C 프로그래밍 1
Lecture Note #04

백윤철
ybaek@smu.ac.kr

윤성우의

열혈 C 프로그래밍

Chapter 04-1. 컴퓨터가 데이터를 표현하는 방식

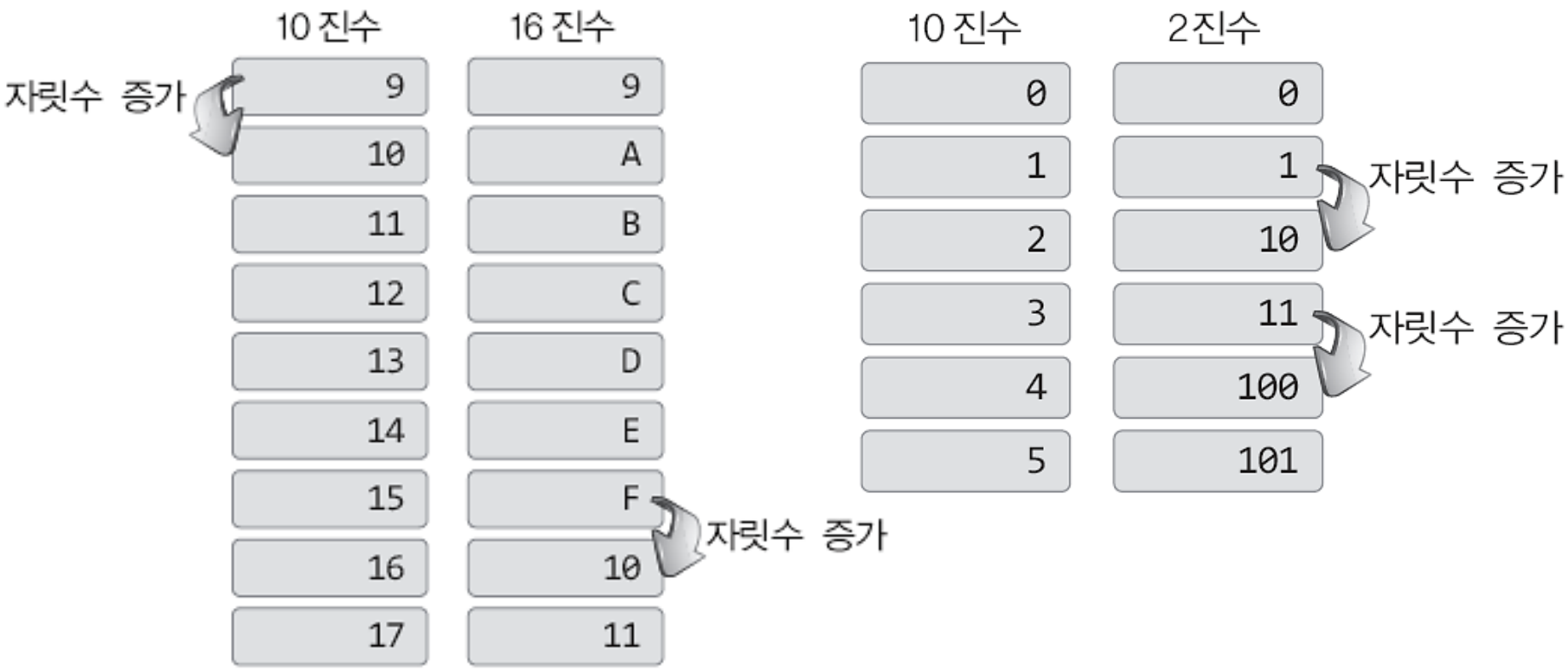
윤성우 저 열혈강의 C 프로그래밍 개정판

2진수란 무엇인가? 10진수, 16진수란 무엇인가?

- 2진수
  - 두 개의 기호를 이용해서 값(데이터)를 표현하는 방식
- 10진수
  - 열 개의 기호를 이용해서 값(데이터)를 표현하는 방식
- N진수
  - N개의 기호를 이용해서 값(데이터)를 표현하는 방식

2진수란 무엇인가? 10진수, 16진수란 무엇인가?

