Annotated slides from Friday

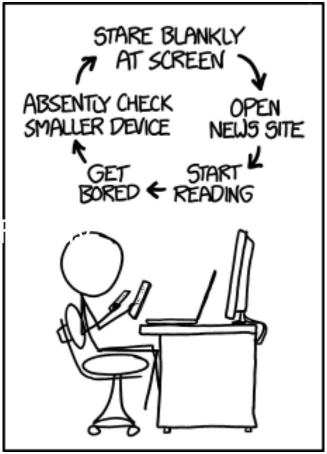
CS319: Scientific Computing

I/O, flow, loops, and functions

Annotated slides from Friday

Dr Niall Madden

Week 3: 29 and 31 January, 2026



Source: xkcd (1411)

Slides and examples: https://www.niallmadden.ie/2526-CS319

0. Inputs for this week's classes:

1. ~~Preview of Lab 1~~
2. ~~Recall from Week 2~~
3. ~~Basic Output~~
4. ~~Output Manipulators~~
   - ~~endl~~
   - ~~setw~~
5. ~~Input~~
6. ~~`if`-blocks~~
7. ~~Loops~~
8. Functions
   - Function Header
   - Function definition
   - A mathematical function
   - E.g, Prime?
   - `void` functions
9. Numerical Integration
   - The basic idea
   - ~~The code~~
   - ~~Trapezium Rule as a function~~

Slides and examples: niallmadden.ie/2526-CS319

2526-CS319

1. 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. Submit your C++ file as it is on Friday. This is just to test that you upload the correct file, and to verify participation. So long as you upload a C++ file, you'll get the mark.

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

3. 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";
```

4. 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. These are defined by the recurrence: $f_0 = 1, \quad f_1 = 1,$ and, for $i > 1$, $f_i = f_{i-1} + f_{i-2}$.

We'll use the `for` construct, which will be explained later in this class.

4. Output Manipulators

01Manipulators.cpp

```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;
  }
```

4. Output Manipulators

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

01Manipulators.cpp

```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;
}
```

4. Output Manipulators

Other useful manipulators:

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

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.

(In fact, `cin` is an **object**, with more sophisticated uses/methods than will be shown here).

5. Input

02Input.cpp

```cpp
#include <iostream>
#include <iomanip> // needed for setprecision
int main()
{
  const double StirlingToEuro=1.19099; // Correct 28/01/2026
  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);
}
```

6. 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
}
```

6. if-blocks

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

Example:

6. 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** `||`.

6. if-blocks

03EvenOdd.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);
}
```

6. 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.
}
```

6. 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;
```

6. `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`

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);
}
```

7. Loops

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).

7. Loops

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.

7. 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.

8. 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).

Warning: In Scientific Computing, we use the term "**function**" in two different, but related ways:

- A mathematical function, such as $f(x) = e^{-x}$ or $u(x, y) = \sin(\pi x) \cos y$.
- A function we code to preform a task, such a determining if an integer is prime.

And often we'll combine both ideas!

8. Functions

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

Example

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

Each function consists of two main parts: Header/Prototype and Body/Definition.

1. Header

The Function "header" or **prototype** 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 header line ends with a semicolon.

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

Syntax for function header:

```
ReturnType FnName (type1, type2, ...);
```

Examples:

```
int find_largest_divisor(int);

int round_to_nearest(float);

int find_lcm(int, int);
```

```
bool  IsClose(float, float, float);
or
bool IsClose(float x, float y, float epsilon);
  // true if |x-y| <= epsilon
```

Note: you can give variable names, optionally, in the header, but they are just place-holders and ignored by the compiler.

2. Function definition

- The **function definition** can be anywhere in the code (after the header).
- First line is the same as the prototype, except variables names need to be included, and that line does not end with a semi-colon.
- That is followed by the body of the function contained within curly brackets.

