Pat O'Sullivan

Mh4718 Week 4

Week 4

0.0.0.1 Storage of ***float*** *type variables (contd.).*

Recall that **float** values are converted to normalised scientific notation and then stored in four bytes according to the scheme:

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

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

0.0.0.2 Largest and smallest 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. This is called *overflow error.*

The largest storable float is therefore:
01111111 01111111 11111111 11111111 i.e.
$2^{127} \times 1.11111111111111111111111$ (mixed notation.) which is
$2^{127}(1 + 2^{-1} + 2^{-2} + ...2^{-23}) = 2^{127} + 2^{126} + ...2^{104}$
$= 2^{128} - 2^{104} = 340282346638528859811704183484516925440.0$

The most negative number is then 11111111 01111111 11111111 11111111 which is just the negative of this number.

When all the exponent bits are 0 *the storage rules* change in order to enable the storage of smaller numbers (i.e. closer to 0.)

The last case in which normal rules apply at the lower end of the range is:
*0000000 10000000 00000000 00000000 i.e. $\pm 2^{-126}$.
Once the exponent reaches 00000000 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 is:
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. This is called *underflow error.*

The following program illustraes underflow error. Note that the program treats 2^{-150} as 0.

```
#include<iostream>
#include<cmath>
using namespace std;

void main()
{  //This program demonstrates underflow error
   int p =0;
   float y=1;
   do
   {
      p--;
      y=y/2;
      cout<<y<<" =2^"<<p<<endl;
    }while(y>0);

}
```

0.0.1 Round Off error

Round off error occurs when a number is within the storable range (i.e. not too big or not too small) but all the bits in the mantissa cannot be stored because the mantissa is too long.

This is obviously the case when a number has an infinite binary decimal representation e.g. 0.2 but it can also happen for a number which has a finite number of digits.

Example 0.1

(i) 0.2 in binary is an infinite "decimal":

$$0.001100110011................ = 1.1001100110011001100.... \times 2^{-3}$$

That is the mantissa is infinitely long 1.1001100110011001100.... and the exponent is 2^{-3}.
This gives a biased exponent = 124 but we can only retain 23 of the infinite digits after the decimal points.

(ii) The number $2^{24}+1$ has binary representation

$$1\overbrace{00000000000000000000000}^{\text{23 zeros}}1$$

In normalised scientific notation (mixed base) this is written as

$$1.\overbrace{00000000000000000000000}^{\text{23 zeros}}1 \times 2^{24}$$

The biased exponent is thus 127+24= 151 but the mantissa-1 is

$$0.\overbrace{00000000000000000000000}^{\text{23 zeros}}1$$

which has 24 decimal places and so is too long to be stored as a float.

If we simply dropped the mantissa digits which do not fit then there would be a consistent undervaluing of all positive floats and overvaluing of all negative floats. This could cause serious accumulation of errors. To avoid this, C++ employs rounding rules which are designed to have some chance of balancing out over many calculations. The rounding rules are as follows:

1. If the digit in the 24th decimal place is 0 (followed by some non-zero digits) then all remaining digits are dropped and no further action is taken. The stored number is then clearly too small.
2. If the digit in the 24th decimal plase is 1 (followed by some non-zero digits) then all remaining digits are dropped and a 1 is added in the 23rd decimal place. (If the 23rd decimal place is already 1 then there will, of course, be a knock on effect from carried 1's.) The stored number is then too big.

3. If the digit in the 24th decimal place is a solitary 1 i.e. followed by all zeros, then it is simply chopped if digit 23 is 0 but if digit 23 is 1 then a 1 is added into the 23rd decimal place.

Example 0.2

(i) Applying these rules to the storage of 0.2 we have:

$$\text{mantissa } = 1.\overbrace{10011001100110011001100}^{23 \text{ places}}11001100\ldots$$

which is rounded to

$$1.\overbrace{10011001100110011001101}^{23 \text{ places}}$$

And so what is stored in the 4 bytes is:

$$00111110\ 01001100\ 11001100\ 11001101$$

that is

$$\underset{\text{sign bit}}{\underset{\uparrow}{0}} \quad \overbrace{01111100}^{\text{biased exponent}} \ \overbrace{10011001100110011001101}^{\text{mantissa-1}}$$

that is

biased exponent $= 124$ and mantissa $- 1 = 0.10011001100110011001101$.

