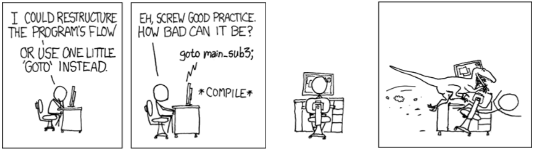
Annotated slides from 4pm class

CS319: Scientific Computing

Getting Started with C++

Dr Niall Madden

Week 2: **9am and 4pm**, 17 January, 2024



Source: xkcd (292)

	Mon	Tue	Wed	Thu	Fri
9 – 10			✓	LAB(?)	
10 – 11					
11 – 12					LAB(?)
12 – 1					LAB(?)
1 – 2					
2 – 3					
3 – 4					
4 – 5			✓		

- My thanks to those who sent me your time-table information.
- Based on that everyone can attend at least two of
  - Thursday 9-10
  - Friday 11-12
  - Friday 12-1.
- First lab is next week (Week 3).
- **Any questions?**

Outline

Variables

The variables/data types we can define include

- `int`
- `float`
- `double`
- `char`
- `bool`

Variables

Integers (positive or negative whole numbers), e.g.,

```
int i; i=-1;
int j=122;
int k = j+i;
```

← Combining declaration & assignment

← assumes j already has a value.

Floats These are not whole numbers. They usually have a decimal places. E.g,

```
float pi=3.1415;
```

Note that one can initialize (i.e., assign a value to the variable for the first time) at the time of definition. We'll return to the exact definition of a `float` and `double` later.

These declarations can be modified.

Eg const int j=122; // j cannot change

Variables

Characters Single alphabetic or numeric symbols, are defined using the `char` keyword:

`char c;` or `char s='7';`

Note that again we can choose to initialize the character at time of definition. Also, the character should be enclosed by single quotes.

Arrays We can declare **arrays** or **vectors** as follows:

`int Fib[10];`

This declares a integer array called `Fib`. To access the first element, we refer to `Fib[0]`, to access the second: `Fib[1]`, and to refer to the last entry: `Fib[9]`.

As in Python, all vectors in C++ are indexed from `0`.

Variables

Here is a list of common data types. Size is measured in bytes.

byte = 8 bits, bit is 0 or 1.

Type	Description	(*min*) **Size**
`char`	character	1
`int`	integer	4
`float`	floating point number	4
`double`	16 digit (approx) float	8
`bool`	`true` or `false`	1

See also: `01variables.cpp`

. . . .

In C++ there is a distinction between **declaration** and **assignment**, but they can be combined.

There are other data types – will come back to them as needed.

As noted above, a `char` is a fundamental data type used to store as single character. To store a word, or line of text, we can use either an *array of `char`s*, or a `string`.

If we've included the *`string`* header file, then we can declare one as in: `string message="Well, hello again";` This declares a variable called *`message`* which can contain a string of characters.

03stringhello.cpp

```
#include <iostream>
#include <string>
int main()
{
   std::string message="Well, hello again";
   std::cout << message << std::endl;
   return(0);
}
```

In previous examples, our programmes included the line
`#include <iostream>`
Further more, the objects it defined were global in scope, and not exclusively belonging to the *std* namespace...

A **namespace** is a declarative region that localises the names of identifiers, etc., to avoid name collision. One can include the line
`using namespace std;`
to avoid having to use `std::`

-=-=-=-

It is a little like using
from MODULE import *
rather than
import MODULE

A closer look at `int`

It is important for a course in Scientific Computing that we understand how numbers are stored and represented on a computer.

Your computer stores numbers in binary, that is, in base 2. The easiest examples to consider are `int`egers.

Examples:

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7

So, $abcd$ in binary

is $d + 2c + 4b + 8a$
$= d(2^0) + c(2^1) + b(2^2) + a(2^3)$.

Same as in decimal
$wxyz$ in decimal is
$z(10^0) + y(10) + x(10^2) + w(10^3)$.

A closer look at `int`

If we use a single byte to store an integer, then we can represent:

Binary	Decimal
0 0 0 0 0 0 0 0	0
0 0 0 0 0 0 0 1	1
0 0 0 0 0 0 1 0	2 .
. . . . . .	
1 0 0 0 0 0 0 0	128
1 1 1 1 1 1 1 1	$255 = 2^8 - 1$

So, we have $2^8 = 256$ different numbers.

A closer look at `int`

In fact, 4 bytes are used to store each `integer`. One of these is used for the sign. Therefore the largest `integer` we can store is $2^{31} - 1$ ...

$= 2 \times 10^9.$

. . . .

We'll return to related types (`unsigned int`, `short int`, and `long int`) later.

A closer look at `float`

C++ (and just about every language you can think of) uses IEEE Standard Floating Point Arithmetic to approximate the real numbers. This short outline, based on Chapter 1 of O'Leary "*Scientific Computing with Case Studies*".

A floating point number ("**float**") is one represented as, say, 1.2345×10^2. The "fixed" point version of this is 123.45.

Other examples:

$$0.01234 = 1.234 \times 10^{-2}$$

Etc.

A closer look at `float`

As with integers, all floats are really represented as binary numbers.

Just like in decimal where ~~3.1425 is~~: 0.03142 is

$$3.142 \times 10^{-2} = (3 \times 10^{0} + 1 \times 10^{-1} + 4 \times 10^{-2} + 2 \times 10^{-3}) \times 10^{-2}$$
$$= 3 \times 10^{-2} + 1 \times 10^{-3} + 4 \times 10^{-4} + 2 \times 10^{-5}$$

For the floating point binary number (for example)

$$1.1001 \times 2^{-2} = (1 \times 2^{0} + 1 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}) \times 2^{-2}$$
$$= 1 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 0 \times 2^{-4} + 1 \times 2^{-6}$$
$$= \frac{1}{4} + \frac{1}{8} + \frac{1}{64} = \frac{25}{16} = 0.390625.$$

But notice that we can choose the exponent so that the representation always starts with 1. That means we don't need to store the 1: it is **implied**.

The format of a 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; Note $(-1)^1 = -1$, $(-1)^0 = 1$
- 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).

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.

A closer look at `float`

The smallest the *Significant* can be is

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

The largest it can be is

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

A closer look at `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 $x = (-1)^0 \times 1.111 \cdots 1 \times 2^{127} \approx 2 \times 2^{127} \approx 2^{128} \approx 3.4028 \times 10^{38}$.

A closer look at `float`

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 $x = 2^{-23} \approx 1.192 \times 10^{-7}$.

That is (1 == 1-1.0e-8) is true!

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`

A closer look at `float`

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).

Finished here
5pm

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