



# 시스템 프로그래밍 기초

Introduction to System Programming

ICT융합학부 조용우

## 7. Bitwise Operators and Enumeration Types



### 비트단위 연산자

- 이진숫자의 문자열로 표현된 정수적 수식에 사용
- 시스템 종속적
- 여기서는 8 비트 바이트, 4 바이트 워드, 2의 보수로 표현되는 정수, 그리고 ASCII 문자 코드를 갖는 컴퓨터를 가정함

## 7.1 Bitwise Operators and Expressions

### 비트단위 연산자

비트단위 연산자 (bitwise operators)		
논리 연산자 (logical operators)	(단항) 비트단위 보수	~
	비트단위 논리곱	&
	비트단위 배타적 논리합	^
	비트단위 논리합	
이동 연산자 (shift operators)	왼쪽 이동	<<
	오른쪽 이동	>>

### signed int의 음수 표현

- 양수

- 부호 비트는 0으로 놓고, 남은 숫자로 2진수 표현

- 음수

- 부호절대값 방법

- ▶ 부호 비트를 제외한 수를 양수값으로 읽고, 마이너스 붙임
    - ▶ 인간의 입장에서 직관적이지만, 음수의 덧셈이 양수의 뺄셈과 처리 방법이 달라짐
    - ▶ 해결을 위해선 피연산자의 절대값을 서로 비교하는 등 추가 연산 필요

### signed int의 음수 표현

#### ■ 음수

##### → 1의 보수 방법

- ▶ 양수값의 비트를 반전시켜서 음수를 표현
- ▶ 음수의 덧셈과 양수의 뺄셈이 동일하게 처리됨
- ▶ 0을 나타내는 값이 0000...0 과 1111...1의 두 가지
- ▶ 덧셈 연산 시 end around carry가 발생해서 1을 더해줘야 함

##### → 2의 보수 방법

- ▶ 1의 보수에 1을 더하는 방법
- ▶ 1111...1이 -1을 의미
- ▶ 음의 부호를 붙일 때 1을 더해주는 연산을 해야함

## 7.1 Bitwise Operators and Expressions

**signed int의 음수 표현: w비트로 표현가능한 범위**

타입	범위
부호 절대값 방법	$-(2^{w-1}-1) \sim (2^{w-1}-1)$
1의 보수의 방법	$-(2^{w-1}-1) \sim (2^{w-1}-1)$
2의 보수의 방법	$-2^{w-1} \sim (2^{w-1}-1)$

## 7.1 Bitwise Operators and Expressions

### 비트단위 보수 : ~ 연산자

- 1의 보수 연산자 또는 비트단위 보수 연산자
- 주어진 인자의 비트열 표현을 반대로 함; 0은 1로, 1은 0으로
- 예

```
int a = 70707;
```

