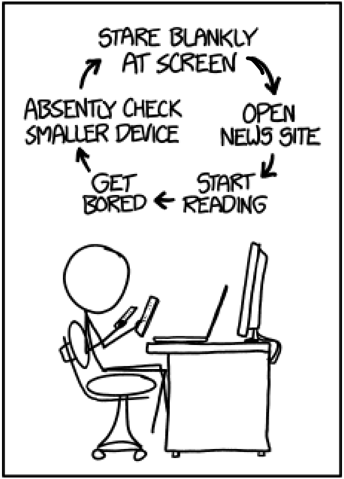
CS319: Scientific Computing

`double`, I/O, flow, loops, and functions

Dr Niall Madden

Week 3: 29 and 31 January, 2025



Source: xkcd (1411)

Slides and examples: https://www.niallmadden.ie/2425-CS319

Outline

1. Preview of Lab 1
2. Recall from Week 2
3. `float`
4. `double`
5. Basic Output
6. Output Manipulators
7. Input
8. Flow of control – `if`-blocks
9. Loops
10. Functions

Slides and examples:
https://www.niallmadden.ie/2425-CS319

Preview of Lab 1

1. Labs start this week.
2. Attend (at least) one hour Thurs 9-10 or Friday 12-1 in AdB-G021.
3. Lab 1 is concerned with program structure, conditionals, and loops. And a little about numbers in C++
4. Nothing to submit: there will be an assignment with Lab 2 next week.

Recall from Week 2

In Week 2 we studied how numbers are represented in C++.

We learned that all are represented in binary, and that, for example,

- An `int` is a whole number, stored in 32 bytes. It is in the range $-2,147,483,648$ to $2,147,483,647$.
- A `float` is a number with a fractional part, and is also stored in 32 bits.

`float`

The format of a variable of type `float` is

$$x = (-1)^{Sign} \times (Significant) \times 2^{(offset+Exponent)},$$

where

- *Sign* is a single bit that determines of the float is positive or negative;
- the *Significant* (also called the "**mantissa**") is the "fractional" part, and determines the precision;
- the *Exponent* determines how large or small the number is, and has a fixed offset (see below).

float

A `float` is a so-called "single-precision" number, and it is stored using 4 bytes ($= 32$ bits). These 32 bits are allocated as:

- 1 bit for the *Sign*;
- 23 bits for the *Significant* (as well as an leading implied bit); and
- 8 bits for the *Exponent*, which has an offset of $e = -127$.

So this means that we write x as

$$x = \underbrace{(-1)^{Sign}}_{\text{1 bit}} \times 1.\underbrace{abcdefghijklmnopqrstuvw}_{\text{23 bits}} \times \underbrace{2^{-127+Exponent}}_{\text{8 bits}}$$

Since the *Significant* starts with the implied bit, which is always 1, it can never be zero. We need a way to represent zero, so that is done by setting all 32 bits to zero.

float

The smallest the *Significant* can be is

$$1.\underbrace{0000000000000000000000}_{\text{22 zeros}}1 \approx 1.$$

The largest it can be is

$$1.\underbrace{11111111111111111111111}_{\text{23 ones}} = 2 - 2^{23} \approx 2.$$

`float`

The smallest the *Significant* can be is

$$1.\underbrace{0000000000000000000000}_{\text{22 zeros}}1 \approx 1.$$

The largest it can be is

$$1.\underbrace{11111111111111111111111}_{\text{23 ones}} = 2 - 2^{23} \approx 2.$$

float

The *Exponent* has 8 bits, but since they can't all be zero (as mentioned above), the smallest it can be is $-127 + 1 = -126$. That means the smallest positive float one can represent is $x = (-1)^0 \times 1.000\cdots 1 \times 2^{-126} \approx 2^{-126} \approx 1.1755 \times 10^{-38}$.

We also need a way to represent ∞ or "Not a number" (`NaN`). That is done by setting all 32 bits to 1. So the largest *Exponent* can be is $-127 + 254 = 127$. That means the largest positive float one can represent is
$x = (-1)^0 \times 1.111\cdots 1 \times 2^{127} \approx 2 \times 2^{127} \approx 2^{128} \approx 3.4028 \times 10^{38}$.

As well as working out how small or large a `float` can be, one should also consider how **precise** it can be. That often referred to as the **machine epsilon**, can be thought of as *eps*, where $1 - eps$ is the largest number that is less than 1 (i.e., $1 - eps/2$ would get rounded to 1).

The value of *eps* is determined by the *Significant*.

For a `float`, this is $x = 2^{-23} \approx 1.192 \times 10^{-7}$.

Probably most important ....

eps is the smallest number such that, if $x = 1$ and $y = 1 - eps$ then $x \mathrel{!}= y$

($x \neq y$ in C++)

As a rule, if a and b are floats, and we want to check if they have the same value, we don't use `a==b`.

This is because the computations leading to a or b could easily lead to some round-off error.

So, instead, should only check if they are very "similar" to each other: `abs(a-b) <= 1.0e-6`

```
if ( 1.0/3.0 ==0.3333333333)
{
    ??
}
```

Don't do this!

`double`

For a `double` in C++, 64 bits are used to store numbers:

- 1 bit for the *Sign*;
- 52 bits for the *Significant* (as well as an leading implied bit); and
- 11 bits for the *Exponent*, which has an offset of $e = -1023$.

