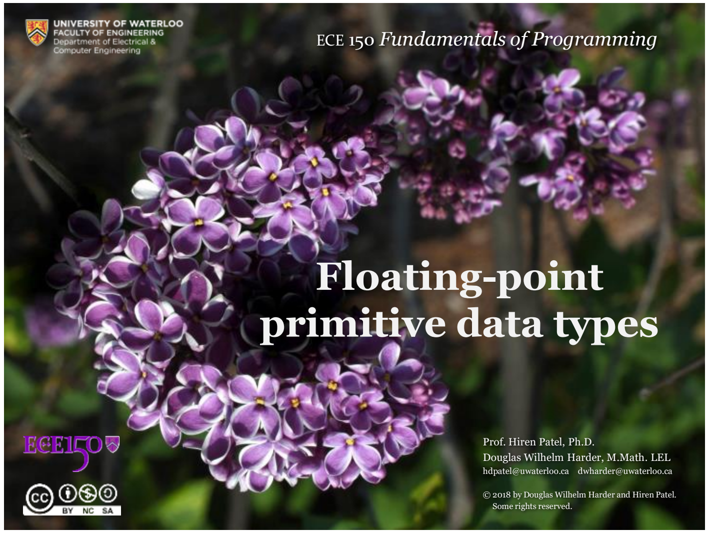
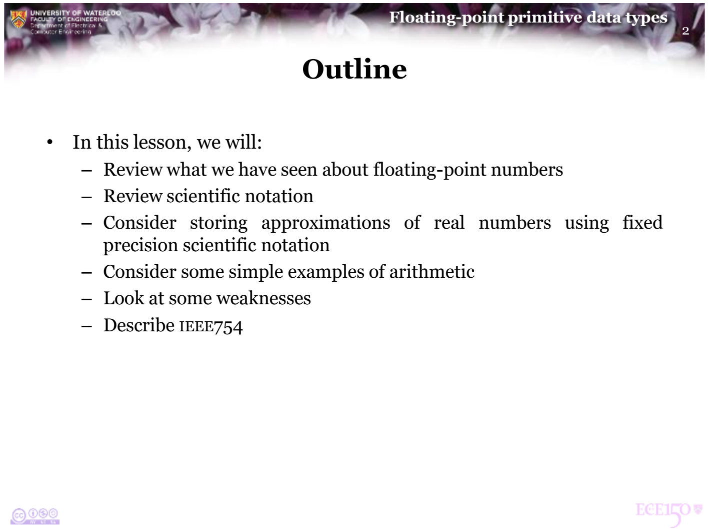


UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

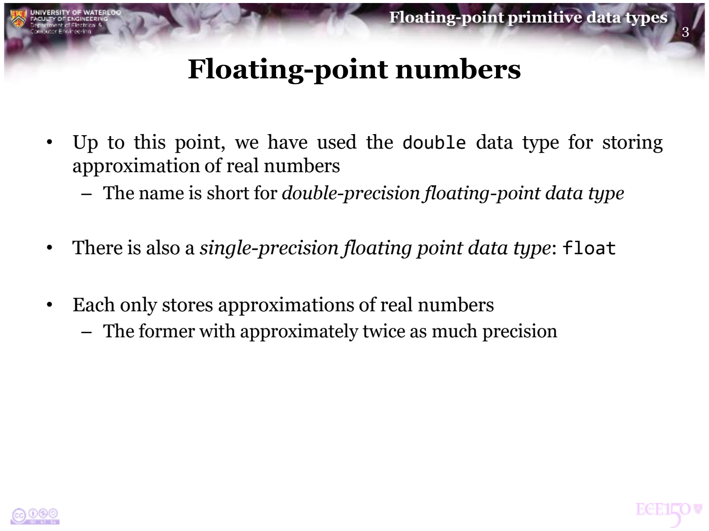
Floating-point primitive data types

Prof. Hiren Patel, Ph.D.
Douglas Wilhelm Harder, M.Math. LEL
hdpatel@uwaterloo.ca dwharder@uwaterloo.ca



Outline

- In this lesson, we will:
  - Review what we have seen about floating-point numbers
  - Review scientific notation
  - Consider storing approximations of real numbers using fixed precision scientific notation
  - Consider some simple examples of arithmetic
  - Look at some weaknesses
  - Describe IEEE754