- 16진수
  - 0~9, a, b, c, d, e, f를 이용해서 데이터를 표현함

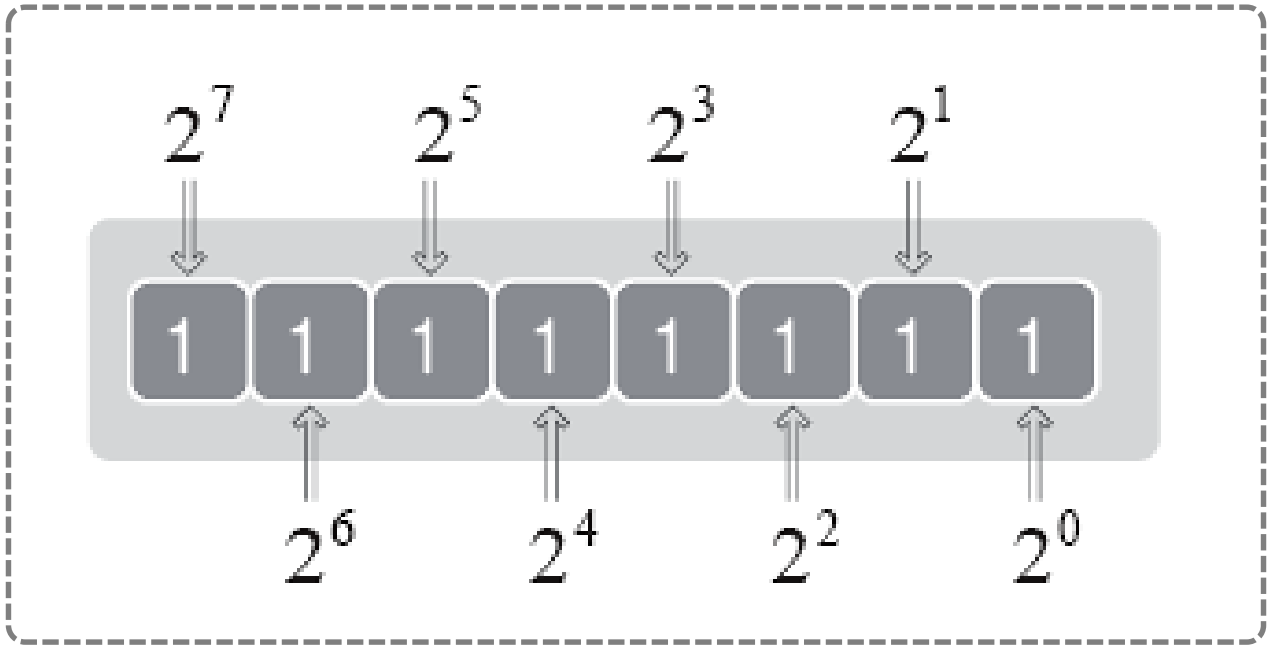


데이터의 표현단위인 비트(Bit)와 바이트(Byte)

- 컴퓨터의 메모리 주소 값은 1바이트당 한 개의 주소가 할당되어 있음
- 바이트는 컴퓨터에 있어서 상당히 의미가 있는 단위임

데이터의 표현단위인 비트(Bit)와 바이트(Byte)

- 아래의 의미 있는 정보를 이용하면 2진수를 쉽게 10진수로 변환 가능



16진수를 이용한 데이터 표현

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

```
int main(void) {
    int num1 = 0xA7, num2 = 0x43;
    printf("0xA7의 10진수 값: %d\n", num1);
    printf("0x43의 10진수 값: %d\n", num2);

    printf("%d - %d = %d\n", num1, num2, num1-num2);
    return 0;
}
```