The smallest positive double that can stored is $2^{-1022} \approx 2.2251e-308$, and the largest is

$$1.111111\cdots111 \times 2^{2046-1023} = \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots\right) \times 2^{2046-1023}$$
$$\approx 2 \times 2^{1023} \approx 1.7977e+308.$$

(One might think that, since 11 bits are devoted to the exponent, the largest would be $2^{2048-1023}$. However, that would require all bits to be set to 1, which is reserved for NaN).

For a `double`, machine epsilon is $2^{-53} \approx 1.1102 \times 10^{-16}$.

An important example: $\underbrace{\frac{1}{n} + \frac{1}{n} + \frac{1}{n} + \cdots \frac{1}{n}}_{n \text{ times}} - 1 = ??$

00Rounding.cpp

```
    int i, n;
    float x=0.0, increment;

    std::cout << "Enter a (natural) number, n: ";
    std::cin >> n;
    increment = 1/( (float) n);

    for (i=0; i<n; i++)
        x+=increment;

    std::cout << "Difference between x and 1: " << x-1
              << std::endl;

    return(0);
```

What this does:
1. Choose n
2. Add $\frac{1}{n}$ to itself $n-1$ times
3. Subtract 1 from the result.

- ~~If we input $n = 8$, we get:~~
- ~~If we input $n = 10$, we get:~~

n	output
2	0
3	0
4	0
5	0
6	0
7	0
8	0

n	output
9	0
10	1.192×10^{-7}
11	"
12	-1.193×10^{-7}
13	0
14	$+1.193 \times 10^{-7}$
15	0
16	0

We now know...

- An `int` is a whole number, stored in 32 bytes. It is in the range $-2,147,483,648$ to $2,147,483,647$.
- A `float` is a number with a fractional part, and is also stored in 32 bits.
  A positive `float` is in the range 1.1755×10^{-38} to 3.4028×10^{38}.
  Its **machine epsilon** is $2^{-23} \approx 1.192 \times 10^{-7}$.
- A `double` is also number with a fractional part, but is stored in 64 bits.
  A positive `double` is in the range 2.2251×10^{-308} to 31.7977×10^{308}.
  Its **machine epsilon** is $2^{-53} \approx 1.1102 \times 10^{-16}$.

Basic Output

Last week we had this example: *To output a line of text in C++:*

```
#include <iostream>
int main() {
  std::cout << "Howya World.\n";
  return(0);
}
```

Recall \n is "new line".

- the identifier `cout` is the name of the **Standard Output Stream** – usually the terminal window. In the programme above, it is prefixed by `std::` because it belongs to the *standard namespace*...
- The operator `<<` is the **put to** operator and sends the text to the *Standard Output Stream*.
- As we will see `<<` can be used on several times on one lines. E.g.

```
std::cout << "Howya World." << "\n";
```

As well as passing variable names and string literals to the output stream, we can also pass **manipulators** to change how the output is displayed.

For example, we can use `std::endl` to print a new line at the end of some output.

In the following example, we'll display some Fibbonaci numbers. Note that this uses the `for` construct, which we have not yet seen before. It will be explained later.

Output Manipulators

01Manipulators.cpp

```
#include <iostream>
#include <string>
#include <iomanip>  ← new
int main()
{
  int i, fib[16];
  fib[0]=1; fib[1]=1;

  std::cout << "Without setw  manipulator" << std::endl;
  for (i=0; i<=12; i++)
  {
    if( i >= 2)
      fib[i] = fib[i-1] + fib[i-2];
    std::cout << "The " << i << "th " <<
      "Fibonacci Number is " <<  fib[i] << std::endl;
  }
```

- `std::setw(n)` will the width of a field to *n*. Useful for tabulating data.

01Manipulators.cpp

```
std::cout << "With the setw  manipulator" << std::endl;
for (i=0; i<=12; i++)
{
  if( i >= 2)
    fib[i] = fib[i-1] + fib[i-2];
  std::cout
    << "The " << std::setw(2) << i << "th "
    << "Fibonacci Number is "
    << std::setw(3) <<  fib[i] << std::endl;
}
```

Other useful manipulators:

- `setfill` → by default we fill with space. But sometimes need, eg, 0 (zero).
- `setprecision`
- `fixed` and `scientific`
- `dec`, `hex`, `oct`

number of decimal places.

Input

In C++, the object *cin* is used to take input from the standard input stream (usually, this is the keyboard). It is a name for the *C***onsole** *IN***put**.

In conjunction with the operator >> (called the **get from** or **extraction** operator), it assigns data from input stream to the named variable.

(In fact, `cin` is an **object**, with more sophisticated uses/methods than will be shown here).

Input

02Input.cpp

```
#include <iostream>
#include <iomanip> // needed for setprecision
int main()
{
  const double StirlingToEuro=1.19326; // Correct 29/01/2025
  double Stirling;
  std::cout << "Input amount in Stirling: ";
  std::cin >> Stirling;
  std::cout << "That is worth "
            << Stirling*StirlingToEuro << " Euros\n";
  std::cout << "That is worth " << std::fixed
            << std::setprecision(2) << "\u20AC"
            << Stirling*StirlingToEuro << std::endl;
  return(0);
}
```

Euro.

Finished here Wed @ 5