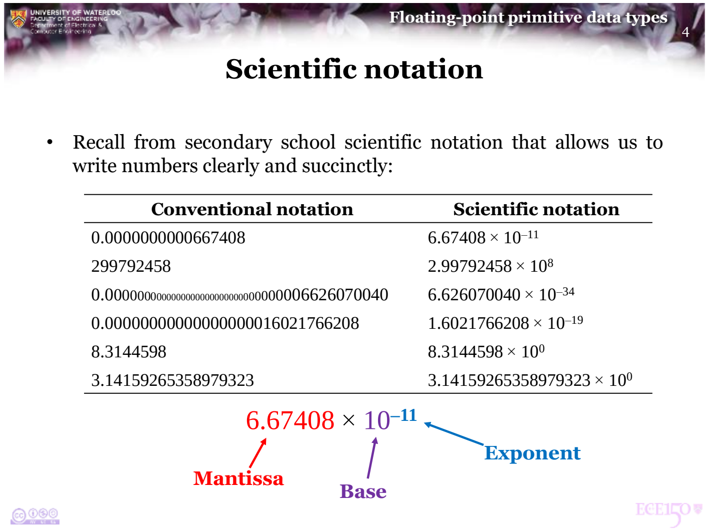


Floating-point numbers

- Up to this point, we have used the `double` data type for storing approximation of real numbers
  - The name is short for *double-precision floating-point data type*

- There is also a *single-precision floating point data type*: `float`

- Each only stores approximations of real numbers
  - The former with approximately twice as much precision

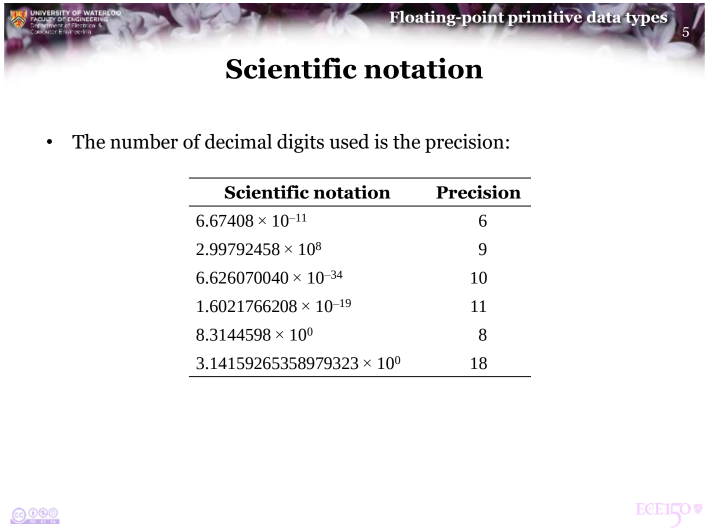


Scientific notation

- Recall from secondary school scientific notation that allows us to write numbers clearly and succinctly:

Conventional notation	Scientific notation
0.0000000000667408	6.67408×10^{-11}
299792458	2.99792458×10^{8}
0.0000000000000000000000000000000006626070040	$6.626070040 \times 10^{-34}$
0.00000000000000000016021766208	$1.6021766208 \times 10^{-19}$
8.3144598	8.3144598×10^{0}
3.14159265358979323	$3.14159265358979323 \times 10^{0}$

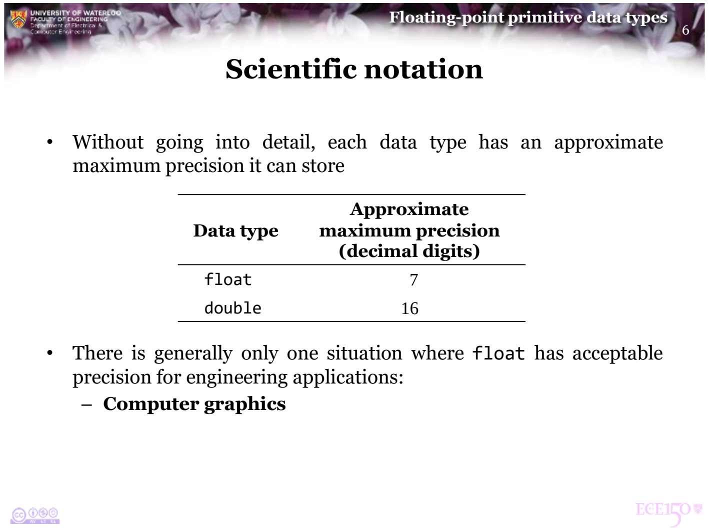
$$\underbrace{6.67408}_{\text{Mantissa}} \times \underbrace{10}_{\text{Base}}^{\overset{\text{Exponent}}{-11}}$$



Scientific notation

- The number of decimal digits used is the precision:

Scientific notation	Precision
6.67408 × 10^{-11}	6
2.99792458 × 10^{8}	9
6.626070040 × 10^{-34}	10
1.6021766208 × 10^{-19}	11
8.3144598 × 10^{0}	8
3.14159265358979323 × 10^{0}	18

