

Pat O'Sullivan

Mh4718 Week 3

Week 3

0.1 Variable Storage in C++ (Contd.)

0.1.1 Storage of Integers

The topmost bit of byte 4 is reserved for the storage of negative integers.

Negative integers are stored using the “twos complement” scheme. The rules of this scheme are:

1. Convert the absolute value of the integer to binary.
2. Fill in the 32 bits as if we were storing the positive integer.
3. Reading from right to left, leave the all bits up to and including the first 1 unchanged and subsequently reverse each bit.

Example 0.1

If we have the line:

```
int n=-1059;
```

in a C++ program then $|n| = 1059 = (10000100011)_2$ and this would be stored as:

byte 4 byte 3 byte 2 byte 1
00000000 00000000 00000100 00100011

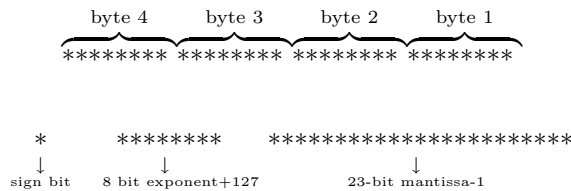
0.1.2 Storage of Non-Integers

If you attempt to assign a non-integer to an **int** type variable only the integer part of the value will be stored. Non-integer values should be assigned to **float** or double variable types.

0.1.2.1 Storage of float type variables.

float type variables are stored using 4 bytes just like int type variables but the bits in the four bytes have a different interpretation.

The value of the variable is first converted to base two normalised scientific notation called *floating point* format by computer scientists. The mantissa and exponent are then stored according to the following scheme:



The stored exponent is the actual exponent + 127 (which allows for storing negative exponents as positive.) The stored mantissa is the actual mantissa less 1. Since the actual mantissa is always 1.*****.....* (being a binary number) there is no need to store the 1.

Example 0.2

The following are the contents of the 4 bytes used to store a **float** type variable. What value is stored?

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32		
1	0	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0

We see that the sign bit is 1 so the number stored is negative.
 The biased exponent is $(1111110)_2 = 126$ therefore the actual exponent is $126 - 127 = -1$.
 The mantissa is $(1.00011111000000000111)_2$.
 The stored value in floating point format (mixed notation) is

$$(1.00011111000000000111)_2 \times 2^{-1}$$

which is $(0.100011111000000000111)_2$ Then

$$\begin{aligned} (0.100011111000000000111)_2 &= 2^{-1} + 2^{-5} + 2^{-6} + 2^{-7} + 2^{-8} + 2^{-9} + 2^{-19} + 2^{-20} + 2^{-21} \\ &= 0.560550212860107421875. \end{aligned}$$

0.1.3 Largest and smallest float values

The cases when all the exponent bits are all 1's or all 0's signals a change in the storage rules.

Once the exponent reaches 11111111 the stored number is treated as infinity and this is known as an *overflow error*.

Therefore the largest positive value that can be stored as a float is

$$01111110111111111111111111111111 = 340282346638528859811704183484516925440$$

Check out the following program. What is happening?

```
float x=pow(2,104);
for(int i = 1;i=24;i++)
{
x*=2;

cout<<x<<" " <<i<<" " <<scientific<<1/x<<endl;
if(1/x==0) cout<<"true"<<endl;
}
```

At the other end of the scale, once the exponent reaches 00000000 the storage rules change. The exponent is held at -126 and the mantissa is treated as 0.***** rather than 1.***** This enables more small numbers to be stored.

The smallest non-zero number that can be stored is therefore:

$$\begin{aligned} &00000000\ 00000000\ 00000000\ 00000001 \text{ i.e.} \\ &2^{-126} \times 0.000000000000000000000001 \text{ (mixed notation.)} \\ &\text{which is } 2^{-126} \times 2^{-23} = 2^{-149} \end{aligned}$$

Attempts to store numbers smaller than this results in zero being stored – underflow error.

```
for(int i = 1;i<=24;i++)
{
```

```
x/=2;
```

```
cout<<x<<" " <<i<<endl;
```

```
}
```