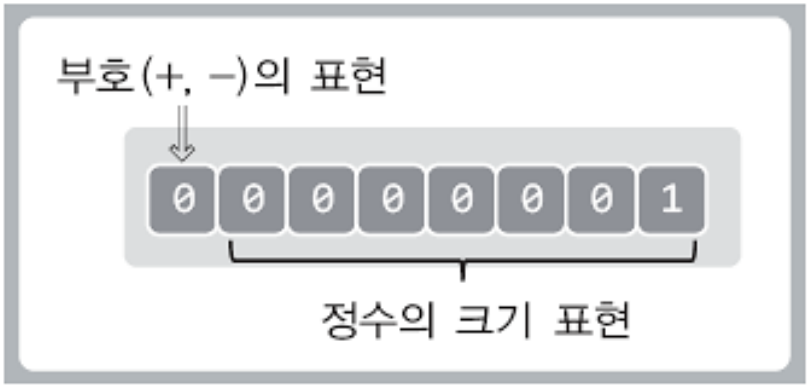
윤성우의

열혈 C 프로그래밍

Chapter 04-2. 정수와 실수의 표현방식

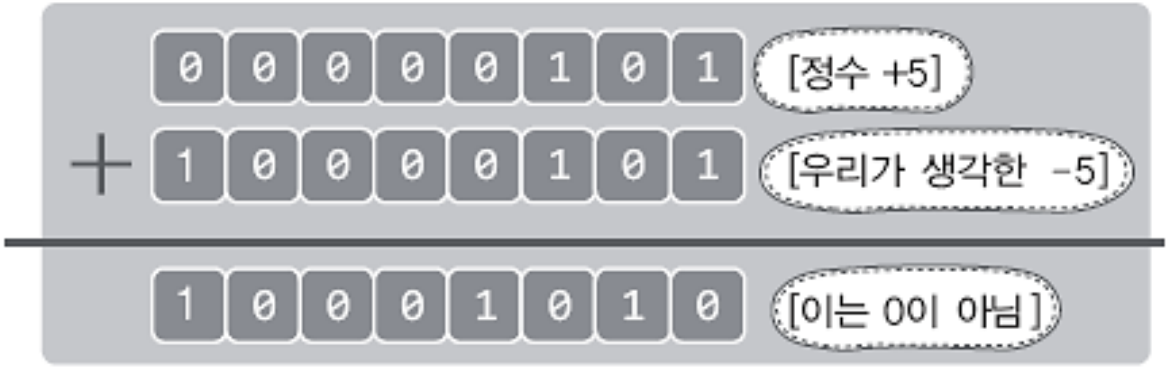
윤성우 저 열혈강의 C 프로그래밍 개정판

정수의 표현방식



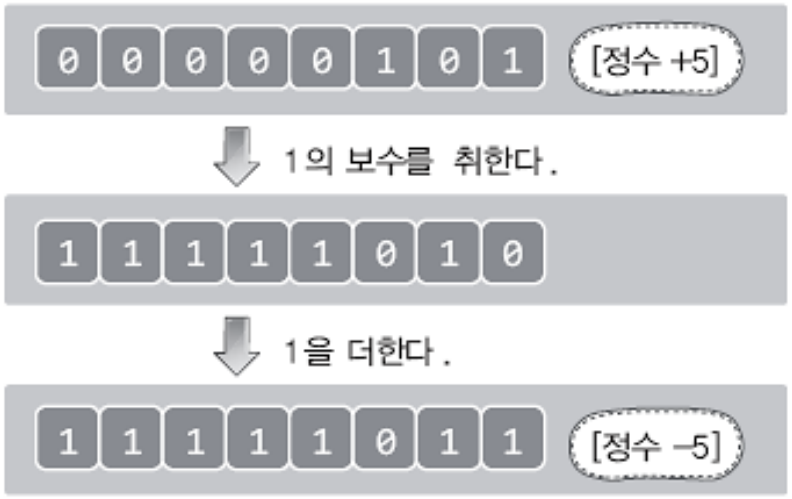
- 가장 왼쪽 비트를 MSB(Most Significant Bit)라고 함
- signed 자료형에서 MSB는 부호를 나타냄
  - MSB를 제외한 나머지 비트는 크기를 나타내는데 사용됨

- 정수를 표현할 때에는 바이트가 늘어나면 그만큼 넓은 범위이 정수를 표현할 수 있음 (예: char, short, int 등)

