Chapter - 19
Floating Point

Floating Point Format

`±` Is the sign (plus or minus).
`f.fff` Is the 4 digit fraction.
`±e` Is the single-digit exponent.
Zero is 0.0
We represent these numbers in "E" format: ±*f.fff*E±*e*.

Examples:

Notation	**Number**
`+1.000E+ 0`	`1.0`
`+3.300E+ 5`	`330000`
`-8.223E-3`	`-0.01`
`+0.000E+ 0`	`0.0`

Floating Point Add/Sub

1. Start with the numbers:

   +2.000E+0 The number is 2.0
   +3.000E-1 The number is 0.3

2. Add guard digits to both numbers

   +2.0000E+0 The number is 2.0
   +3.0000E-1 The number is 0.3

3. Shift the number with the smallest exponent to the right one digit and incre-ment its exponent. Continue until the exponents of the two numbers match.

   +2.0000E+0 The number is 2.0
   +0.3000E-0 The number is 0.3

4. Add the two fractions. The result has the same exponent as the two numbers.

   +2.0000E+0 The number is 2.0
   +0.3000E-0 The number is 0.3
   +2.3000E+0 Result 2.3

Floating Point Add/Sub

5. Normalize the number by shifting it left or right until there is just one non-zero digit to the left of the decimal point. Adjust the exponent accordingly. A number like +0.1234E+0 would be normalized to +1.2340E-1. Because the number +2.3000E+0 is already normalized we, do nothing.
6. Finally, if the guard digit is greater than or equal to 5, round the next digit up; otherwise truncate the number.

   +2.3000E+0 Round last digit

   +2.300E+0 Result 2.3
7. For floating-point subtraction, change the sign of the second operand and add.

Multiplication

1. Add the guard digit:

   +1.2000E-1 The number is 0.12
   +1.1000E+1 The number is 11.0

2. Multiply the two fractions and add the exponents.
   (1.2 * 1.1 = 1.32) (-1 + 1 = 0)

   +1.2000E-1 The number is 0.12
   +1.1000E+1 The number is 11.0
   +1.3200E+0 The result is 1.32

3. Normalize the result. If the guard digit is less than or equal to 5, round the next digit up. Otherwise, truncate the number.

   +1.3200E+0 The number is 1.32

Division

1. Add the guard digit:

   +1.0000E+2 The number is 100.0
   +3.0000E+1 The number is 30.0

2. Divide the fractions, subtract the exponents:

   +1.0000E+2 The number is 100.0
   +3.0000E+1 The number is 30.0
   +0.3333E+1 The result is 3.333

3. Normalize the result:

   +3.3330E+0 The result is 3.333

4. If the guard digit is less than or equal to 5, round the next digit up. Other-wise, truncate the number:

   +3.333E+0 The result is 3.333

Overflow and Underflow

`9.000E+9 × 9.000E+9`

is:

8.1×10^{19}

That too big for our representation (overflow).

`1.000E-9 × 1.000E-9`

is

1.0×10^{-18}

That's to small (underflow).

Roundoff Error

1/3 + 1/3 != 2/3

2/3 as floating-point is 6.667E-1
1/3 as floating-point is 3.3333-1
    +3.333E-1
    +3.333E-1
    +6.666E-1 or 0.6666
which is not:
    +6.667E-1

Accuracy

```
1 - 1/3 - 1/3 - 1/3
```

```
 1.000E+0
```

- 3.333E-1
- 3.333E-1
- 3.333E-1

or:

1.000E+0
- 3.333E-1
- 3.333E-1
- 3.333E-1

0.0010E+0 or 1.000E-3

Minimizing error:

- Use double instead of float
- Other techniques are beyond the scope of this course.

Determining Accuracy

```
#include <iostream>
#include <iomanip.h>
int main(){
    // two number to work with
    float number1, number2;
    float result;          // result of calculation
    int   counter;         // loop counter and accuracy check

    number1 = 1.0;
    number2 = 1.0;
    counter = 0;

    while (number1 + number2 != number1) {
        ++counter;
        number2 = number2 / 10.0;
    }
    std::cout <<setw(2)<<counter<<
           " digits accuracy in calculations\n";

    number2 = 1.0;
    counter = 0;

    while (1) {
        result = number1 + number2;
        if (result == number1)
            break;
        ++counter;
        number2 = number2 / 10.0;
    }
    std::cout <<setw(2) <<counter <<
           " digits accuracy in storage\n";
    return (0);
}
```

Precision and Speed

Some older compilers do everything in double.

```
float answer, number1, number2;

answer = number1 + number2;
```

C++ must perform the following steps:

1) Convert number1 from single to double precision.
2) Convert number2 from single to double precision.
3) Double precision add.
4) Convert result into single precision and store in `answer`.

If the variables were of type **double**, C++ would only have to perform the steps:

1) Double precision add.
2) Store result in `answer`.

Power Series

$$\sin(x) = 1 + x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

$\sin(\pi/2)$

	Term	Value	Total
1	x	1.571E+ 0	
2	$x^3/3!$	6.462E-1	9.248E-1
3	$x^5/5!$	7.974E-2	1.005E+ 0
4	$x^7/7!$	4.686E-3	9.998E-1
5	$x^9/9!$	1.606E-4	1.000E+ 0
6	$x^{11}/11!$	3.604E-6	1.000E+ 0

Sin(pi)

	Term	Value	Total
1	x	3.142E+ 0	
2	x^3/3!	5.170E+ 0	-2.028E+0
3	x^5/5!	2.552E-0	5.241E-1
4	x^7/7!	5.998E-1	-7.570E-2
5	x^9/9!	8.224E-2	6.542E-3
6	x^{11}/11!	7.381E-3	-8.388E-4
7	$x1^3$/13!	4.671E-4	-3.717E-4
8	x^{15}/15!	2.196E-5	-3.937E-4
9	x^{17}/17!	7.970E-7	-3.929E-4
10	x^{19}/19!	2.300E-8	-3.929E-4