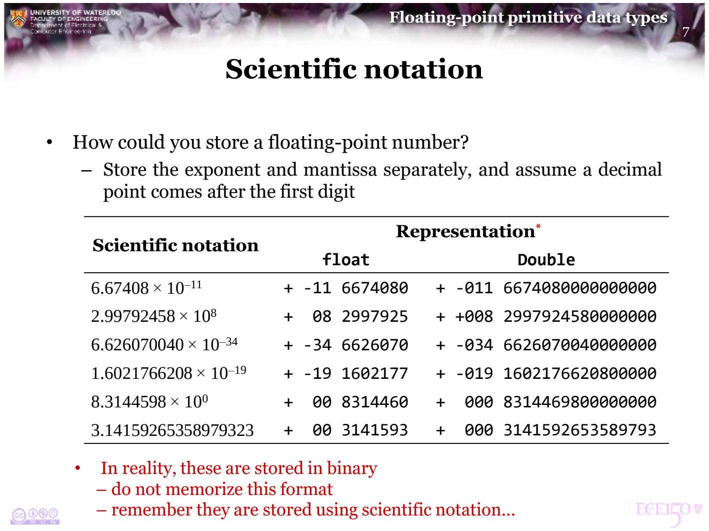


Scientific notation

- Without going into detail, each data type has an approximate maximum precision it can store

Data type	Approximate maximum precision (decimal digits)
`float`	7
`double`	16

- There is generally only one situation where `float` has acceptable precision for engineering applications:
  - **Computer graphics**

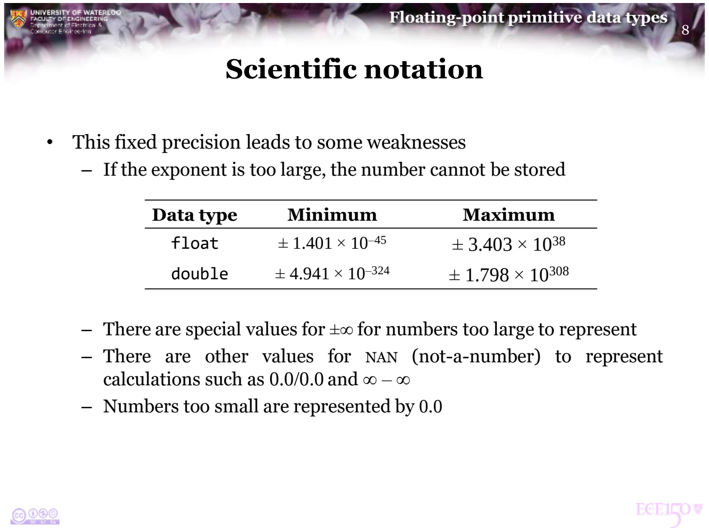


Scientific notation

- How could you store a floating-point number?
  - Store the exponent and mantissa separately, and assume a decimal point comes after the first digit

Scientific notation	Representation*	
	`float`	`Double`
6.67408 × 10^{-11}	`+ -11 6674080`	`+ -011 6674080000000000`
2.99792458 × 10^{8}	`+  08 2997925`	`+ +008 2997924580000000`
6.626070040 × 10^{-34}	`+ -34 6626070`	`+ -034 6626070040000000`
1.6021766208 × 10^{-19}	`+ -19 1602177`	`+ -019 1602176620800000`
8.3144598 × 10^{0}	`+  00 8314460`	`+  000 8314469800000000`
3.14159265358979323	`+  00 3141593`	`+  000 3141592653589793`

- In reality, these are stored in binary
  – do not memorize this format
  – remember they are stored using scientific notation…



Scientific notation

- This fixed precision leads to some weaknesses
  - If the exponent is too large, the number cannot be stored

Data type	Minimum	Maximum
`float`	± 1.401 × 10^{-45}	± 3.403 × 10^{38}
`double`	± 4.941 × 10^{-324}	± 1.798 × 10^{308}

  - There are special values for ±∞ for numbers too large to represent
  - There are other values for NAN (not-a-number) to represent calculations such as 0.0/0.0 and ∞ – ∞
  - Numbers too small are represented by 0.0

Weaknesses

- This fixed precision leads to some weaknesses
  - It can happen that $x + y = x$ even if $y \neq 0$
  - The calculation $x - y$ can be problematic if $x \approx y$
  - Even associativity is lost: sometimes $x + (y + z) \neq (x + y) + z$

- We will look quickly at these
  - In your courses on numerical analysis you will learn how to mitigate or avoid these weaknesses

