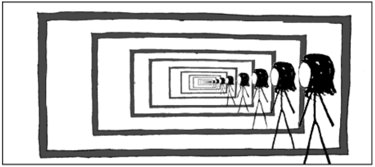
Annotated slides from 4pm class

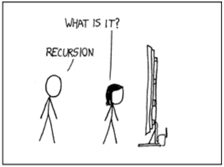
CS319: Scientific Computing

Quadrature, and Functions in C++

Dr Niall Madden

Week 4: **9am and 4pm** 31 January, 2024





Slides and examples: https://www.niallmadden.ie/2324-CS319

Class times

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

Reminder: **labs again this week** (and every week).

- Thursday 9-10
- Friday 11-12
- Friday 12-1.

For more, see https://www.niallmadden.ie/2324-CS319/#labs

Outline

Class times

9am: Section 1, 2, and 3.1

1. Overview of this week's classes
   - Why quadrature?
2. Functions (again)
   - Header
   - Function definition
   - E.g, Prime?
   - void functions
3. Numerical Integration
   - The basic idea

   Start here 4pm

   - The code
   - Trapezium Rule as a function
4. Functions as arguments to functions
5. Functions with default arguments
6. Pass-by-value
7. Function overloading
8. Detailed example

Slides and examples:
https://www.niallmadden.ie/2324-CS319

Overview of this week's classes

This week, we will study the use of functions in C++, which we started at the end of Week 3.

However, we'll motivate some of this study with a key topic in Scientific Computing: **Quadrature**, which is also known as **Numerical Integration**.

Later, we'll use this as an opportunity to study the idea of **experimental analysis** of algorithms.

Next week.

- A **Quadrature** method, in one dimension, is a method 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.
- They are methods for estimating integrals of functions. So this gives us two reasons to code functions:
  (i) As the functions we want to integrate;
  (ii) As the algorithms for doing the integration.

But before we get on to actual methods, we'll review some notes from last week, and examples I didn't get to.

Simplest common method: Trapezium Rule.

Functions (again)

In Week 3 we studied how to write functions in C++.

- Each function consists of two main parts: a **header** (or "prototype") and the function definition.
- The "header" is a single line of code that appears (usually) before the `main()` function. It 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 parameter list can include variable names, but they are treated as comments.
- The header line ends with a semicolon.

Syntax for function header:

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

Functions (again)

Examples:

A function that takes an int as argument and returns of float; and is called Convert:

```
float  Convert(int i);
```

A function that returns nothing, but takes as inputs a double, a char, and a string:

```
void  DoSomething(double x, char c, string s);
```

However, a function can't return more than one value:

```
float, float = ComputeTwoThings(int x)
```

is not allowed! But we can return arrays (next week).

Functions (again) — 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.

Since the variable names in the header. are optional or, at best, place-holders, they can be different in the function definition.

Functions (again)

Function definition

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
  - 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 `{...}`.

Functions (again)

00IsComposite.cpp (header)

```
// 00IsComposite.cpp
// An example of a simple function.
// Author: Niall Madden
// Date: 31 Jan 2024
// Week 4: CS319 - Scientific Computing

#include <iostream>

bool IsComposite(int i);
```

comments

function header.

Return type is "bool" which can be "true" or "false".
Takes a single argument (which is an int).

This will return "true" if the input is composite, and false otherwise (i.e., is not composite/prime).

Functions (again) E.g, Prime?

00IsComposite.cpp (main)

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

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

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

  return(0);
}
```

Here we use the ?: operator.

Functions (again)

E.g, Prime?

00IsComposite.cpp (function definition)

```
bool IsComposite(int i)
{
  int k;
  for (k=2; k<i; k++)
    if ( (i%k) == 0) if ("remainder on dividing i by k is zero)
      return(true); returns "true" AND exits function.

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

Functions (again)

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

`01Kth.cpp` (header)

```
// 01Kth.cpp:
// Another example of a simple function.
// Author: Niall Madden
// Date: 31 Jan Feb 2024
// Week 04: CS319 - Scientific Computing
#include <iostream>
void Kth(int i);  ← header
```

This function returns nothing, but will send one of "st", "nd", "rd" to cout.