8. Functions

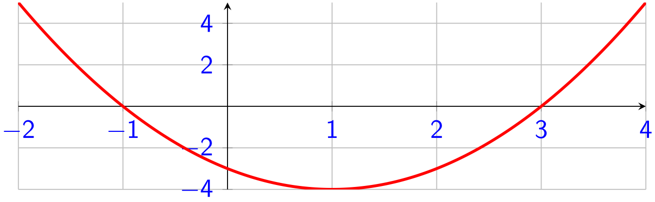
Syntax:

```
ReturnType FnName (type1 param1, type2 param2, ...)
{
      statements
}
```

- ReturnType is the data type of the data returned by the function.
- FnName the identifier by which the function is called.
- type1 param1, ... consists of (eg int, float, ...)
  - 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 {...}.

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

```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);
}
```

Note: for simple functions that appear before the main() the function header can be omitted.

8. 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` (header)

```cpp
// 07IsComposite.cpp
// An example of a simple function, to check if an int is composite
// Author: Niall Madden
// Week 3: 2526-CS319 - Scientific Computing

#include <iostream>

bool IsComposite(int i);
```

Calling the `IsComposite` function

`07IsComposite.cpp` (main)

```
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);
}
```

Defining the IsComposite function

01IsComposite.cpp (function definition)

```
// 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);  // note: function terminates at first "return()"

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

Most functions will return some value (they are sometimes called "*fruitful*" functions). In rare situations, they don't, and so have a `void` return value.

`08Kth.cpp` (header)

```
// 08Kth.cpp:
// An example of a void function.
// CS319, Week 3
// Author: Niall Madden
// Date: Jan 2026
// Week 3: CS319 - Scientific Computing

#include <iostream>

void Kth(int i);   ← header
```

Puzzle: What is the next term in the sequence
s, t, n, d, r, d, t, h, t, h, t, h,
st, nd, rd, th, th, ...

8. Functions

02Kth.cpp (main)

```
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;

  return(0);
}
```

08Kth.cpp (function definition)

```
// 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; //  remainder on dividing i by 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";
}
```

AND.

Recall : OR is ||

9. Numerical Integration

A **Numerical Integration** (aka "Quadrature") method is an algorithm for estimating definite integrals. The applications are far too numerous to list, but feature in just about every area of Applied Mathematics, Probability Theory, and Engineering, and even some areas of pure mathematics.

Although the history is ancient, it continues to be a hot topic of research, particularly when computing with high-dimensional data.

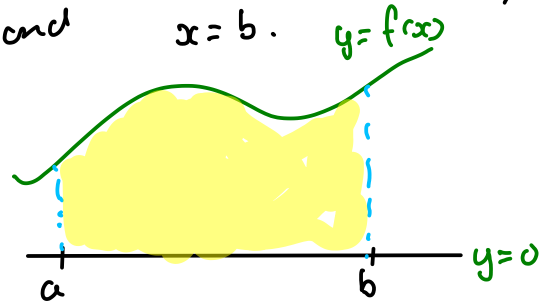
In this section, we want to estimate definite integrates of one-dimensional functions:

$$\int_a^b f(x)dx.$$

We'll use one of the simplest methods: the Trapezium Rule.

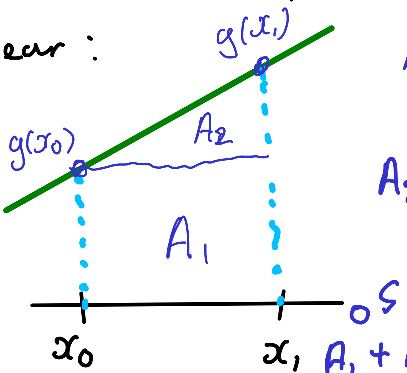
9. Numerical Integration

The value of $\int_a^b f(x)\,dx$ is the area between $y=f(x)$, $y=0$, $x=a$ and $x=b$.



Usually there is no formula for $\int f(x)\,dx$.

But we can write down a formula for $\int_{x_0}^{x_1} g(x)\,dx$ for some simple function $g(x)$ .. e.g, if g is linear:



A_1 has area $(x_1 - x_0)\,g(x_0)$

A_2 has area $\frac{1}{2}(x_1 - x_0)(g(x_1) - g(x_0))$

so

$A_1 + A_2 = \frac{1}{2}(x_1 - x_0)\left(g(x_0) + g(x_1)\right)$

QuadratureV01.cpp (headers)