→ a의 이진수 표현

```
00000000 00000001 00010100 00110011
```

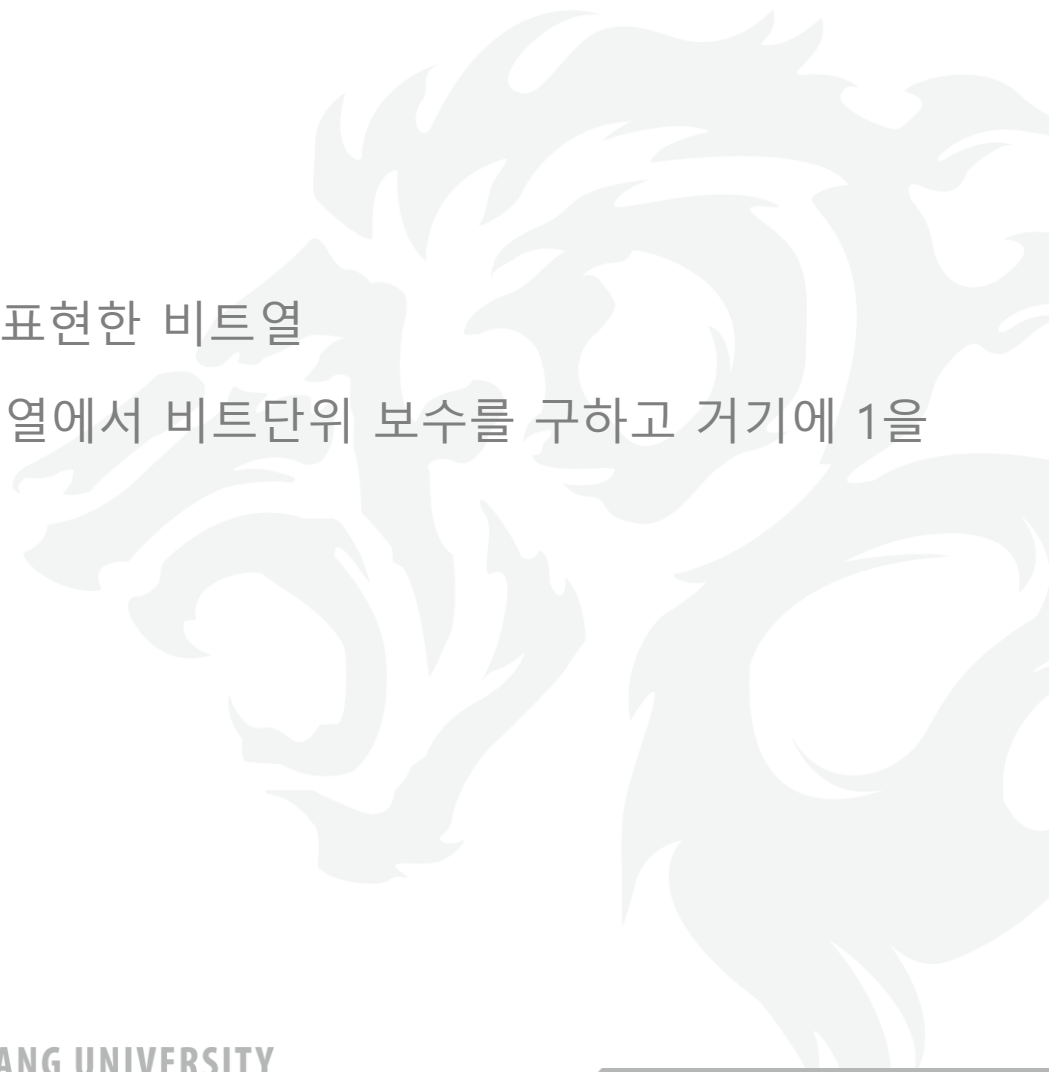
→ ~a의 이진수 표현

```
11111111 11111110 11101011 11001100(-70708)
```



### 2의 보수

- $n$ 이 음이 아닌 정수일 때
  - $n$ 의 2의 보수는  $n$ 을 이진수로 표현한 비트열
  - $-n$ 의 2의 보수표현은  $n$ 의 비트열에서 비트단위 보수를 구하고 거기에 1을 더한 비트열



## 7.1 Bitwise Operators and Expressions

### 2의 보수

Value of n	Binary representation	Bitwise complement	Two's complement representation of -n	Value of -n
7	00000000 00000111	11111111 11111000	11111111 11111001	-7
8	00000000 00001000	11111111 11110111	11111111 11111000	-8
9	00000000 00001001	11111111 11110110	11111111 11110111	-9
-7	11111111 11111001	00000000 00000110	00000000 00000111	7

#### ■ 참고

- 0의 2의 보수: 모든 비트가 0
- -1의 2의 보수: 모든 비트가 1

### 비트단위 이진 논리 연산자

- 이진 논리 연산자
- $\&$ (논리곱),  $\wedge$ (배타적 논리합),  $\mid$ (논리합)
- 정수적 수식을 피연산자로 가짐
- 적절히 형 변환된 두 피연산자는 대응되는 비트끼리 연산됨

a	b	a & b	a ^ b	a   b
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

## 7.1 Bitwise Operators and Expressions

(예)

Declaration and initializations

```
int a = 33333, b = -77777;
```

Expression	Representation				Value
a	00000000	00000000	10000010	00110101	33333
b	11111111	11111110	11010000	00101111	-77777
a & b	00000000	00000000	10000000	00100101	32805
a ^ b	11111111	11111110	01010010	00011010	-110054
a   b	11111111	11111110	11010010	00111111	-77249
~(a   b)	00000000	00000001	00101101	11000000	77248
(~a & ~b)	00000000	00000001	00101101	11000000	77248

### 왼쪽과 오른쪽 이동 연산자

- 이동 연산자의 두 피연산자는 정수적 수식이어야 함
- 각 피연산자에 정수적 승격이 일어남
- 전체 수식의 형은 승격된 왼쪽 피연산자의 형이 됨

