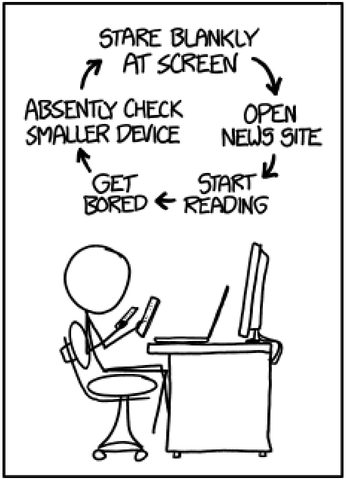
CS319: Scientific Computing

`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

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

machine epsilon

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

Comparing floats

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`

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

$$1.111111\cdots 111 \times 2^{2046-1023} = \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots\right) \times 2^{2046-1023}$$
$$\approx 2 \times 2^{1023} \approx 1.7977e + 308.$$

(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 $2^{-53} \approx 1.1102 \times 10^{-16}$.

double

An important example:

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:

`double`

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

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);
}
```

- 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";
```

Output Manipulators

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>
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;
  }
```

Output Manipulators

- `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;
}
```

Output Manipulators

Other useful manipulators:

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

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);
}
```

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
}
```

Flow of control – `if`-blocks

Note: { and } are optional if the block contains a single line.

Example:

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** `||`.

Flow of control – if-blocks

03EvenOdd.cpp

```cpp
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);
}
```

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.
}
```

Flow of control – if-blocks

04Grades.cpp

```
   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 a "
             << LetterGrade << "." << std::endl;
```

Flow of control – `if`-blocks

The other main flow-of-control structures are

- the ternary the `?:` operator, which can be useful for formatting output, in particular, and
- `switch ... case` structures.

Exercise 2.1

Find out how the `?:` operator works, and write a program that uses it.
Hint: See Example `07IsComposite.cpp`

Exercise 2.2

Find out how `switch... case` construct works, and write a program that uses it.
Hint: see `https://runestone.academy/ns/books/published/cpp4python/Control_Structures/conditionals.html`

Loops

We meet a `for`-loop briefly in the Fibonacci example. The most commonly used loop structure is `for`

```
for (initial value; test condition; step)
{
    // code to execute inside loop
}
```

Example: `05CountDown.cpp`

```
int main(void)
{
  int i;
  for (i=10; i>=1; i--)
    std::cout << i << "... ";
  std::cout << "Zero!\n";
  return(0);
}
```

1. The syntax of `for` is a little unusual, particularly the use of semicolons to separate the "arguments".

2. All three arguments are optional, and can be left blank. Example:

3. But it is not good practice to omit any of them, and very bad practice to leave out the middle one (test condition).

4. It is very common to define the increment variable within the for statement, in which case it is “local” to the loop. Example:

5. As usual, if the body of the loop has only one line, then the “curly braces”, { and }, are optional.

6. There is no semicolon at the end of the `for` line.

Loops

The other two common forms of loop in C++ are

- `while` loops
- `do ... while` loops

Exercise 2.3

Find out how to write a `while` and `do ... while` loops. For example, see
https://runestone.academy/ns/books/published/cpp4python/Control_Structures/while_loop.html
Rewrite the **count down** example above using a

1. `while` loop.
2. `do ... while` loop.

Functions

A good understanding of **functions**, and their uses, is of prime importance.

Some functions return/compute a single value. However, many important functions return more than one value, or modify one of its own arguments.

For that reason, we need to understand the difference between **call-by-value** and **call-by-reference** ($\longleftarrow$ later).

Functions

Every C++ program has at least one function: `main()`

Example

```
#include <iostream>
int main(void )
{
   /* Stuff goes here */
   return(0);
}
```

Functions

Each function consists of two main parts:

- Function “header” or **prototype** which gives the function’s
  - return value data type, or `void` if there is none, and
  - parameter list data types or `void` if there are none.

  The prototype is often given near the start of the file, before the **main()** section.

- **Function definition**. Begins with the function’s name (identifier), parameter list and return type, followed by the body of the function contained within curly brackets.

Functions

Syntax:

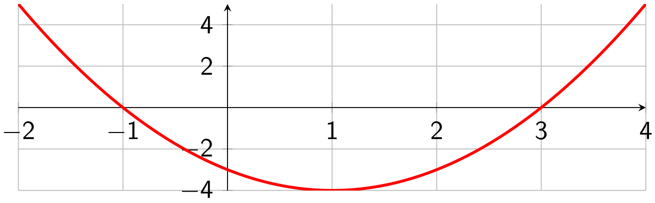
```
ReturnType FnName ( param1, param2, ...)
{
      statements
}
```

- `ReturnType` is the data type of the data returned by the function.
- `FnName` the identifier by which the function is called.
- `Param1, ...` consists of
  - the data type of the parameter
  - the name of the parameter will have in the function. It acts within the function as a local variable.
- the statements that form the function's body, contained with braces `{...}`.

A mathematical function

Since this is a course on scientific computing, we'll often need to define mathematical functions from $\mathbb{R} \to \mathbb{R}$ (more or less), such as $f(x) = e^{-x}$. Typically, such functions map one or more `double`s onto another `double`.

The example we'll look at is $f(x) = x^2 - 2x - 3$.



A mathematical function

06MathFunction.cpp

```
#include <iostream>
#include <iomanip>

double f(double x) // x^2 − 2x − 3
{
  return (x*x - 2*x -3);
}

int main(void){
  double x;
  std::cout << std::fixed << std::showpoint;
  std::cout << std::setprecision(2);
  for (int i=0; i<=10; i++)
  {
    x = -1.0 + i*.5;
    std::cout << "f("<< x << ")="<< f(x) << std::endl;
  }
  return(0);
}
```

Functions

In this example, we write a function that takes an non-negative integer input and checks it its a composite (`true`) or prime (`false`).

07IsComposite.cpp

```
// Check if i as a Composite number (i.e., not prime)
// Return "true" if it is composite.
// Return "false" if it is prime.
bool IsComposite(int i) // should check i > 0
{
  int k;
  for (k=2; k<i; k++)
    if ( (i%k) == 0)
      return(true);

  // If we get to here, then i has no divisors between 2 and i-1
  return(false);
}
```

Calling the `IsComposite` function:

07IsComposite.cpp

```
int main(void )
{
  int i;

  std::cout << "Enter a natural number: ";
  std::cin >> i; // Warning: should check this is positive

  std::cout << i << " is a " <<
    (IsComposite(i) ? "composite":"prime") << " number."
          << std::endl;

  return(0);
}
```

Functions

Most functions will return some value. In rare situations, they don't, and so have a void return value.

08Kth.cpp

```
void Kth(int i);

int main(void )
{
  int i;

  std::cout << "Enter a natural number: ";
  std::cin >> i;

  std::cout << "That is the ";
  Kth(i);
  std::cout << " number." << std::endl;
```

Functions

08Kth.cpp

```
// FUNCTION KTH
// ARGUMENT: single integer
// RETURN VALUE: void (does not return a value)
// WHAT: if input is 1, displays 1st, if input is 2, displays 2nd,
// etc.
void  Kth(int i)
{
  std::cout << i;
  i = i%100;
  if ( ((i%10) == 1) && (i != 11))
    std::cout << "st";
  else if ( ((i%10) == 2) && (i != 12))
    std::cout << "nd";
  else if ( ((i%10) == 3) && (i != 13))
    std::cout << "rd";
  else
    std::cout << "th";
}
```