. 비트 연산자

윤성우 저 열혈강의 C 프로그래밍 개정판

## 비트 연산자(비트 이동 연산자)

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: 비트단위 AND

```
int main(void)
{
    int num1 = 15;    // 00000000 00000000 00000000 00001111
    int num2 = 20;    // 00000000 00000000 00000000 00010100
    int num3 = num1 & num2;    // num1과 num2의 비트단위 & 연산
    printf("AND 연산의 결과: %d \n", num3);
    return 0;
}
```

• 0 & 0	0을 반환
• 0 & 1	0을 반환
• 1 & 0	0을 반환
• 1 & 1	1을 반환

AND 연산의 결과: 4

실행결과

	00000000	00000000	00000000	000	01111
& 연산	00000000	00000000	00000000	000	10100
<hr/>					
	00000000	00000000	00000000	000	00100

# | 연산자: 비트단위 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);
    return 0;
}
```

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

OR 연산의 결과: 31

실행결과

	00000000	00000000	00000000	000	01111
연산	00000000	00000000	00000000	000	10100
	00000000	00000000	00000000	000	11111

## ^ 연산자: 비트단위 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);
    return 0;
}
```

• 0 & 0	0을 반환
• 0 & 1	1을 반환
• 1 & 0	1을 반환
• 1 & 1	0을 반환

XOR 연산의 결과: 27

실행결과

	00000000	00000000	00000000	000	01111
^ 연산	00000000	00000000	00000000	000	10100
	00000000	00000000	00000000	000	11011

## ~ 연산자

```
int main(void)
{
    int num1 = 15;    // 00000000 00000000 00000000 00001111
    int num2 = ~num1;
    printf("NOT 연산의 결과: %d \n", num2);
    return 0;
}
```

- ~ 0      1을 반환
- ~ 1      0을 반환

NOT 연산의 결과: -16

실행결과

~ 연산 전    00000000 00000000 00000000 00001111  
~ 연산 후    11111111 11111111 11111111 11110000



## << 연산자: 비트의 왼쪽 이동(Shift)

- `num1 << num2`    `num1`의 비트 열을 `num2`칸씩 왼쪽으로 이동시킨 결과를 반환
- `8 << 2`            정수 8의 비트 열을 2칸씩 왼쪽으로 이동시킨 결과를 반환

```
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);
    return 0;
}
```

실행결과

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

```
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 증가한다.**

반대로 오른쪽으로 한 칸씩 이동할 때마다 정수의 값은 **반으로 줄어든다.**

## >> 연산자: 비트의 오른쪽 이동(Shift)

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



CPU에 따라서 달라지는 연산의 결과

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

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

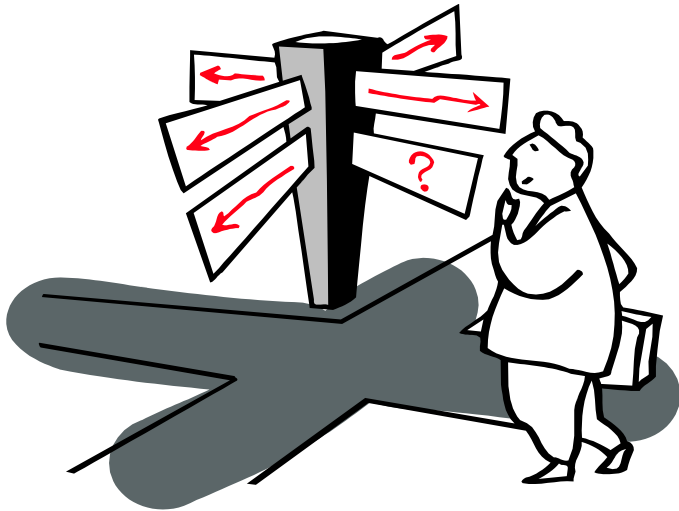
왼쪽 비트가 0으로 채워진 음수의 경우 오른쪽으로 비트를 이동시킨 결과는 CPU에 따라서 달라진다.  
따라서 호환성이 요구되는 경우에는 >> 연산자의 사용을 제한해야 한다.

```
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);
    return 0;
}
```

실행결과

2칸 오른쪽 이동의 결과: -4

3칸 오른쪽 이동의 결과: -2



Chapter 이가 끝났습니다. 질문 있으신지요?