- **Number of decimal places of precision has reduced!**

  We supplied a value correct to 12 decimal places

- **The output is correct only to 6 decimal places, and the sixth digit is wrong.**

- **How did this happen?**

Floats: Mantissa-exponent representation

- **Any float (real number) can be written as**

  $r = s \times M \times B^{e-E}$

  s = sign

  M = mantissa

  B = base

  e = exponent

  E = bias

- **The choice of mantissa and exponent is not unique.**
- **If r = 0.234, r can be written as**

  0.234
  0.000234 $\times 10^{3}$
  23.4×10^{-2}

  etc.

Floats: IEEE representation

- **In computers, usually B = 2.**
- **For B = 2, the mantissa is normalised if it is written in the form**

  $1.f$
  where $0 \le f < 1$

- **The exponent, too, is represented in binary.**
- **A float corresponds to 4 bytes = 32 bits. Thus, the declaration**

```
float radius;
```

- **reserves 32 bits of space in the computer memory.**
- **According to the IEEE floating point standard 754 of 1985 these bits are distributed as follows.**

```
s   eeeeeeee  fffffffffffffffffffffff
(1)   (8)       (23)
```


Floats: IEEE representation

- **Using these 32 bits, the floating point number is recovered as follows,**
  - **according to the IEEE specification 754 of 1985.**
- **(a) Treat the sign bit as an integer** s **such that** $0 \le s \le 1$**.**
- **(b) Treat the 8 exponent bits as the bits of the binary representation of an integer** e **such that** $0 \le e \le 255$
  - **Thus, if the bits are designated by** e_i**, then**

$$e = \sum_{i=0}^{7} e_i \; 2^i$$

- (c) Treat the 23 fraction bits as the bits in the binary representation of a fraction, f, such that $0 \le f < 1$.

  – Thus, if these bits are designated by f_i then

$$f = \sum_{i=1}^{23} f_i \; 2^{-i}$$

- (d) The bias is 127.

- (e) The number V represented by these 32 bits is now

$$V = (-1)^s \times 2^{e-127} \times 1.f$$

  – where $1.f$ is the number obtained by prefixing f with an implicit leading 1, and is the same as $(1+f)$.

Other cases

- The above rule applies only for the case, where
- (1) $0<e<255$ and $f \neq 0$.
- The remaining special cases are as follows.
- (2) If $e=255$, $f=0$, $s=0$, then $V = INF$
- (3) If $e=255$, $f=0$, $s=1$, then $V = -INF$
- (4) If $e=255$, $f \neq 0$, then $V = NaN$ (Not a number)
- (5) If $e=0$, $f=0$, $s=0$, then $V = 0$ (zero).

- **(6) e=0, f=0, s=1, then $V = -0$.**
- **(7) If e=0, $f \neq 0$, then $V = (-1)^s \times 2^{e-126} \times 0.f$**
  - **(non-normalised number), where $0.f$ is the same as f.**

Examples

```
0 00000000 00000000000000000000000 = 0
1 00000000 00000000000000000000000 = −0

0 11111111 00000000000000000000000 = INF
1 11111111 00000000000000000000000 = −INF

0 11111111 00010000000000000000000 = NaN
1 11111111 00100000000000100000000 = NaN

0 01111111 10000000000000000000000 =
```
$(-1)^0 \times 2^{127-127} \times 1.1 = 1+½= 1.5$
```
1 10000000 00000000000000000000000 =
```
$(-1)^1 \times 2^{128-127} \times 1.0 = -2$

- **Conversely, to convert a float, such as 2.5, to a bit pattern, according to IEEE specifications, we proceed as follows.**

- **First convert the number into its binary representation.**
  $\mathbf{2.5 \equiv 10.1 = 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1}}$

- **Next, convert this to the mantissa-exponent form with a normalized mantissa.**
  $\mathbf{10.1 = 1.01 \times 2^1}$

- **Next, rewrite this as**
  $\mathbf{(-1)^s \times 2^{e-127} \times 1.f}$

- **Comparing the two expressions, we see that**
  $\mathbf{s = 0,\ e = 128,\ f = 0.1}$

- **and the corresponding bit patterns are**

  s = 0

  e = 1000 0000

  f = 1000 0000 0000 0000 0000 000,

  so that

  2.5 = 0 1000 0000 1000 0000 0000 0000 0000 000

Maximum and minimum floating point values

- **The maximum floating point value is thus,**

```
0 1111 1110 1111 1111 1111 1111 1111 111

                         = MAXFLOAT
```

- **The above bit pattern can be written more compactly, using hex as**

```
7F7FFFFF
```

- **To what number does this correspond?**

- **We see that**

  s = 0

  e = 254

$$f = \frac{1}{2} + \frac{1}{2^2} + \ldots + \frac{1}{2^{23}}$$
$$= \tfrac{1}{2} \cdot \frac{1-2^{-23}}{\tfrac{1}{2}} = 1-2^{-23}$$

- **Thus, MAXFLOAT = $2^{127} \times (1+1-2^{-23})$**

- We can now calculate the value of MAXFLOAT using logarithms.

