



4. 데이터 형(data type)

```
#include <stdio.h>
```

```
void main() {  
    char c=129;  
    printf("%d\n",c);  
}
```

- 위 프로그램의 실행결과는 얼마일까요?

“char는 1바이트 정수형인데, 앞에 선언 변경자 unsigned 가 없으므로, 부호있는signed 1바이트 정수형입니다. C 언어는 정수를 나타내기 위해 2의 보수를 사용하는데, 129의 부호 있는 2의 보수 표현은 -127이므로, -127이 출력됩니다.”

정수의 음수를 표현하는 방법

- 부호와 가중치(sign & magnitude)에 의한 방법
- 1의 보수(1's complement)
- 2의 보수(2's complement) 방법

4. 변수는 쓰기 전에 선언해야 한다 3

- 부호와 가중치에 의한 방법은 가장 왼쪽 비트(MSB; most significant bit■)를 부호 비트(sign bit)로 사용하며, 나머지 비트들을 가중치(magnitude)로 사용

표현	값
000	0
001	1
010	2
011	3
100	-0
101	-1
110	-2
111	-3

- 음수를 표현하지 않는다면, 0 ~ 7까지의 수를 표현하는 것이 가능합니다.

$$0 \sim 2^n - 1$$

- 위의 경우는 3비트를 사용하므로, $0 \sim 2^3 - 1$ 의 범위의 정수를 표현할 수 있습니다.

- 이진수 110을 십진수 -2로 결정하는 것은 두 단계를 거칩니다.

(1) 부호 결정: MSB가 1이므로, 음수입니다.

(2) 값 결정: MSB를 제외한 나머지 비트들의 가중치를 계산합니다. 이진수 10이므로 그 값은 2입니다.

- 1의 보수와 2의 보수 각각에 대해 값을 결정한 표를 아래에 나타내었습니다.

표현	값	표현	값
000	0	000	0
001	1	001	1
010	2	010	2
011	3	011	3
100	-3	100	-4
101	-2	101	-3
110	-1	110	-2
111	-0	111	-1

- 이진수 100의 보수에 의한 값은 다음 두 단계를 거쳐 결정합니다.

(1) 부호 결정: MSB가 1이므로 음수입니다.

(2) 값 결정: MSB를 포함한 전체 비트의 보수가 값입니다. 100의 1의 보수는 011이므로 3입니다. 100의 2의 보수는 100이므로 4입니다.

- 이진수 100은 1의 보수에 의해 $-3_{(10)}$, 2의 보수에 의해 $-4_{(10)}$ 입니다.

- 2의 보수 표현이 음수를 표현하는지, 양수만 표현하는지 어떻게 알 수 있을까요?

`unsigned char i;`

- 부호 있는 정수형은 다음과 같이 선언합니다.

`signed char i;`

- 2바이트, 4바이트 정수는 char 대신에 각각 short, int를 사용합니다. 그러므로 4바이트 부호 있는 정수는 아래와 같이 선언합니다.

```
signed int i;
```

- 일반적으로 부호의 유무를 나타내는 signed 나 unsigned는 생략하면 signed가 되므로, signed를 생략하여 아래와 같이 나타냅니다.

```
int i;
```

- short나 long다음에는 이것이 정수(integer)임을 확실히 하기 위하여 **int**가 위치해도 됩니다.

```
long int i;
```

```
[signed|unsigned] [short|long] int <variable>;
```



초기화(initialization)

- 아래의 예는 정수형 변수 i를 선언한 뒤 i를 1로 초기화하고 있습니다.

```
int i;  
i=1;
```

- 위의 코드는 다음과 같이 간단하게 적을 수 있습니다.

```
int i=1;
```

- 초기값은 브레이스({})로 묶어도 됩니다.

```
int i={1};
```



```
#include <stdio.h>
```

```
void main() {  
    int i;  
    int j=2;  
    float k={1.2};  
  
    i=1;  
    printf("%d,%d,%f\n", i, j, k);  
}
```

- 결과는 아래와 같습니다.

1,2,1.200000■



함수형 초기화(functional initialization)

- C++의 클래스(class)가 소개된 이후 기본형의 함수형 초기화가 가능해졌습니다.

```
int i=1;
```

은

```
int i(1);
```

처럼 적을 수 있습니다.

