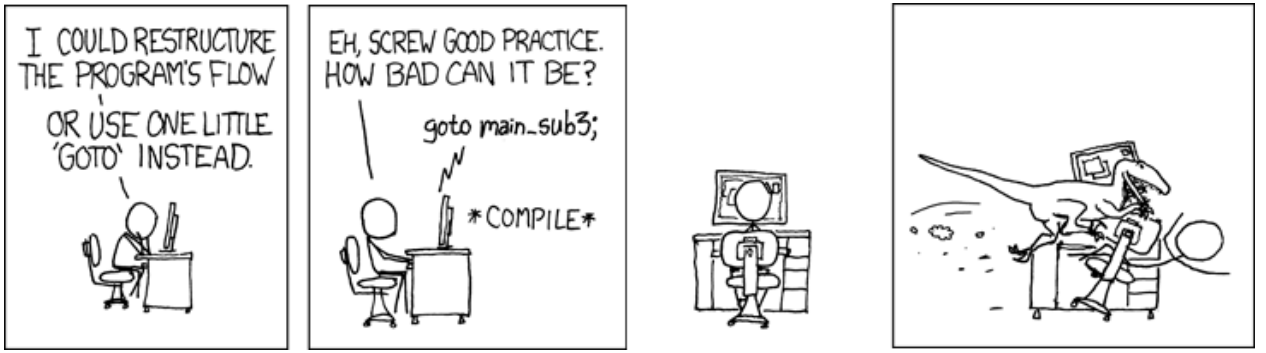
CS319: Scientific Computing

Data types in C++; Input and output [DRAFT!]

Dr Niall Madden

Week 2: 22nd and 24th January, 2025



Source: xkcd (292)

	Mon	Tue	Wed	Thu	Fri
9 – 10				Lab	
10 – 11					???
11 – 12					Lecture
12 – 1					Lab
1 – 2					
2 – 3					
3 – 4					
4 – 5			Lecture		

... still working on the lab times...

Outline

Reminder: Programming Platform

To get started, we'll use an online C++ compiler. Try one of the following

- `http://cpp.sh`
- `https://www.programiz.com/cpp-programming/online-compiler/`
- `https://www.onlinegdb.com` (seems not to be working at the moment...)

If using a PC in the lab, try `Code::blocks`.

You should also install a compiler and IDE on your own device (see notes from Week 1).

Recall: variables

Last week, we learned that **variables** are used to temporarily store values (numerical, text, etc, ...) and refer to them by name, rather than value.

- Unlike Python, variables must be defined before they can be used. That means, we need to tell the compiler the variable's **name** and **type**
- Their **scope** is from the point they are declared to the end of the function.
- Formally, the variable's **name** is an example of an **identifier**. It must start with a letter or an underscore, and may contain only letters, digits and underscores.
- The **data types** we can define include `int`, `float`, `double`, `char`, and `bool`

Types

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

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

← most common.

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.

Modifiers:

* unsigned int i = 12; // can't be negative.
* const int i =12 ; // can't be changed later

Types

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

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

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.

Types

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`.

Types

Here is a list of common data types. Size is measured in bytes. = 8 bits

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: `00variables.cpp`

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

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.

01stringhello.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...

This is a bit like, in Python:

```
import MOD;  % now access members as MOD.whatever
```

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;
```

*A bit like: from MOD import **

to avoid having to use `std::`

(The C++ world is a little divided on whether `using namespace std` is good or bad practice. Personally, I don't use it, because it can impact the portability of my code. Only exception is that sometimes it helps fit my code examples onto a slide!)

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:

decimal	binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110

etc.

A closer look at `int`

If just one byte were used to store an integer, then we could represent:

Assuming the int is non-negative we can store

$$11111111 = 255 = 2^8 - 1$$

64 + 32 + 16 + 8 + 4 + 2 + 1

+128

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

4 bytes = 4x8 = 32 bit.
One is needed for the sign (+ or -)
and 31 for the value

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

$2^{31} - 1 = 2,147,483,647$

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:

$50.0 = 5.00 \times 10^1$

$0.123 = 1.23 \times 10^{-1} = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3}$

$0.0045 = 4.5 \times 10^{-3}$

$9.15 = 9.15 \times 10^0$

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

Just like in decimal where 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;
- 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

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

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.

Finish here.