Introductory Lecture

CS319: Scientific Computing (with C++)

Lecturer: Niall Madden

Week 1: 11 Jan, 2017

Source: Wikipedia

http://www.maths.nuigalway.ie/~niall/CS319/

Outline

Lecturer details:

Who: Dr Niall Madden, School Mathematics, Statistics and Applied Mathematics.

Where: Office: AdB-1013, Arás de Brún.

Contact: Email: Niall.Madden@NUIGalway.ie
Phone (091 49) 3803.

Students:

Who: 3rd year Computer Science (3BS9) and Mathematics Science (3BMS2), and 4th Year Applied Mathematics.

Why: As a core subject in your CS degree, or as a compliment to your studies in pure and applied mathematics.

Officially) In CS319 we are primarily concerned with two issues:

1. How to use a computer to solve a scientific problem. That is:
   - how to determine the best algorithm to apply in a given situation.
   - how to understand the potentials and limitations of the algorithm.
2. Implementing that algorithm: How to write the code in C++.

(In reality): this is a course on **object oriented programming in C++**. The primary learning outcome is that, by the end of the semester, you will be a reasonably proficient C++ programmer, and can honestly list "object-oriented programming in C++" as one of your skills in your CV.

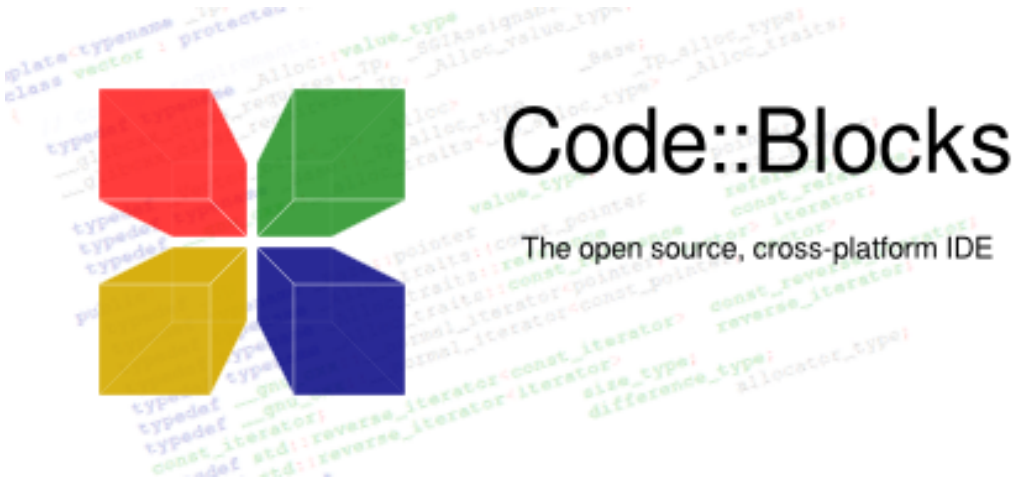
But to give the course context, we'll motivate our studies with examples based on Scientific Computing. This could involve

- Understanding how computers represent numbers;
- Learning how to store and manipulate dense and sparse matrices;
- Study examples in, e.g., data fitting, image processing, network analysis, encryption, ...

Lectures: Wednesday at **9.00 and 16.00** in AdB-G021. (Do you have access?)

Labs:

- 2 hours per week, in AdB-G021, starting in Week 2.
- We will use `Code::Blocks`, which is installed in labs, and is freely available to install on your own PC (Linux/Windows/Mac)



Source: http://wiki.codeblocks.org/

The final grade for CS319 will be based on

- Participation in lectures and labs, **regular lab assignments** (30%)
- An individual project (10%)
- a 2 hour written exam at the end of Semester 2 (60%)

	Mon	Tue	Wed	Thu	Fri
9 – 10	✓	✗	✓		
10 – 11	✓	✗			✗
11 – 12	✗	✗	✗		
12 – 1	✗	✗	✗	✗	✗
1 – 2		✗	✗	✗	✗
2 – 3		✗	✗	✗	
3 – 4	✗	✗	✗	✗	
4 – 5	✗	✗	✗	✗	
5 – 6	✗	✗	✗	✗	