- 비록 후자의 i선언이 함수 i를 호출하는 것 같지만, 이것은 정수 변수 i를 1로 초기화하는 문장입니다. 함수 호출처럼 보이므로, 함수형 초기화라고 합니다.



실수(real number)는 어떻게?

- 고정 소수점(fixed point) 표기
- 2.3을 고정 소수점 표기법으로 나타내면, 2는 0000 0010, 3은 0000 0011 이므로 2바이트 실수를 저장할 때, 다음과 같이 저장할 수 있습니다.

0000 0010 0000 0011

- 소수점 이상 부분이 음수를 표현한다면, -128 ~ 127까지를 표현할 수 있습니다.
- 소수점 이하부분은 음수를 나타낼 필요가 없으므로, 0 ~ 255까지 표현할 수 있습니다.
- 그러므로 표현할 수 있는 가장 작은 수는 -128.0이며 가장 큰 수는 127.255입니다.



부동 소수 표현(floating point notation)

- 4바이트 실수(float) -14.24 는 컴퓨터에 어떻게 저장될까요?
- 아래의 예는 실수 -14.24 를 16진수로 인쇄하는 프로그램입니다.
- -14.24 는 16진수로 아래와 같이 표현됩니다.

0xc163d70a

- 2진수로는 아래와 같이 표현됩니다.

1100 0001 0110 0011 1101 0111 0000 1010

```
#include <iostream>
```

```
struct BITFIELD {//이것은 비트 필드 구조체입니다.  
    unsigned m0:4;//아주 특별해 보이는 이 선언은 m0가 4비트를 차지하는  
        //것을 의미합니다.
```

```
    unsigned m1:4;
```

```
    unsigned m2:4;
```

```
    unsigned m3:4;
```

```
    unsigned m4:4;
```

```
    unsigned m5:4;
```

```
    unsigned m6:4;
```

```
    unsigned m7:4;
```

```
};//struct BITFIELD
```

```
union FLOAT {//공용체는 필드(field)를 공유합니다. f와 b는 같은 메모리를  
    //공유하며, f의 크기가 32비트, b의 크기 역시 32비트이므로  
    //실제 메모리는 64비트가 아닌 32비트가 할당됩니다.
```

```
    float f;
```

```
    BITFIELD b;
```

```
};//union FLOAT
```

```

void main() {
    FLOAT f;

    f.f=-14.24;
    //cout은 ostream 클래스의 전역 객체입니다.
    std::cout << f.f << std::endl << std::hex;
        //endl과 hex 조작자(manupulator)는 출력
        //스트림의 형식(format)을 조작합니다.
        //이것은 '함수 포인터'에서 자세히 다룹니다.
    std::cout << f.b.m0 << f.b.m1 << f.b.m2 << f.b.m3
        << f.b.m4 << f.b.m5 << f.b.m6 << f.b.m7 << std::endl;
    //output: c163d70a■
}

```



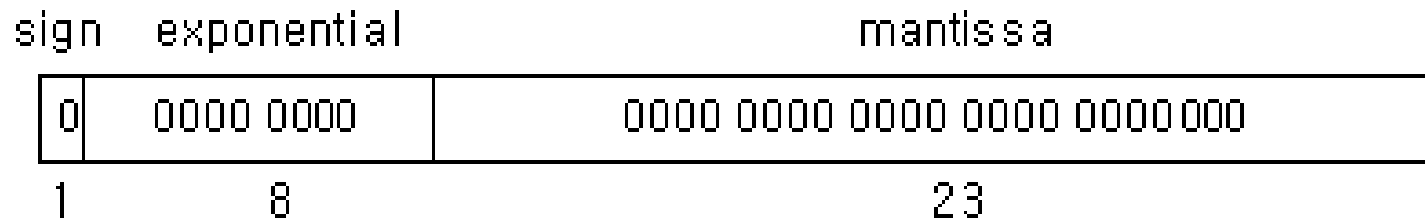
프로그램의 실제 출력 결과는 a07d361c인데 이것은 인텔 CPU가 역워드 방식으로 수(number)를 저장하기 때문입니다. 이 사항은 후에 '연산자'에서 자세히 다룹니다.

- 실수를 수학적으로 나타내는 방법을 살펴봅시다.
- -14.24를 나타내는 방법은 다양합니다.