Weakness: $x + y = x$ even if $y \neq 0$

- Non-zero numbers act like zero
  - Suppose we add these two numbers:

```
+ +000 3141592653589793
+ -019 5749522264293560
```

π

e^{-42}

  - Calculating this:

$$\begin{array}{r} 3.141592653589793 \\ +\ \underline{0.0000000000000000000574952226429356 0} \\ 3.1415926535897930005749522264293560 \end{array}$$

  - The representation of this sum is

```
+000 3141592653589793
```

  - There is no difference…

Weakness: $x + y = x$ even if $y \neq 0$

- For example:

```
           Pi + 1e-10 = 3.1415926536897931
    (Pi + 1e-10) - Pi = 1.000000082740371e-10

           Pi + 1e-16 = 3.1415926535897931
    (Pi + 1e-16) - Pi = 0
```

```
#include <iostream>
#include <cmath>

int main();

int main() {
        std::cout.precision( 17 );  // Print floating-point numbers to 17 digits of precision

        double x{std::acos(-1.0)};
        double y{1e-10};
        double z{x + y};

        std::cout << "       Pi + 1e-10 = " <<    z    << std::endl;
        std::cout << "(Pi + 1e-10) - Pi = " << (z - x) << std::endl;
        std::cout << std::endl;

        y = 1e-16;
        z = x + y;

        std::cout << "       Pi + 1e-16 = " <<    z    << std::endl;
        std::cout << "(Pi + 1e-16) - Pi = " << (z - x) << std::endl;

        return 0;
}
```

Weakness: subtractive cancellation

- Subtraction results in a loss of precision
  - Suppose we subtract these two numbers:

```
+ -001 8414709848079505
+ -001 8414709848078965
```

$\sin(1.0000000000001)$

$\sin(1)$

  - Calculating this:

$$\begin{array}{r} 0.8414709848079505 \\ -\ \underline{0.8414709848078965} \\ 0.0000000000000540 \end{array}$$

  - The representation of this sum is

```
+ -014 5400000000000000
```

  - The correct answer is $5.403023058680976 \times 10^{-14}$

Weakness: subtractive cancellation

- We can define the derivative of sin(*x*) at *x* = 1:

$$\frac{\mathrm{d}}{\mathrm{d}x}\sin(1)=\lim_{h\to 0}\frac{\sin(1+h)-\sin(1-h)}{2h}$$

- Thus, in theory, as *h* gets smaller and smaller,

$$\frac{\sin(1+h)-\sin(1-h)}{2h}$$

should be a better and better approximation

Weakness: subtractive cancellation

- Let's try this in C++:

$$\frac{\sin(1+h)-\sin(1-h)}{2h}$$

```
#include <iostream>
#include <cmath>

int main();

int main() {
        std::cout.precision( 17 );
        double h{1e-10};

        std::cout << "Calculating the derivative of sin(x) at x = 1:" << std::endl;
        std::cout << "       cos(1) = " << std::cos( 1.0 ) << std::endl;

        double dsin1{(std::sin( 1.0 + h ) - std::sin( 1.0 - h ))/(2*h)};
        std::cout << "When h = " << h << ", " << dsin1 << std::endl;

        h = 1e-15;
        dsin1 = (std::sin( 1.0 + h ) - std::sin( 1.0 - h ))/(2*h);
        std::cout << "When h = " << h << ", " << dsin1 << std::endl;

        return 0;
}
```

```
Calculating the derivative of sin(x) at x = 1:
       cos(1) = 0.54030230586813977
When h = 1e-10, 0.54030224738710331
When h = 1.0000000000000001e-15, 0.55511151231257827
```

IEEE 754-2008

- Originally written in 1985, this document specifies the representations of both `float` and `double`

- Whether you use C++, FORTRAN, Python, or MATLAB, your calculations will result in exactly the same result
  - Only the quality of your algorithms will affect your outcomes
  - Java is not IEEE 754 compliant… ☹

    Kahan and Darcy, *How Java's Floating-Point Hurts Everyone Everywhere*

- Some computers internally store approximately 20 decimal digits of precision for intermediate calculations
  - As soon as the number is written to main memory, only 16 decimal digits are stored…

Summary

- Following this lesson, you now
  - Know floating-point numbers are stored using fixed-precision scientific notation
  - Understand that there are issues—they are not perfect
    - In a course on numerical analysis, you will learn to mitigate these weaknesses
  - The `float` data type is insufficiently precise for most engineering computation
    - Graphics are the one exception…
  - Understand that this is defined by the IEEE754 standard

References

[1] No references?

Acknowledgements

[1] Zhuo En Dai

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.

Disclaimer

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.