- If

  $\log_{10} 2 = x$

  then

  $2 = 10^{x}$

- Hence,

  $2^{y} = (10^{x})^{y} = 10^{xy}$

- Putting in the values

  $x = \log_{10} 2 = 0.3010$

- For $y = 127$, we have $xy = 38.227$

  Thus, $2^{127} = 10^{38.227}$

- Similarly, for $y = -23$, we have $xy = -6.923$

  Thus, $2^{-23} = 10^{-6.923}$

- Thus, $f = 1 - 2^{-23} \approx 1 - 10^{-7} = 0.9999999$

- **Putting it all together,**

$$\text{MAXFLOAT} = 10^{38} \times 10^{0.227} \times 1.9999999$$

$$= 10^{38} \times 1.687 \times 1.9999999$$

$$= 3.37 \times 10^{38}$$

- **We can check this out with a small program.**

Floats: maximum and minimum

Program 5

```
/*Program name: MaxMin.c
Function: To show the maximum and minimum
floating point values */

#include <stdio.h>
#include <conio.h>
#include <values.h>
main()
{
  float a, b;
  a =  MAXFLOAT;
  b =  MINFLOAT;
  system (“cls”);
```

```
  printf ("\nMaximum floating point value ="
                            "%e", a);
  printf ("\n\n\nMinimum floating point value"
                            "= %e", b);
  getch();
  return 0;
}
```

- **Output:**

```
Maximum floating point value = 3.37000E+38

Minimum floating point value = 8.43000E-37
```


Minimum normal floating point value

- **For the minimum floating point value, the above program gives the output**
  - **MINFLOAT = 8.43E-37**
- **According to the IEEE standard, the min floating point value corresponds to the bit pattern**

0 00000001 00000000000000000000000

$= 2^{-126} = 10^{-37.926} =$
$10^{-37.926} = 10^{-38} \cdot 10^{0.074} \approx 1.18 \times 10^{-38}$

The discrepancy

- **Thus, the program output is:**

  MINFLOAT = 8.43E-37

- **The calculation, using the IEEE standard is**

  MINFLOAT = 1.18E-38

- **8.4E-37 ≠ 1.18E-38**

- **$8.43 \times 10^{-37} \neq 1.18 \times 10^{-38}$**

- **Q. Can you explain the discrepancy?**

A Turbo C Bug

- **$10^{-37} \cdot 10^{0.926} = 8.43E{-}37$??**

- **This is the value given in TURBO C. But the preceding step involves a mistake. The correct value would be, as we calculated.**

  $10^{-37.926} = 10^{-38} \cdot 10^{0.074} \approx 1.18 \times 10^{-38}$

- **The exact value is 1.17549421E-38, on UNIX systems.**

Moral

- **The machine is NOT always right!**
- **All programs have bugs.**
- **IDE's are programs.**
- **Hence, the Turbo C IDE also has bugs.**
- **Don't trust it blindly!**

- **Good programming requires a clear understanding of what is going on.**

A doubt

- While calculating MINFLOAT we used the bit pattern

```
MINFLOAT =
0 00000001 00000000000000000000000
```

- But clearly, the following bit pattern

```
0 00000000 00000000000000000000001
```

  - corresponds to a smaller number

- Is something wrong here?

Smaller than the smallest is not normal

- Recollect the IEEE specification Rule (7).
- (7) If e=0, f≠0, then $V = (-1)^s \times 2^{e-126} \times 0.f$
  - (non-normalised number), where $0.f$ is the same as f.
- MINFLOAT is NOT the smallest floating point value.
- MINFLOAT is only the smallest NORMAL floating point value.

- **Smaller values CAN be represented, but they are called subnormal, or non-normal.**

Minimum subnormal floating point value

- **Minimum subnormal float is**

  0 00000000 00000000000000000000001

  $= 1 \cdot 2^{-126} \cdot 0.00000000000000000000001$

  $= 1 \cdot 2^{-126} \cdot 2^{-23}$

  $= 2^{-149} \approx 1.415 \times 10^{-45}$

- **This is smallest value for any float in a C program.**

Summary

- **MAXFLOAT ≈3.37E38**
  - **numbers larger than MAXFLOAT correspond to INF.**
  - **Negative numbers smaller than − MAXFLOAT correspond to −INF**
- **MINFLOAT ≈ 1.18E-38**
  - **Positive numbers smaller than MINFLOAT actually CAN be represented in a C program.**
  - **But these numbers are called subnormal**