(Data here includen time-table information from KMcG, CH, GC, OMcD).

http://www.maths.nuigalway.ie/~niall/CS319	}	Lecture and lab notes
http://nuigalway.blackboard.com	}	Emails and gradebook
https://github.com/niallmadden/CS319.git	}	C++ Examples

Content:

- Some (but not all!) lecture notes
- Lab notes
- Course news
- Resources
- All C++ examples done in class. These are available from both the course website, and github; the latter has the advantage that they can be all downloaded at once.

Important: I'll use the Blackboard system to send you emails. Please ensure that you have configured Blackboard so that you are sent emails to the account you usually use.

Here are several texts that are useful for additional reading.

Practical C++ programming , S. Oualline, ISBN: 0596004192 (005.133 C++.0) This will be our primary C++ text.

Also... C++ in a Nutshell, by Ray Lischner, ISBN 059600298X (005.133 C.LIS).

More will be added later that deal specifically with Scientific Computing. For example:

Scientific Computing with Case Studies by Dianne P. O'Leary, SIAM Press , 2009. In particular, our first lectures on errors and computer representation of numbers will be based on this.

Solving PDEs in C++ by Yair Shapira. Although we won't study partial differential equations in any detail, some of the C++ content will be very useful.

Scientific Computing

Diane O'Leary[1] describes a **computational scientist** as someone whose focus is the intelligent use of mathematical software to analyse mathematical models.
These models arise from problems formulated by scientists and engineering. Solutions/models can then be constructed using statistics and mathematics. Numerical methods are then employed to design algorithms for extracting useful information from the models.

In scientific computing, we are interested in the correct, reliable and efficient implementation of these algorithms. This requires knowledge of how computers work, and particularly how numbers are represented and stored. History has shown that mistakes can be very, very costly.

Source: Wikipedia

[1] *Scientific Computing with Case Studies*, p7

Object oriented programming in C++

In order to be able to write the code to implement a nontrivial algorithm, one needs to a good grasp of a programming language. Efficiency (usually) requires a high-level complied language. We'll use C++.[2]

1. From Python to C++: input and output, data types and variable declarations, arithmetic, loops, Flow of control (`if` statements), conditionals, and functions.
2. File management and data streams.
3. Arrays, pointers, strings, and dynamic memory allocation.
4. Abstract data types: objects and classes.
5. Programming utilities such as **make**, **gdb**, **git**, ...

[2] This is still the go-to language for scientific computing in industry, though often mixed with C. Fortran is still hanging in there. Python will likely take over from C++ in the near future.

What is Object-Oriented Programming?

Programing languages may be divided into three groups:

(i) *Procedural*

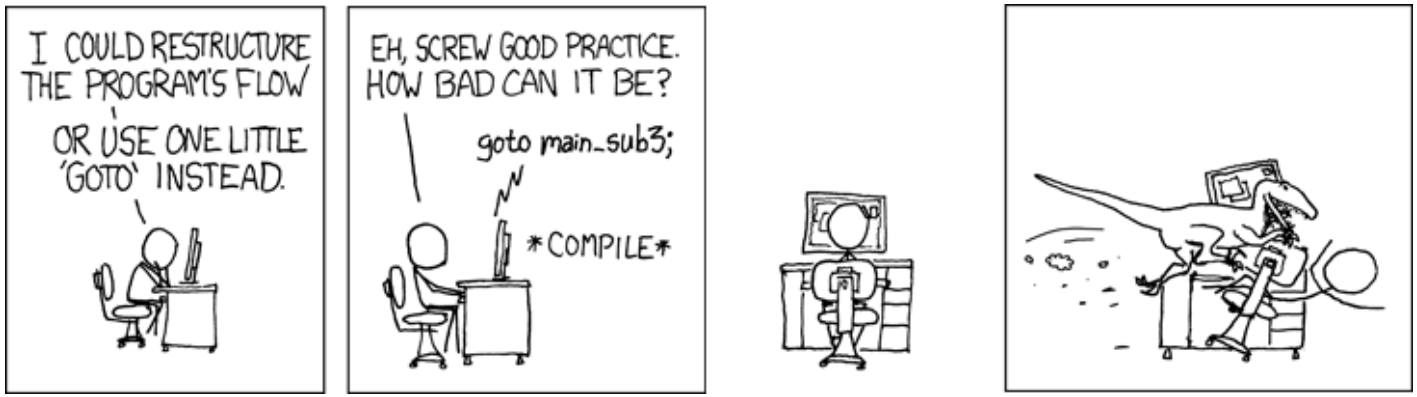
(ii) *Structured*, and

(iii) *Object-Oriented*.

Languages, such as very early versions of Fortran containing, the heavily relied on `goto` instructions are considered *procedural*.

Further evolution led to languages such as C and Pascal which allow *structured programming*:

- Modular programming style – each task is divided and subdivided into functions and programming blocks.
- Recursion, and global & local variables are allowed.
- Goto is not used. (Why not? See `http://xkcd.com/292/`)



Object-Oriented

OO languages approach problem solving by constructing *objects* – pieces of data and the functions that manipulate them. The programmer can capitalise on the relationship between various objects in order to simplify his/her tasks.

Most modern languages are classified, not by whether they are procedural, structured or object-oriented – but by which of these paradigms is promoted most strongly by the language. For example, one could write a C++ program that is entirely procedural or structured.

At the core of OOP are the three concepts of

1. *Encapsulation* 2. **Polymorphism** 3. **Inheritance**.

With **encapsulation** data and the program code to manipulate it are bound together to form an *object*.
A datum or function belonging to an object is called a *member*.
Within an object, code and data may be either

- **Private**: accessible only to another part of that object, or
- **Public**: other parts of the program can access it even though it belongs to a particular object.
  Public parts of an object provide a interface to the object for other parts of the program.

An object should be thought of as an **Abstract Data Type (ADT)**: a specialised type of variable that the user can define.

Polymorphism means "many forms". The idea is that we can use one function name or operator to describe a set of distinct but related tasks. The compiler then chooses the appropriate definition depending on the context in which it is used.