```
// QuadratureV01.cpp:
// Trapezium Rule (TR) quadrature for a 1D function
// Author: Niall Madden
// Date: Jan 2026
// Week 03: CS319 - Scientific Computing
#include <iostream>
#include <cmath>  // For exp()

double f(double); // prototype
double f(double x) {  return(exp(x)); } // definition
```

QuadratureV01.cpp (main)

```cpp
int main(void )
{
  std::cout << "Using the TR to integrate f(x)=exp(x)\n";
  std::cout << "Integrate f(x) between x=0 and x=1.\n";
  double a=0.0, b=1.0;
  double Int_f_true = exp(1)-1;
  std::cout << "Enter value of N for the Trap Rule: ";
  int N;
  std::cin >> N; // Lazy! Should do input checking.
```

9. Numerical Integration

QuadratureV01.cpp (main continued)

```
  double h=(b-a)/double(N);
  double Int_f_TR = (h/2.0)*f(a);
  for (int i=1; i<N; i++)
    Int_f_TR += h*f(a+i*h);
  Int_f_TR += (h/2.0)*f(b);

  double error = fabs(Int_f_true - Int_f_TR);

  std::cout << "N=" << N << ", Trap Rule=" << Int_f_TR
            << ", error=" << error << std::endl;
  return(0);
}
```

9. Numerical Integration

Typical output:

9. Numerical Integration — Trapezium Rule as a function

Next it makes sense to write a function that implements the Trapezium Rule, so that it can be used in different settings.

The idea is pretty simple:

- As before, `f` will be a globally defined function.
- We write a function that takes as arguments `a`, `b` and `N`.
- The function implements the Trapezium Rule for these values, and the globally defined `f`.

9. Numerical Integration Trapezium Rule as a function

QuadratureV02.cpp (header)

```
// QuadratureV02.cpp: Trapezium Rule as a function
// Trapezium Rule (TR) quadrature for a 1D function
// Author: Niall Madden
// Date: Jan 2026
// Week 03: CS319 - Scientific Computing
#include <iostream>
#include <cmath>  // For exp()
#include <iomanip>

double f(double x) {  return(exp(x)); } // definition
double TrapRule(double a, double b, int N);
```

QuadratureV02.cpp (main)

```
int main(void )
{
  std::cout << "Using the TR to integrate in 1D\n";
  std::cout << "Integrate between x=0 and x=1.\n";
  double a=0.0, b=1.0;
  double Int_true_f = exp(1)-1; // for f(x)=exp(x)

  std::cout << "Enter value of N for the Trap Rule: ";
  int N;
  std::cin >> N; // Lazy! Should do input checking.

  double Int_TR_f = TrapRule(a,b,N);
  double error_f = fabs(Int_true_f - Int_TR_f);

  std::cout << "N=" << std::setw(6) << N <<
    ", Trap Rule=" << std::setprecision(6) <<
    Int_TR_f << ", error=" <<  std::scientific <<
    error_f << std::endl;
  return(0);
```

9. Numerical Integration Trapezium Rule as a function

QuadratureV02.cpp (function)

```cpp
double TrapRule(double a, double b, int N)
{
  double h=(b-a)/double(N);
  double QFn = (h/2.0)*f(a);
  for (int i=1; i<N; i++)
    QFn += h*f(a+i*h);
  QFn += (h/2.0)*f(b);
  return(QFn);
}
```