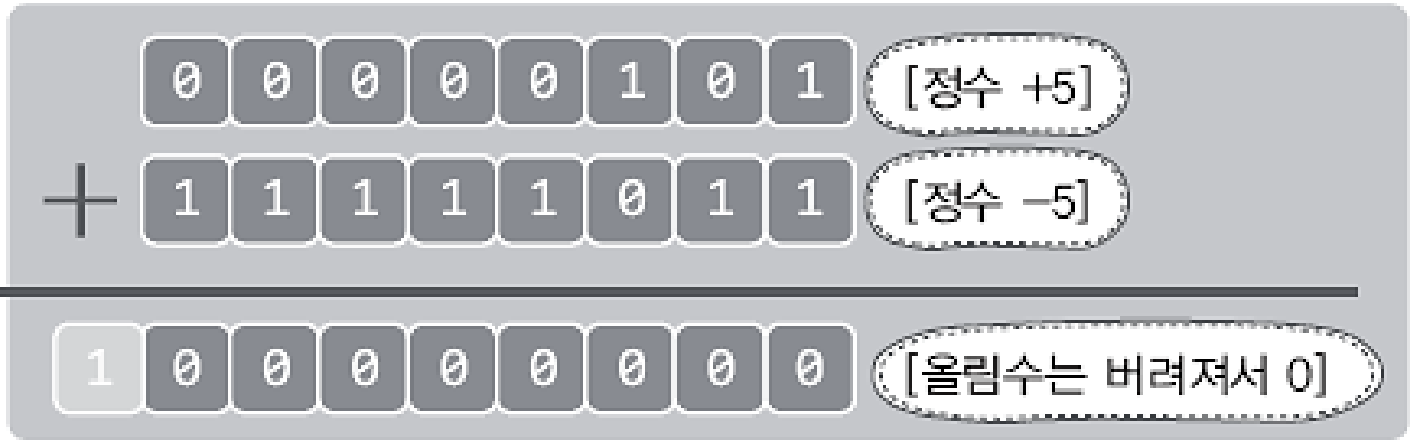


- 단순히 부호 비트만 바꾸어서 정수를 표현하면 안됨

음의 정수 표현방식



- 음의 정수를 표현하는 방식은 2의 보수를 취함
  - 비트를 전환한 후에 1을 더함

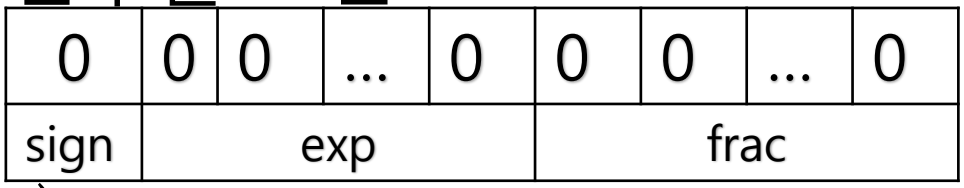


- 2의 보수 표현법이 음수를 표현하는데 맞는지 확인

실수 표현방식

- 일반적으로 대부분의 프로그래밍 언어들은 IEEE에서 정한 표준 방식을 따름
- 실수 표현방식에서는 정밀도를 포기하고 표현할 수 있는 값의 범위를 넓힘
- 따라서 컴퓨터는 완벽하게 정밀한 실수를 표현 못함
- IEEE에서는 아래 식으로 실수를 표현

$$(-1)^{sign} \times frac \times 2^{\exp}$$



- 단정밀도(single-precision)
  - exp – 8비트, frac – 23비트
- 배정밀도(double-precision)
  - exp – 11비트, frac – 52비트

실수 표현의 오차 확인하기

- 실수의 표현 방법때문에 오차없이 모든 실수를 완벽하게 표현할 수 있는 컴퓨팅 환경은 없음
- 실수 표현때문에 발생하는 오차 발생

```
int main(void) {
    int i;
    float num = 0.0;
    for(i = 0; i < 100; i++)
        num += 0.1; // 이 연산을 100회 진행

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

실행결과