...
 -0.01424×10^3
 -0.1424×10^2
 -1.424×10^1
 -14.24
 -142.4×10^{-1}
 -1424×10^{-2}
 -14240×10^{-3}
 ...

- 컴퓨터는 이러한 방법 중에 소수점 이상의 자리를 모두 0으로 한 표현법인 -0.1424×10^2 을 사용하는데, 소수점 이하 부분과 지수 부분만을 관리하는 이점과 큰 수와 정밀한 수를 같이 표현할 수 있는 이점 때문입니다.
- 이렇게 임의의 실수를 유효자리 수가 모두 소수점 아래에 오도록 변환하는 것을 **정규화normalization**라고 합니다.

- 정규화의 결과를 컴퓨터에 세부분으로 나누어 저장할 수 있습니다.
- **부호sign, 지수exponential와 가수mantissa 부분**이 그것입니다.
- -14.24의 경우 부호는 음수, 지수는 2, 가수는 1424를 저장합니다.
- 모든 C컴파일러는 4바이트 부동 소수점을 나타내기 위해 IEEE 754 표준 방식(IEEE 32-bit standard)을 사용하고 있으며, 이것은 제안된 키워드 **float**를 사용합니다.
- float는 부호, 지수와 가수자리를 위해 각각 1, 8 그리고 23비트를 사용합니다.



부동 소수 표기: float(IEEE 32-bit standard)는 부호를 위해 1비트, 지수를 위해 8 비트, 가수를 위해 23비트, 모두 32비트를 사용합니다.

- float의 정규화 식은 아래와 같습니다.

$$(-1)^s(1.m)2^{e-127}$$

- 위의 식에서 s, m 과 e 를 결정하여, 각각의 32비트 자리를 채웁니다.
- s 가 1이면 $(-1)^1$ 은 -1이므로 음수를 의미하며, 0이면 $(-1)^0=1$ 이므로 양수를 의미합니다.
- 정규화를 하면, 소수점 이하 첫 번째 비트는 항상 1이므로 이 공간을 절약하기 위해, 처음 1은 항상 있는 것으로 간주하여, m 의 자리에 포함시키지 않습니다.
- e 는 음수를 나타내는 것을 피하기 위하여, 128초과 코드(excess-128 code)를 사용합니다. 예를 들면, e 가 0이면, 지수는 -127을 의미하며, e 가 128이면 1을 의미한다.

- -14.24의 s,e와 m을 결정해 봅시다.

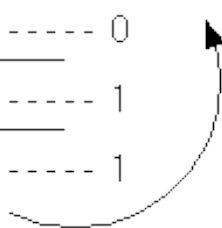
■ 단계 1: s의 결정

부호는 음수이므로 $s=1$ 입니다.

■ 단계 2: m의 결정

먼저 14.24의 2진수를 구해야 한다. 14는 이진수로 아래와 같습니다.

1110

$$\begin{array}{r}
 2 \overline{) 14} \\
 \underline{2} \\
 2 \overline{) 7} \text{ ----- } 0 \\
 \underline{2} \\
 2 \overline{) 3} \text{ ----- } 1 \\
 \underline{2} \\
 1 \text{ ----- } 1
 \end{array}$$


0.24는 2진수로 아래와 같습니다.

0.0011 1101 0111 0000 1010

0.24	1.36
$\times 2$	$\times 2$
<hr/>	<hr/>
0.48	0.72
$\times 2$	$\times 2$
<hr/>	<hr/>
0.96	1.44
$\times 2$	$\times 2$
<hr/>	<hr/>
1.92	0.88
$\times 2$	$\times 2$
<hr/>	<hr/>
1.84	1.76
$\times 2$	$\times 2$
<hr/>	<hr/>
1.68	1.52
$\times 2$...
<hr/>	
1.36	

- 14.24는 아래와 같습니다.

1110.0011 1101 0111 0000 1010

- 정규화 과정을 거치면 아래와 같이 표현됩니다.

1.1100 0111 1010 1110 0001 010 $\times 2^3$

- 그러므로, 부동소수점 표현의 m은 소수점 이하 부분인 아래의 이진열이 됩니다.

1100 0111 1010 1110 0001 010

■ 단계 3: e의 결정

- 정규화의 결과 지수 부분이 3이므로, $e=130(e-127=3)$ 입니다. 130의 2진수는 아래와 같습니다.