That is

$$\text{exponent } = -3, \text{ mantissa } = 1.10011001100110011001101.$$

Therefore that value that is actually stored (in mixed notation) is

$$2^{-3} \times 1.10011001100110011001101.$$

Converting this to base ten notation and grouping powers of 2 we get:

$$\begin{aligned} 2^{-3}\times 1.10011001100110011001101 &= \frac{1}{2^3}\left(\frac{3}{2}+\frac{3}{2^5}+\frac{3}{2^9}+\frac{3}{2^{13}}+\frac{3}{2^{17}}+\frac{3}{2^{21}}+\frac{1}{2^{23}}\right) \\ &= \frac{3}{2^4}+\frac{3}{2^8}+\frac{3}{2^{12}}+\frac{3}{2^{16}}+\frac{3}{2^{20}}+\frac{3}{2^{24}}+\frac{1}{2^{26}} \end{aligned}$$

The following program proves that the exact value for a **float** type variable assigned the value 0.2 is indeed

$$\frac{3}{2^4}+\frac{3}{2^8}+\frac{3}{2^{12}}+\frac{3}{2^{16}}+\frac{3}{2^{20}}+\frac{3}{2^{24}}+\frac{1}{2^{26}}$$

```
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;
void main()
{
    float x =0.2;
    float y=3*(pow(2.0,-4)+pow(2.0,-8)+pow(2.0,-12)+pow(2.0,-16)
                                    +pow(2.0,-20)+pow(2.0,-24))+pow(2.0,-26);
    cout<<setprecision(30);
    cout<<x<<endl;
    cout<<y<<endl;
    if(x==y)
    {
        cout<<"They are equal."<<endl;
    }
    else
    {
        cout<<"They are not equal."<<endl;
     }
}
```

The values stored under y and x in the above program are the same.

In order to get a more concise expression for the actual stored value $2^{-3} \times 1.10011001100110011001101$ we note that

$$2^{-3} \times 1.10011001100110011001101$$

$$= 0.2 - \overbrace{2^{-3} \times 0.0000000000000000000000011001100\ldots}^{\substack{\text{chopped} \\ \downarrow}} + \overset{\substack{\text{round up} \\ \downarrow}}{2^{-3}} \times \frac{1}{2^{23}}$$

(That is, the value stored is 0.2 less the value which was chopped when rounding, plus the value of the rounding-up digit.)

$$= 0.2 - 2^{-3} \times 0.\overbrace{00000000000000000000000}^{\text{23 zeros}}11001100\cdots + 2^{-3} \times \frac{1}{2^{23}}$$

$$= 0.2 - \frac{1}{2^3} \times \left(\frac{1}{2^{24}} + \frac{1}{2^{25}} + \frac{1}{2^{28}} + \frac{1}{2^{29}} + \ldots \right) + \frac{1}{2^{26}}$$

$$= 0.2 - \frac{1}{2^3} \times \left(\frac{3}{2^{25}} + \frac{3}{2^{29}} + \ldots \right) + \frac{1}{2^{26}}$$

$$= 0.2 - \frac{1}{2^3} \times \frac{\frac{3}{2^{25}}}{1 - \frac{1}{2^4}} + \frac{1}{2^{26}}$$
$$= 0.2\left(1 + \frac{1}{2^{26}}\right)$$

Therefore if we have the line

$$\textbf{float}\ x{=}0.2;$$

in a C++ program then the exact value assigned to x is not 0.2 but the slightly bigger $0.2\left(1 + \frac{1}{2^{26}}\right)$.
The difference between the true value and the stored value is therefore $0.2 \times \frac{1}{2^{26}} = \frac{1}{2^{26} \times 5}$. In base ten this is a decimal with 26 decimal places, $[25 \log_{10} 5] + 1 = 18$ significant digits and thus 8 leading zeros.

(ii) If we have the line

$$\textbf{float}\ x{=}\mathrm{pow}(2.0,24){+}1;$$

in a C++ program rounding rules are applied to the storage of $2^{24} + 1$ as follows:

$$2^{24} + 1 = 1\overbrace{00000000000000000000000}^{\text{23 zeros}}1.$$

In normalised scientific notation (mixed base) this is written as

$$1.\overbrace{00000000000000000000000}^{\text{23 zeros}}1 \times 2^{24}$$

The mantissa-1 is

$$0.\overbrace{00000000000000000000000}^{\text{23 zeros}}1$$

which has 24 decimal places and so is too long to be stored as a float. When rounding rules are applied we see that the only excess digit is a solitary 1 which is preceded by a 0 and so it is simply dropped. The value that is actually stored then is

$$1.\overbrace{00000000000000000000000}^{\text{23 zeros}} \times 2^{24} = 2^{24}.$$

Notice that the 1 is never added to 2^{24}.

(iii) If we have the line

$$\textbf{float } \mathrm{x{=}pow(2.0,24){+}3;}$$

in a C++ program, rounding rules are applied to the storage of $2^{24}+3$ as follows:

$$2^{24}+3 = 1\overbrace{00000000000000000000001}^{\text{23 zeros}}1.$$

In normalised scientific notation (mixed base) this is written as

$$1.\overbrace{00000000000000000000001}^{\text{23 zeros}}1 \times 2^{24}$$

The mantissa-1 is

$$0.\overbrace{00000000000000000000001}^{\text{23 zeros}}1$$

which has 24 decimal places and so is too long to be stored as a float. When rounding rules are applied we see that the only excess digit is a solitary 1 which is preceded by a 1 and so 1 is added to the 23rd bit of the mantissa and so the value that is actually stored then is

$$1.\overbrace{00000000000000000000010}^{\text{23 zeros}} \times 2^{24} = 2^{24}+4$$

which is too big.