```
0.1을 100번 더한 결과: 10.000002
```

윤성우의
열혈 C 프로그래밍

Chapter 04-3. 비트 연산자

윤성우 저 열혈강의 C 프로그래밍 개정판

비트 연산자 (비트 이동 연산자 포함)

연산자	기능	
&	비트단위로 AND연산을 함 예) num1 & num2;	→
|	비트단위로 OR연산을 함 예) num1 | num2;	→
^	비트단위로 XOR 연산을 함 예) num1 ^ num2;	→
~	단항 연산자로서 피연산자의 모든 비트를 반전시킴 예) ~num /* num은 변화없음. 반전 결과만 반환함 */	←

비트 연산자 (비트 이동 연산자 포함)

연산자	기능	
<<	피연산자의 비트 열을 왼쪽으로 이동시킴 예) num << 2; num은 변화없음. 왼쪽으로 두 칸 이동 결과만 반환	
>>	피연산자의 비트 열을 오른쪽으로 이동시킴 예) num >> 2; num은 변화없음. 오른쪽으로 두 칸 이동 결과만 반환	

& 연산자: 비트 단위 AND

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

실행결과

```
AND 연산의 결과: 4
```

0	0	0
0	1	0
1	0	0
1	1	1

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

| 연산자: 비트 단위 OR

```
int main(void) {
    int num1 = 15; /* 0~0 0~0 0~0 00001111 */
    int num2 = 20; /* 0~0 0~0 0~0 00010100 */
    int num3 = num1 | num2;
    printf("OR 연산의 결과: %d\n", num3);
    return 0;
}
```

실행결과

```
OR 연산의 결과: 31
```

0	0	0
0	1	1
1	0	1
1	1	1

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

^ 연산자: 비트 단위 XOR

```
int main(void) {
    int num1 = 15; /* 0~0 0~0 0~0 00001111 */
    int num2 = 20; /* 0~0 0~0 0~0 00010100 */
    int num3 = num1 ^ num2;
    printf("XOR 연산의 결과: %d\n", num3);
    return 0;
}
```

실행결과

```
XOR 연산의 결과: 27
```

0	0	0
0	1	1
1	0	1
1	1	0

```
          00000000 00000000 00000000 000 01111
^ 연산    00000000 00000000 00000000 000 10100
-------------------------------------------------
          00000000 00000000 00000000 000 11011
```

~ 연산자

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

실행결과

```
NOT 연산의 결과: -16
```

0	1
1	0

~ 연산 전 00000000 00000000 00000000 00001111

~ 연산 후 11111111 11111111 11111111 11110000

<< 연산자: 비트 왼쪽 이동(Shift)

- num1 << num2
  - num1의 비트 열을 num2칸씩 왼쪽으로 이동시킨 결과 반환
  - 8 << 2 정수 8의 비트열을 2칸씩 왼쪽으로 이동시킴

```
int main(void) {
    int num = 15; /* 0~0 0~0 0~0 00001111 */
    int result1 = num << 1;
    int result2 = num << 2;
    int result3 = num << 3;
    printf("1칸 이동: %d\n 2칸 이동: %d\n 3칸 이동:
%d\n", result1, result2, result3);
    return 0;
}
```

실행결과

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

<< 연산자: 비트 왼쪽 이동(Shift)

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

- 왼쪽으로 이동하면서 생기는 오른쪽 빈 자리는 0으로 채움
- 왼쪽으로 한 칸씩 이동할 때마다 정수 값은 2 배씩 증가
- 오른쪽으로 한 칸씩 이동할 때마다 정수 값은 절반으로 줄어듬

>> 연산자: 비트 오른쪽 이동(Shift)

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

⬇ **CPU에 따라서 달라지는 연산의 결과**

```
00111111 11111111 11111111 11111100 // 0으로 채움
11111111 11111111 11111111 11111100 // 1로 채움
```

- 양수의 경우에는 오른쪽으로 비트 이동 시 0으로 채움 (문제없음)
- 음수의 경우에는 오른쪽으로 비트 이동 시 CPU에 따라 0으로 채우거나 1로 채움 (결과가 달라짐)
- 호환성이 요구되는 경우에는 >> 연산자의 사용에 주의를 기울여야 함

>> 연산자: 비트 오른쪽 이동(Shift)

```
int main(void) {
    int num = -16; /* 1~1 1~1 1~1 11110000 */
    printf("2칸 오른쪽으로 이동: %d\n", num >> 2);
    printf("3칸 오른쪽으로 이동: %d\n", num >> 3);
    return 0;
}
```

실행결과

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