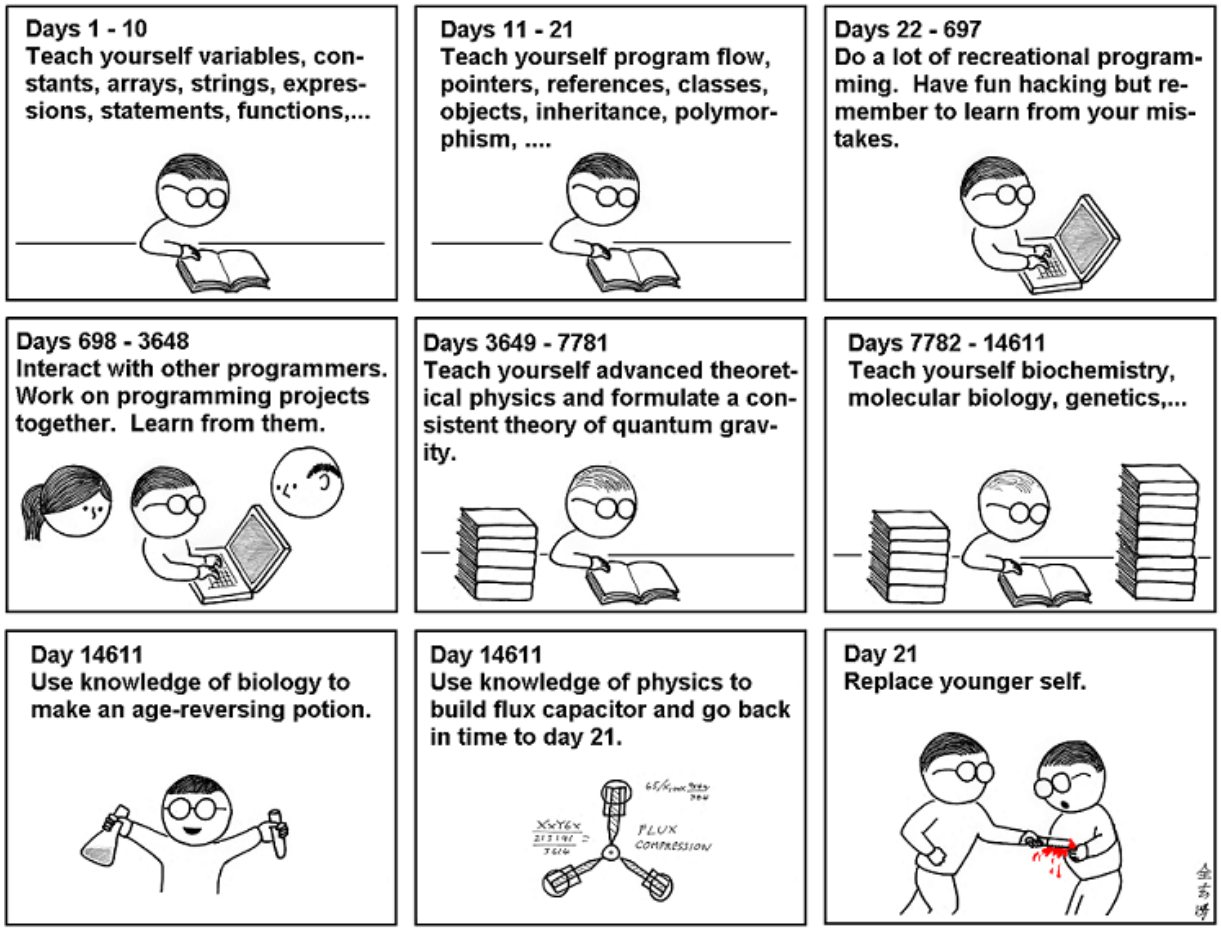
For example, the C language provides three functions: `abs()`, `labs()`, and `fabs()` to calculate the absolute value of integers, long integers and floats respectively. In C++ we only require `abs()` – its exact definition (and in particular, its return value) will be chosen depending on the type the data passed to it.
This is *Function Overloading*, an example of polymorphism.

Also important is **Operator Overloading**. This is where a symbol can take its meaning from the context in which it is used. E.g, + could be used to add integers, floats, vectors, matrices, strings, etc.

Inheritance is the process whereby one object can acquire the properties of another. More specifically, an object can inherit a general set of properties to which it can add features that are specific only to itself.
Suppose we are working with a university database. We might have

- a `student` object containing name, address and ID number, and methods of querying and setting these values.
- a `science` type which has all the properties of `student` but also information on whether that student is majoring in CS, Maths, Applied Physics, etc.
- a `CS319` type which as properties of `student`, and `science` but also information such as lab schedules, homework marks, etc.

As a more pertinent example, we might consider a base `Matrix` type that could be extended ...



Source: Abstruse Goose.

Fortunately, we have 84 days to teach ourselves C++.

C++ is a programming language developed as an extension to C. It is a *superset* of C, so C programs can be compiled on a C++ compiler.

The convention is the give C++ programs the suffix `.cpp`, e.g., `hello.cpp`. Other valid extensions are `.C`, `.cc`, `.cxx`, and `.c++`.

.....

We'll use `Code::Blocks` for developing our C++ code. It is available for Mac, Linux, and Windows.

If you *really* want, use can just install gcc, and use it with your favourite text editor, e.g., `gedit`, `kedit`, `kwrite`, `pico`, `emacs`, `WordPad`, ...

The GNU projects C/C++ compiler is used. The invocation is

```
$ g++ hello.cpp
```

If there is no error in the code, an executable file called `a.out` is created.

If the present working directory is contained in your PATH variable, just type its name to run it. Otherwise, indicate that it is in your PWD: `./a.out`

- A "header file" is used to provide an interface to standard libraries. For example, the *iostream* header introduces I/O facilities. Every program that we will write will include the line:
  `#include <iostream>`

- All variables must be declared before begin used. However the definitions do not have to be grouped together at the start of a function – they can occur at any point in the function. Their **scope** is from the point they are declared to the end of the function.

