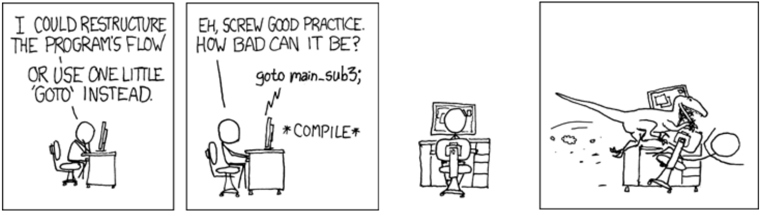
CS319: Scientific Computing (with C++)

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

`ints` and `float`; input & output; flow; loops; functions

Week 2: **9am and 4pm**, 17 Feb 2021



Source: xkcd (292)

Course materials, lectures, etc

This module will run “remotely” in its entirety. As this is the first (and, hopefully, last) time that will happen, we will adapt...
For now, the plan is

- We won’t distinguish between “lectures” and “labs”; and will call them all “classes”.
- There will be two classes per week, (probably) increasing to three from Week 3.
- For the first two weeks, classes will be similar to traditional lectures, but from Week 3, there will be more interactive lab-type sessions.
- **All non-interactive parts will be recorded, and recordings will be made available the day after classes**.
- Recordings will be broken into chunks of 10-15 minutes, and published in “Videos” section.
- Slides will be made available separately.
- This will be reviewed regularly; expect numerous short surveys!

This module will run “remotely” in its entirety. As this is the first (and, hopefully, last) time that will happen, we will adapt...
For now, the plan is

- We won’t distinguish between “lectures” and “labs”; and will call them all “classes”.
- There will be two classes per week, (probably) increasing to three from Week 3.
- For the first two weeks, classes will be similar to traditional lectures, but from Week 3, there will be more interactive lab-type sessions.
- **All non-interactive parts will be recorded, and recordings will be made available the day after classes**.
- Recordings will be broken into chunks of 10-15 minutes, and published in “Videos” section.
- Slides will be made available separately.
- This will be reviewed regularly; expect numerous short surveys!

The schedule is problematic, involving various clashes.
To help resolve this ...

Exercise (Your time-table)

TODAY, send Niall you time-table (in any format).
In addition, identify times that you cannot participate in classes.

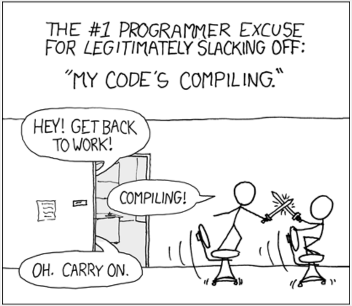
Exercise (Bitbucket)

You should have access to the CS319 git repository at `https://bitbucket.org/niallmadden/2021-cs319/src` *If not, check your email for an invitation. If still not working, send Niall an email.*

From last week:

Exercise

Using https://xkcd-excuse.com make your own version of:



Source: https://xkcd.com/303

and send it to me. If possible, make it relate to something we cover this week or last week.

Today:

- Another exercise

1. Part 1: C++ fundamentals
   - Recap on last week
   - Strings
   - Header files and Namespaces
2. Part 2: a closer look at `int`
3. Part 3: a closer look at `float`
   - `double`
4. Part 4: Output Manipulators
   - endl
   - setw
5. Part 5: Input
6. Part 6: Flow of control – `if`-blocks

- A "header file" is used to provide an interface to standard libraries. Every program that we write has the line:
  `#include <iostream>`

- 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.

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

- Every (logical) line is terminated by a semicolon;

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

- To send output to the console, use `std::cout`, along with the `<<` ("put to") operator E.g.
  ```
  std::cout << "Howya World." << "\n";
  ```
- **Variables** are used to temporarily store values (numerical, text, etc, ...). All s must be defined before they can be used.
- Every variable has a **type**

  `int` stores (positive or negative) whole numbers

  `floats`: stores non-integer numbers. So does the `double` type (more on this later).

  `char`: stores alphabetic or numeric symbols.

  Arrays: of any type, are indexed from zero, with index in square brackets.

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 `chars`*, 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.

02stringhello.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>`
Furthermore, 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.

With ANSI/ISO (Standardised) C++ they are placed within a namespace called `std`.

One could include the following line to make them visible:

```
using namespace std;
```

and then one can use `cout` rather than `std::cout`.

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.