- **The minimum subnormal number ≈ 1.4E-45.**

Checking it out

- **What happens if we use floats outside this range?**
- **We can check this out with a small program.**

```
/*Program name: infinity.c
Function: To check what happens when we use
numbers outside the range of MAXFLOAT and
MINFLOAT*/
#include <stdio.h>
#include <conio.h>
#include <values.h>
main()
{
  float a, b, c;
```



```
  a =  MAXFLOAT;
  b =  MINFLOAT;
  printf ("\nMax = %e, \n Min= %e", a, b);
  getch();
/*Now try putting in values of a, and b,
larger than  MAXFLOAT or values of b smaller
than MINFLOAT */
  printf ("\n\n Enter a = ");
  scanf( "%f", &a);
  printf ("a = %e", a);
  printf ("\n Enter b = ");
  scanf ("%f", &b);
  printf ("\n b = %e \n", b);
  c = a/b;
  printf ("%e/%e = %e", a, b, c);
  getch(); return 0; }
```

Significant figures

- **If the float data type can be used to represent numbers as small as 10^{-38} or 10^{-45}, then why can't the computer print the value of π correct to 45 (or 38) decimal places?**

- 10^{-45} **=**

  0.000000000000000000000000000000000000000000001

  (44 zeros after the decimal point)

- **NOTE: The number π is NOT $\frac{22}{7}$ or 3.14.**

- $\pi=$
  3.141592658979323846243383279502884197169399

  (accurate to 45 decimal places)

Solving the puzzle

- **Smallest representable number depends upon the** *exponent***,**

  BUT

- **accuracy of a calculation depends upon the** *mantissa***.**

Decimal places of precision

- **According to the IEEE specifications, 23 bits are available to represent the mantissa.**

- **To how many decimal places does this correspond?**

Converting bits to decimals

- **We can convert bits to decimal places, as before, using** $\log_{10}2 = 0.3010$
- **Thus, 23 bits corresponds to**

  $23 \times 0.3010 = 6.923$

  or about 7 decimal places of precision.
- **Thus,** $1.023 \times 10^{-7} = 1.0 \times 10^{-7}$ **on a computer.**
- **Conclusion:** *The simplest floating point calculation using C on a computer cannot be accurate to more than 6 or 7 decimal places.*

Rounding

- **Without going into the finer points here, we can see that**
- **the error INCREASES with each operation with floats.**
  - **such as addition or multiplication**
- **Hence, in the Area.c program the value of** π **was rounded off to 5 decimal places.**

Understanding the solution

- **How does the computer add two numbers with a different exponents?**
- **It first makes the two exponents equal: the exponent of the smaller number is made equal to that of the larger number.**
- **In the process the mantissa must be bit shifted.**

Significant figures (contd)

- **Thus, to get**
  $1+\varepsilon$
  where $\varepsilon=1\times10^{-6}$**,**

  the computer first represents both to the same exponent,
  $1=1\times10^{0}$
  $\varepsilon=0.000001\times10^{0}$**,**

  and then adds the mantissae

  $1+\varepsilon=1.000001\times10^{0}$

Bit shifting

- That is,
  - Step 1: Make the two exponents equal.
  - Step 2: Adjust the mantissa of the smaller number.

- (Naturally, the preference is to adjust the smaller number.)

- In binary representation, the mantissa is adjusted by bit shifting.

Example

- To perform 1.5 + 64
  - Above numbers are in decimal representation.

- In binary representation
  $1.5 \equiv 1.1\times 2^0$
  $64 \equiv 1.0\times 2^6$

- Thus, the number 1.5 must be adjusted, and written as
  $1.1 \times 2^0 \equiv 0.0000011 \times 2^6$

- The original mantissa corresponded to the bit pattern
  f = 1000 0000 0000 0000 0000 000

- **After adjusting the exponent, the new mantissa corresponds to the bit pattern**

  f = 0000 0110 0000 0000 0000 000

- **The bits in the mantissa have been shifted to the right.**

- **Q. What happens if the original mantissa was**

  f = 1000 0000 0000 0000 0000 001

- **A. The tail end bit disappears when the mantissa is bit shifted.**

- **There are only 23 bits in the mantissa.**
- **Q. What happens if we have to shift by more than 23 bits?**

- A. If we have to shift the mantissa of a float by more than 23 bits, the entire mantissa disappears!

Summary

- When two numbers with unequal exponents are added, the mantissa of the smaller number is bit shifted to the right.
- In this process, the tail bits of the mantissa disappear.
- Since a float has only 23 bits for the mantissa, if the mantissa has to be shifted by more than 23 bits, the entire mantissa disappears.
- 23 bits corresponds to about 7 decimal places.

- **Conclusion:** *if the exponent of two floats differs by more than 7 decimal places, then adding the two floats gives the larger float.*

- **That is, if**

$$\varepsilon = \mathbf{10}^{-7}$$

- **then**

$$\mathbf{1} + \varepsilon = \mathbf{1}$$

- **That is, relatively insignificant quantities are discarded or "zeroed" in the process of a calculation.**

Historical note

- **As stated earlier, the term "algorithm" comes from the Latin name Algorismus of Al Khwarizmi,**
  - **a 9th c. Arab scholar who translated the works of Brahmagupta etc.**

- **These arithmetic techniques were imported into Europe, beginning with Pope Sylvester from the 10th. century.**

- **These algorismus techniques competed with the abacus techniques in Europe for FIVE centuries.**