- The heart of the program is the `main()` function – every program needs one. `void` is the default argument list and can be omitted.

- The C++ language is case-sensitive. E.g., the functions `main()` and `Main()` are not the same.

- "Curly brackets" are used to delimit a program block.

- Every (logical) line is terminated by a semicolon;
  Lines of code not terminated by a semicolon
  are assumed to be continued on the next line;

- The backslash **escape** character is used for output. For example, `\n` is used to output a new line.

- Two forward-slashes `//` indicate a comment – everything after them is ignored until an end-of-line is reached.

Basic Output

Basic components:

- `#include <iostream>`
- `cout` and `cin`, *also*
- The operators `<<` and `>>`

To output a line of text in C++:

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

- 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";
  ```

Variables

Variables are used to temporarily store values (numerical, text, etc, ...) and refer to them by name, rather than value.

More formally, the variable's name is called the **identifier**. It must start with a letter or an underscore, and may contain only letters, digits and underscores.
Examples:

All variables must be defined before they can be used. That means, we need to tell the compiler before we use them. This can be done at any stage in the code, up to when the variable is first used.

Every variable should have a **type**; this tells use what sort of value will be stored in it.

The variables/data types we can define include

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

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

Variables

- 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` later.
- 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.
- 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.

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: `02variables.cpp`

In C++ there is a distinction between **declaration** and **assignment**, but they can be combined. (Later we'll see how to use the `const` modifier so that a variables value can't be changed later).

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. Later we'll see that `string` is an example of an object.

03stringhello.cpp

```
#include <iostream>
#include <string>
int main()
{
  std::string message="Well, hello again.";

  std::cout << message << "\n";
  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. In traditional C++, names of library functions are placed in the global namespace, as in C. With ANSI/ISO (Standardised) C++ they are placed within a namespace called `std`. Officially one should include the following line to make them visible:

```
using namespace std;
```

Hello with `std` namespace

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
   string message="Well, hello again.";

   cout << message << endl;
   return(0);
}
```

Here we have used the identifier `endl` to end a line. This is referred to as a "**manipulator**".

Later, we'll return to the concept of output manipulators to see, for example, how to use them to format C++ output into tables.

int: a closer look

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:

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

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

. . . . . .

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

float: a closer look

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*". The format of a float is $x = (-1)^{Sign} \times (Significant) \times 2^{Exponent}$ 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 usually involves an 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 $e = -127$.

float: a closer look

So this means that we write x as

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

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.
Thus, the smallest the *Significant* can be is $1.\underbrace{00000000000000000000}_{22\text{ zeros}}1 \approx 1$.

The largest it can be is $1.\underbrace{11111111111111111111}_{23\text{ ones}} = 2 - 2^{23} \approx 2$.

Here it helps to remember that the binary faction 1.1 means (in decimal) $1 + \frac{1}{2}$, 1.11 means $1 + \frac{1}{2} + \frac{1}{4}$, etc.

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
$x = (-1)^0 \times 1.000\cdots1 \times 2^{-126} \approx 2^{-126} \approx 1.1755e-38$.

float: a closer look

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 $x = (-1)^0 \times 1.111 \cdots 1 \times 2^{127} \approx 2 \times 2^{127} \approx 2^{128} \approx 3.4028e + 38$.

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

For a `double` in C++, 64 bits are used to store numbers. These are allocated as

- 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

$$1.111111\cdots111 \times 2^{-1023} = \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots\right) \times 2^{-1023}$$
$$\approx 2^{-1024} \approx 1.5730e+308.$$

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

An important example:

Week01/04Rounding.cpp ← link!

```
int i, max;
float x, increment;

cout << "Enter a (natural) number, n: ";
cin >> max;
x=0.0;
increment = 1/( (float) max);

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

std::cout << "Difference between x and 1 is " << x-1 << "\n";
```

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