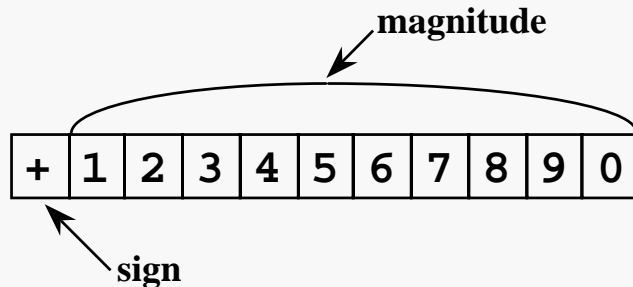
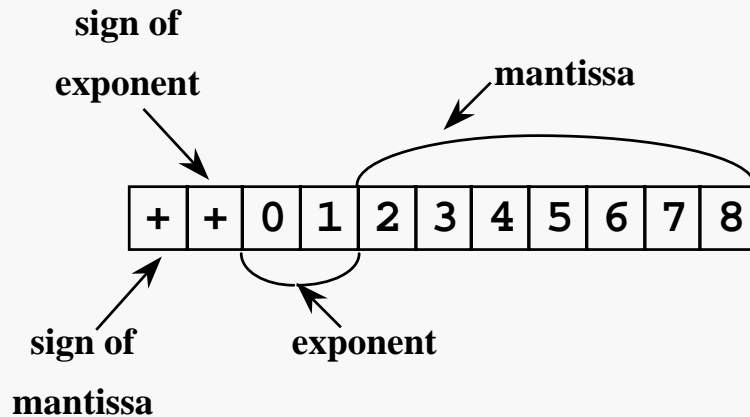


Integer



**On standard 32 bit machines: `INT_MAX = 2147483647` which gives 10 digits of precision, (i.e. the maximum number of significant digits).
Range is the interval from the smallest representable value to the largest representable value.**

Float



Real numbers are stored in a normalized format, (the first digit of the mantissa is nonzero).

Storage

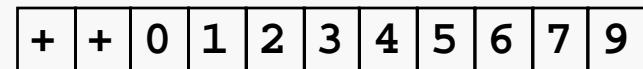
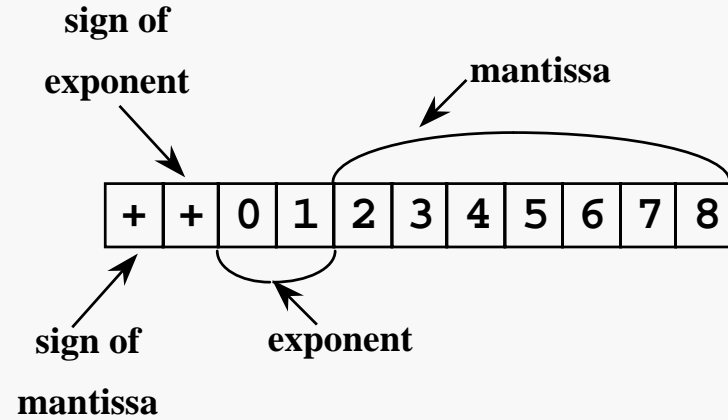
The representation to the right corresponds to the real number(s):

2.345678

2.3456789
2.34567891

The last two, not represented exactly due to limited precision, are examples of representational, (truncation), error.

Some machines round instead of truncate and would thus represent the last 2 numbers as shown at the right. The difference between the actual value and the rounded value introduces rounding error.



Operations

The addition, $0.9999999 \times 10^{99} + 1$, results in an overflow error, an attempt to represent a number larger than the maximum representable value.

Analogously, underflow is an attempt to represent a number smaller than the minimum representable value.

Integer overflow & underflow occurs when an expression evaluates to a value outside the range [INT_MIN ... INT_MAX].

The C++ reference manual states that the result of overflow is undefined, (i.e. compiler dependent).

