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:

$$\overbrace{00000000}^{\text{byte 4}}\ \overbrace{00000000}^{\text{byte 3}}\ \overbrace{00000100}^{\text{byte 2}}\ \overbrace{00100011}^{\text{byte 1}}$$

and so n will be stored as:

$$\underbrace{11111111}_{\text{byte 4}}\ \underbrace{1111111}_{\text{byte 3}}\ \underbrace{11111011}_{\text{byte 2}}\ \underbrace{11011101}_{\text{byte 1}}$$

Since the leftmost bit is always 0 when a positive integer is stored it follows that the leftmost bit will always be 1 when a negative integer is stored. And so when this bit is 1 it indicates a negative number and twos complement storage.

The twos complement storage scheme gives correct results when negative integers are added to other negatives or to positive integers.

Example:

	Base Ten	Base Two	
	3	00000000000000000000000000000011	
+	-5	11111111111111111111111111111011	(twos complement)
	-2	11111111111111111111111111111110	

An attempt to store an integer greater than the largest possible one ($2^{31} - 1$) will produce what is known as *integer overflow*:

Example:

	Base Ten	Base Two
	2147483647	01111111111111111111111111111111
+	1	00000000000000000000000000000001
	2147483648	10000000000000000000000000000000

But 10000000000000000000000000000000 has a 1 in the 32nd place and so will be interpreted by C++ as a negative integer in twos complement form. When the rules of twos complement storage are applied by the computer 10000000000000000000000000000000 becomes -10000000000000000000000000000000. (That is every bit except the first non-zero bit is reversed - but the there is only one non-zero bit and so this is not changed.)
-10000000000000000000000000000000=-2147483648!! which is what will be reported by the computer.
That is, C++ will calculate that 2147483647+1 = -2147483648.

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:

$$\overbrace{********}^{\text{byte 4}}\ \overbrace{********}^{\text{byte 3}}\ \overbrace{********}^{\text{byte 2}}\ \overbrace{********}^{\text{byte 1}}$$

$$\underset{\text{sign bit}}{\underset{\downarrow}{*}}\qquad \underset{\text{8 bit exponent+127}}{\underset{\downarrow}{********}}\qquad \underset{\text{23-bit mantissa-1}}{\underset{\downarrow}{***********************}}$$

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 acutal 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	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 foating point format (mixed notation) is

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

which is $(0.10001111100000000000111)_2$ Then

$$(0.10001111100000000000111)_2 = 2^{-1}+2^{-5}+2^{-6}+2^{-7}+2^{-8}+2^{-9}+2^{-19}+2^{-20}+2^{-21}$$
$$= 0.560550212860107421875.$$

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

0111111011111111111111111111111 = 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:
00000000 00000000 00000000 00000001 i.e.
$2^{-126} \times 0.00000000000000000000001$ (mixed notation.)
which is $2^{-126} \times 2^{-23} = 2^{-149}$
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;

}
```