Part 2: 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:

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

Part 2: 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 $2^{31} - 1$ ...

. . . .

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

Part 3: 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*".

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 has an offset (See below).

Part 3: a closer look at `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.

Part 3: 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 $1.\underbrace{11111111111111111111111}_{\text{23 ones}} = 2 - 2^{23} \approx 2$.

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
$x = (-1)^0 \times 1.000 \cdots 1 \times 2^{-126} \approx 2^{-126} \approx 1.1755 \times 10^{-38}$.

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

Part 3: 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 $x = 2^{-23} \approx 1.192 \times 10^{-7}$.

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:

03Rounding.cpp

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

std::cout << "Enter a (natural) number, n: ";
std::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
          << std::endl;
```

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

Part 4: Output Manipulators — endl

As well as passing variable names and strings to the output stream, we can also pass manipulators to change how variable values are displayed. Some manipulators (e.g., `setw`) require that *iomanip* is included.

- `endl` print a new line (and flush)

04Manipulators.cpp

```
#include <iomanip>

int main()
{
  int i, fib[16];
  fib[0]=1; fib[1]=1;

  std::cout << "\n\nWithout 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 " << i << "th " <<
      "Fibonacci Number is " <<  fib[i] << std::endl;
  }
```

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

04Manipulators.cpp

```
 std::cout << "\n\nWith the setw  manipulator" << 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] << endl;
 }
 return(0);
}
```

Other useful manipulators:

- `setfill`
- `setprecision`
- `fixed` and `scientific`
- `dec`, `hex`, `oct`

Part 5: 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.

(Later we will see that `cin` is an **object**, with more sophisticated uses/methods than is going to be shown here. However, we will defer this discussion until we have studied something of **objects** and **classes**).

Part 5: Input

05Input.cpp

```
#include <iostream>
#include <iomanip> // needed for setprecision

int main()
{
  const double StirlingToEuro=1.17703; // Correct 22/01/2020
  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);
}
```

Part 6: Flow of control – if-blocks

`if` statements are used to conditionally execute part of your code.

Structure (i):

```
if( exprn )
{
  statements to execute if exprn evaluates as
               non-zero
}
else
{
   statements if exprn evaluates as 0
}
```

Part 6: Flow of control – if-blocks

The argument to `if()` is a **logical expression**.

Example

- `x == 8`
- `m == '5'`
- `y <= 1`
- `y != x`
- `y > 0`

More complicated examples can be constructed using **AND** `&&` and **OR** `||`.

Part 6: Flow of control – if-blocks

06EvenOdd.cpp

```
#include <iostream>

int main(void)
{
  int Number;

  std::cout << "Please enter an integrer: ";
  std::cin >> Number;

  if ( (Number%2) == 0)
    std::cout <<  "That is an even number." << std::endl;
  else
    std::cout <<  "That number is odd." << std::endl;

  return(0);
}
```

Part 6: Flow of control – `if`-blocks

More complicated examples are possible:

Structure (ii):

```
if( exp1 )
{
    statements to execute if exp1 is "true"
}
else if (exp2)
{
    statements run if exp1 is "false" but exp2 is "true"
}
else
{
    "catch all" statements if neither exp1 or exp2 true.
}
```

Part 6: Flow of control – if-blocks

07Grades.cpp

```
int main(void)
{
   int NumberGrade;
   char LetterGrade;

   std::cout << "Please enter the grade (percentage): ";
   std::cin >> NumberGrade;

   if ( NumberGrade >= 70 )
      LetterGrade = 'A';
   else if ( NumberGrade >= 60 )
      LetterGrade = 'B';
   else if ( NumberGrade >= 50 )
      LetterGrade = 'C';
   else if ( NumberGrade >= 40 )
      LetterGrade = 'D';
   else
      LetterGrade = 'E';

   std::cout << "A score of " << NumberGrade << "% cooresponds to
            << LetterGrade << "." << std::endl;
}
```

Part 6: Flow of control – `if`-blocks

The other main flow-of-control structures are the

- `switch ... case` structures
- the use of the `?` and `:` operators.

Exercise 2.1

- Find out how `switch.. case` works. Rewrite the Even/Odd example above using `switch ... case`.
- What errors/bugs/problems are there with the `Grades` example? That is, how could you get it to break?
- Read up on the *`switch / case`* construct. Can it be used to write an improved version of the programme. (Hint: yes, but you need a recent C++ compiler...).