The manual states that the result of underflow is also undefined, (compiler dependent).

Operations

Consider the simple problem of summing real numbers stored in a file one per line, (example given at right).

The result after the first three additions would be 3000000.0. The second & fourth numbers having no effect on the result due to limited precision & the large differences in the numbers, termed cancellation error.

```
1000000.0
0.000123
2000000.0
0.000456
.
.
.
```

In the worst case, if the entire file was organized in this manner only half of the numbers would be added. Solution: sort the numbers & add the smallest first so they may affect the total.

Comparisons

Due to representational errors, (and conversion errors), float values should never be compared for equality.

Looping

For the same reasons float variables should not be used for loop control.

Float Equality Check

`abs (real1 - real2) < epsilon`

where `epsilon = 0.00001` an appropriate value near the precision of the machine.

Dangerous

```
// loop from 0.0 to 1.0 in increments of 0.1
float realVar = 0.0 ;
while ( realVar < 1.0 ) {
    // ...
    realVar = realVar + 0.1 ;
}
```

Better

```
// loop from 0.0 to 1.0 in increments of 0.1
i = 0 ;
while ( i < 10 ) {
    realVar = i / 10.0 ;
    // ...
    i = i + 1 ;
}
```

The short program below illustrates the perils of direct comparison of float values. Logically the variable X should take on values 0.000, 0.001, 0.002, . . . , 0.999, 1.000. However, when executed on a Pentium II cpu, the program produces output as shown below.

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

void main() {

    float X      = 0.0f;
    float DeltaX = 0.001f;
    int Counter = 0;

    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);

    while (X < 1.0) {
        cout << setw( 4) << Counter
             << setw(10) << setprecision(5) << X << endl;
        X = X + DeltaX;
        Counter++;
    }
    cout << setw( 4) << Counter
         << setw(10) << setprecision(5) << X << endl;

}
```

```
993  0.99299
994  0.99399
995  0.99499
996  0.99599
997  0.99699
998  0.99799
999  0.99899
1000 0.99999
1001 1.00099
```

**An infinite loop would
result if the condition
were changed to**

X != 1.0

The short program below illustrates a float variable underflowing to zero. Logically the while loop should be infinite. However, when executed on a Pentium II cpu, the program produces output as shown below.

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

void main() {

    const float One = 1.0f;
    float X = 1.0f;
    int Counter = 1;

    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);

    while (X != 0.0f) {
        cout << setw( 3) << Counter
            << setw(15) << setprecision(6) << X << endl;
        X = X / 2.0f;
        Counter++;
    }
}
```

1	1.000000
2	0.500000
3	0.250000
4	0.125000
5	0.062500
6	0.031250
7	0.015625
8	0.007813
9	0.003906
10	0.001953
11	0.000977
12	0.000488
13	0.000244
14	0.000122
15	0.000061
16	0.000031
17	0.000015
18	0.000008
19	0.000004
20	0.000002
21	0.000001
22	0.000000
.	.
149	0.000000
150	0.000000

The short program below illustrates a float variable underflowing to zero. Logically the while loop should be infinite. However, when executed on a Pentium II cpu, the program produces output as shown below.

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

void main() {

    const float One = 1.0f;
    float X = 1.0f;
    int Counter = 1;

    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);

    while (One + X != One) {
        cout << setw( 3) << Counter
            << setw(15) << setprecision(6) << X << endl;
        X = X / 2.0f;
        Counter++;
    }
}
```

1	1.000000
2	0.500000
3	0.250000
4	0.125000
5	0.062500
6	0.031250
7	0.015625
8	0.007813
9	0.003906
10	0.001953
11	0.000977
12	0.000488
13	0.000244
14	0.000122
15	0.000061
16	0.000031
17	0.000015
18	0.000008
19	0.000004
20	0.000002
21	0.000001
22	0.000000
...	...
52	0.000000
53	0.000000