### 왼쪽 이동 연산자

**`expr1 << expr2`**

- `expr1`의 비트 표현을 `expr2`가 지정하는 수만큼 왼쪽으로 이동
- 하위 비트로는 0이 들어옴
- 수식에서 `c`는 `int` 형으로 승격됨
- 따라서 결과는 4 바이트에 저장됨

## 7.1 Bitwise Operators and Expressions

### 왼쪽 이동 연산자

Declaration and initialization

```
char c = 'Z';
```

Expr	Representation	Action
c	00000000 00000000 00000000 01011010	unshifted
c << 1	00000000 00000000 00000000 10110100	left-shifted 1
c << 4	00000000 00000000 00000101 10100000	left-shifted 4
c << 31	00000000 00000000 00000000 00000000	left-shifted 31

### 오른쪽 이동 연산자

**`expr1 >> expr2`**

- 왼쪽 이동 연산자와 대칭적이지 않음
- 부호가 없는 정수적 수식에서는 상위 비트로 0이 들어옴
- 부호가 있는 형일 때에는 시스템에 따라 상위 비트로 0이 들어오는 것도 있고, 1이 들어오는 것도 있음



## 7.1 Bitwise Operators and Expressions

### 오른쪽 이동 연산자

#### Declaration and initialization

```
int      a = 1 << 31;  /* shift 1 to the high bit */
unsigned b = 1 << 31;
```

Expr	Representation	Action
a	10000000 00000000 00000000 00000000	unshifted
a >> 3	11110000 00000000 00000000 00000000	right-shifted 3
b	10000000 00000000 00000000 00000000	unshifted
b >> 3	00010000 00000000 00000000 00000000	right-shifted 3

## 7.1 Bitwise Operators and Expressions

### 이동 연산자

Declaration and initialization

```
unsigned a = 1, b = 2;
```

Expression	Equivalent expression	Representation	Value
<code>a &lt;&lt; b &gt;&gt; 1</code>	<code>(a &lt;&lt; b) &gt;&gt; 1</code>	00000000 00000010	2
<code>a &lt;&lt; 1 + 2 &lt;&lt; 3</code>	<code>(a &lt;&lt; (1 + 2)) &lt;&lt; 3</code>	00000000 01000000	64
<code>a + b &lt;&lt; 12 * a &gt;&gt; b</code>	<code>((a + b) &lt;&lt; (12 * a)) &gt;&gt; b</code>	00001100 00000000	3072

### 마스크

- 마스크 : 다른 변수나 수식으로부터 원하는 비트를 추출하는데 사용되는 상수나 변수

- int 형 상수 1의 비트 표현:

**00000000 00000000 00000000 00000001**

→ 이것을 사용하여 int 형 수식의 최하위 비트를 알아낼 수 있음

### 마스크

- 예제 코드 1

```
int i, mask = 1;  
for (i = 0; i < 10; ++i)  
printf("%d", i & mask);
```