1000 0010

- 그러므로 위 프로그램의 출력 결과는 아래와 같습니다.

C163D70A

sign	exponential	mantissa
1	0000 0000	0000 0000 0000 0000 0000000
1	8	23

sign	exponential	mantissa
1	0000 0000	1100 0111 1010 1110 0001010
1	8	23

sign	exponential	mantissa
1	1000 0010	1100 0111 1010 1110 0001010
1	8	23



-14.24의 float 표현: -14.24는 위의 그림처럼 저장됩니다. 이것을 16진수로 표현하면, C163D70A입니다.



형의 종류(sort of types)

■ 기본형(fundamental type)

char : 1바이트의 정수를 나타낼 수 있습니다.

short: 2바이트의 정수를 나타낼 수 있습니다.

long : 4바이트의 정수를 나타낼 수 있습니다.

float : 4바이트로 실수를 표현합니다.

double : 8바이트로 실수를 표현합니다.

bool : C++에 새로 추가된 형으로 false,true를 표현합니다.

wchar_t : Wide CHARACTER Type을 의미하는데, 2바이트 문자형을 나타냅니다.

■ 복합형(compound type)

- array : 배열array은 같은 형의 변수를 하나의 대표이름으로 여러 개 선언하기 위해 사용합니다.
- pointer : 주소 변수를 선언하기 위해서 사용합니다.
- reference : 참조reference는 C++에 새로 추가되었습니다. 파라미터 전달에 사용된 참조는 파라미터의 값을 전달하지 않고 참조를 전달합니다.
- class : 클래스class는 다른 모든 데이터 형을 포함할 수 있는 자료 구조입니다. C++에 추가되었습니다.
- struct : C에서 구조체는 여러 다른 타입을 포함하는 타입입니다. C++에서 구조체는 클래스와 거의 같습니다.
- union : 구조체와 거의 같지만, 필드field가 같은 메모리 공간을 공유합니다.
- pointer to non-static class member : C++에 추가된 포인터 타입 입니다. 기본 주소에 대한 상대 주소relative address로 데이터를 참조하기 위해서 사용합니다. 새로운 연산자 `.*` 와 `->*` 를 사용해서 접근할 수 있습니다.

특별한 형 void

- 함수 선언에서 파라미터 리스트가 비었음을 나타냅니다.

```
int func(void); //func()는 파라미터를 가지지 않습니다.
```

- 함수가 값을 리턴하지 않는 것을 나타냅니다.

```
void func(int n); //func()는 값을 리턴하지 않습니다.
```

- 일반 포인터(generic pointer)를 나타냅니다. C에서 void*는 어떠한 것도 가리킬 수 있습니다.

```
void* ptr; //ptr에는 어떤 포인터도 대입할 수 있습니다.
```

- 형 변환type conversion에서 리턴형을 무시합니다.

```
extern int errfunc(); //int타입의 에러 값을 리턴합니다.
```

```
...
```

```
(void)errfunc(); //리턴 값을 버립니다.
```



실습문제

1. 아래 프로그램 코드를 Visual C++ 6.0과 Borland C++ 4.5에서 실행시킨 결과가 있습니다. 결과가 다르게 출력된 이유를 설명하세요.

```
#include <stdio.h>
```

```
int i, j;
```

```
void main() {  
    printf("%p, %p\n", &i, &j);  
}
```

결과(VC++): 004135CC,004135D0

결과(16bit BC++): 6B6F:0ACE,6B6F:0AD0

```
#include <stdio.h>
```

```

void main() {
    int i,j;

    printf("%p,%p\n",&i,&j);
}

```

결과(VC++): 0064FDF8,0064FDF4

결과(16bit BC++): 6F6F:2016,6F6F:2014

```

#include <stdio.h>

```

```

void main() {
    int i;
    int j;

    printf("%p,%p\n",&i,&j);
}

```

결과(VC++): 0064FDF8,0064FDF4

결과(16bit BC++): 1117:2016,1117:2014

2. 소수점 이상의 어떤 수를 2로 계속해서 나눈 나머지를 거꾸로 쓰는 것이 왜 2진수가 됩니까? 책에서 보여준, 소수점 이하의 부분을 2진수로 구하는 과정은 또 왜 그렇습니까?