- 예제 코드 2

```
(v & (1 << 2)) ? 1 : 0
```

- 예제 코드 3

```
v & 255
```



### 패킹 - 4개의 문자를 하나의 int형에 패킹하는 함수

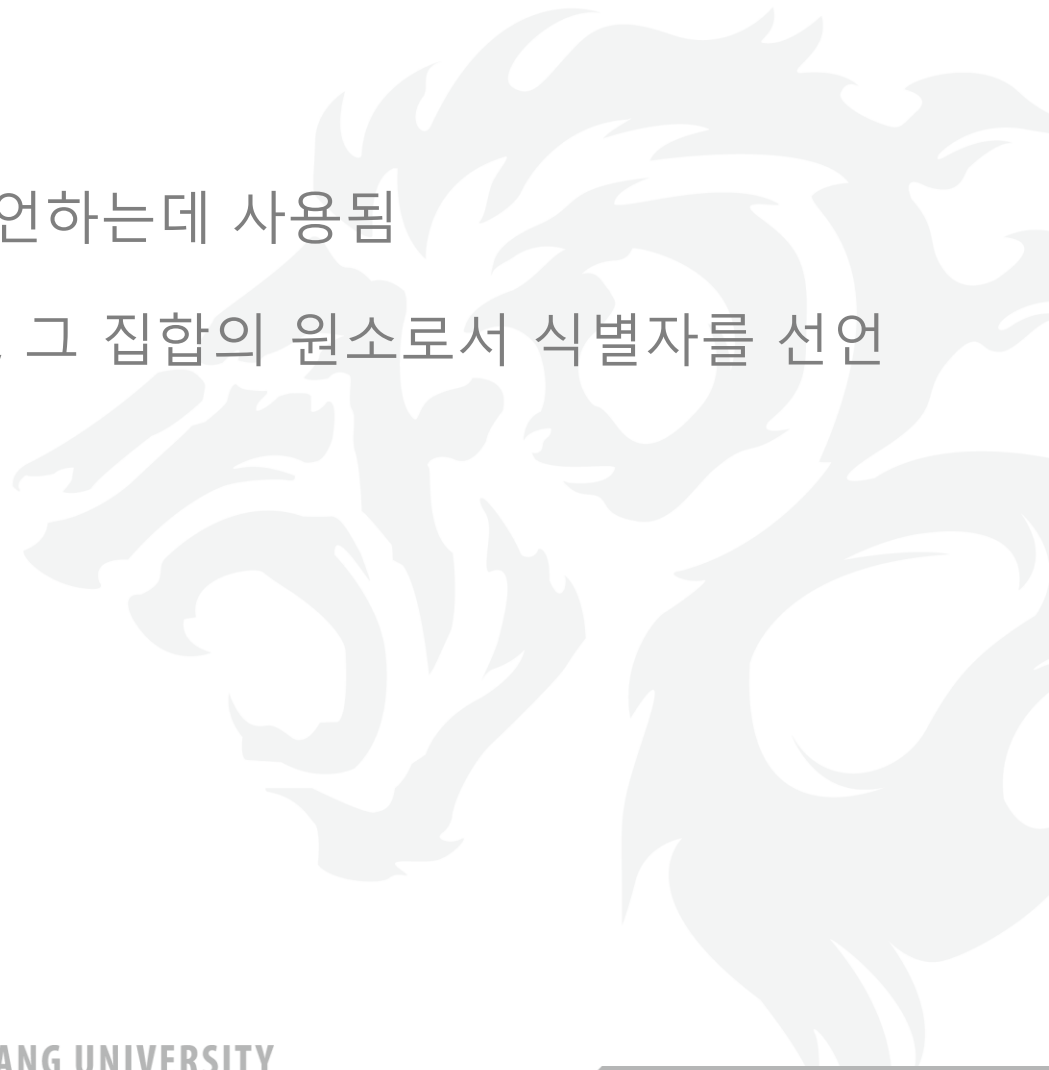
```
/* Pack 4 characters into an int. */  
  
#include <limits.h>  
  
int pack(char a, char b, char c, char d)  
{  
    int p = a;          /* p will be packed with a, b, c, d */  
  
    p = (p << CHAR_BIT) | b;  
    p = (p << CHAR_BIT) | c;  
    p = (p << CHAR_BIT) | d;  
    return p;  
}
```

### 언패킹 - 32비트 int안에 있는 문자를 검색하는 함수

```
/* Unpack a byte from an int. */  
  
#include <limits.h>  
  
char unpack(int p, int k)          /* k = 0, 1, 2, or 3 */  
{  
    int      n = k * CHAR_BIT;    /* n = 0, 8, 16, or 24 */  
    unsigned mask = 255;          /* low-order byte */  
  
    mask <<= n;  
    return ((p & mask) >> n);  
}
```

### 열거형

- 키워드 enum은 열거형을 선언하는데 사용됨
- 이것은 유한집합을 명명하고, 그 집합의 원소로서 식별자를 선언하는 수단을 제공함



### 열거형

- 예제

```
enum day {sun, mon, tue, wed, thu, fri, sat};
```

- 사용자 정의형 enum day 생성
- day : 태그 이름
- 열거자는 식별자 sun, mon, ..., sat이고, 이들은 int 형 상수임
- 디폴트로 첫 번째 원소는 0이고, 각 원소는 이전 원소의 값보다 하나 큰 값을 가짐
- 이것은 형 정의임



### 열거형

- enum day 형 변수의 선언

```
enum day d1, d2;
```

- d1, d2 변수의 사용

```
d1 = fri;
```

```
if (d1 == d2)
```

```
..... /* do something */
```



### 열거형

- 선언 예제 1

```
enum suit {clubs = 1, diamonds, hearts, spades} a, b, c;
```

- 선언 예제 2

```
enum fruit {apple = 7, pear, orange = 3, lemon} frt;
```

- 선언 예제 3

```
enum veg {beet = 17, carrot = 17, corn = 17} vege1, vegw2;
```

- 선언 예제 4

```
enum {fir, pine} tree;
```

- 선언 예제 5

```
enum veg {beet, carrot, corn} veg;
```

## Homework

- Exercises #1, 6, 8, 21, 26