Question: what is next in the sequence {s,t,n,d,r,d,t,h,t,h,...}

Functions (again)

01Kth.cpp (main)

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

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

  std::cout << "That is the ";
  Kth(i); // outputs "st", "nd", "rd" or "th"
  std::cout << " number." << std::endl;

  return(0);
}
```

Functions (again)

01Kth.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;
  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";
}
```

first digit is 1 but second is not 11.

Notice - no return value.

Numerical Integration

Numerical integration is an important topic in scientific computing. 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:

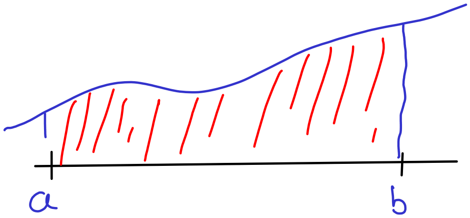
$$\int_a^b f(x)dx.$$

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

Numerical Integration — The basic idea

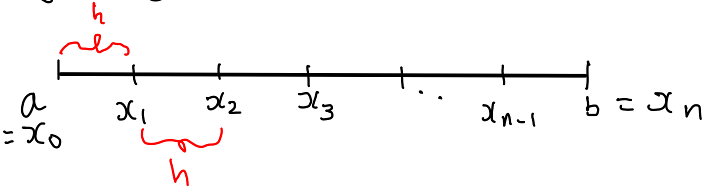
Idea $\int_a^b f(x)\,dx$ is the area between $f(x)$ and the x-axis, and between a & b.

Eg



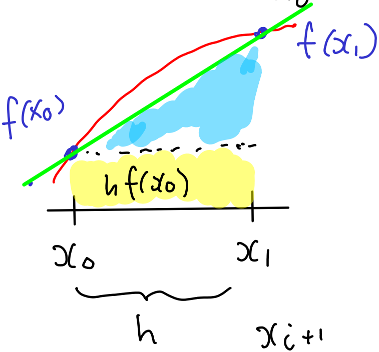
Idea

- Chose a natural number, n
- Divide $[a,b]$ into n intervals of length $h = \frac{b-a}{n}$



Numerical Integration

- Estimimate $\int_{x_0}^{x_1} f(x)\,dx$ as $\frac{h}{2}\left(f(x_0)+f(x_1)\right)$

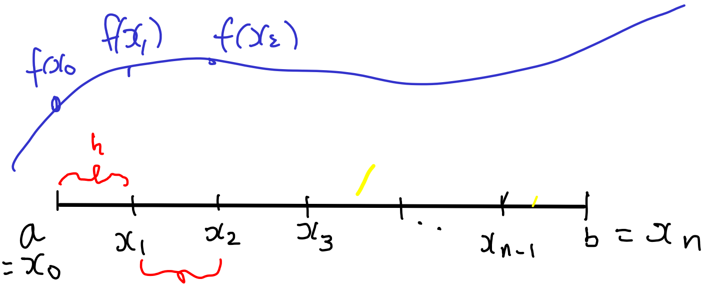


$$\int_{x_0}^{x_1} f(x) \approx h f(x_0) + \frac{h}{2}\left(f(x_1)-f(x_0)\right)$$
$$= \frac{h}{2}\left(f(x_0)+f(x_1)\right).$$

- Indeed $\int_{x_i}^{x_{i+1}} f(x)\,dx = \frac{h}{2}\left(f(x_i)+f(x_{i+1})\right).$

Numerical Integration

The basic idea



$$\int_a^b f(x)\,dx = \int_{x_0}^{x_1} f(x)\,dx + \int_{x_1}^{x_2} f(x)\,dx + \cdots + \int_{x_{n-1}}^{x_n} f(x)\,dx$$

$$= h\left(\tfrac{1}{2} f(x_0) + f(x_1) + f(x_2) + \cdots + \tfrac{1}{2} f(x_n)\right),$$

$$= \frac{h}{2} f(x_0) + h \sum_{i=1}^{n-1} f(x_i) + \frac{h}{2} f(x_n)$$

Finished here at 10am

02QuadratureV01.cpp (headers)

```
// 02QuadrateureV01.cpp:
// Trapezium Rule (TR) quadrature for a 1D function
// Author: Niall Madden
// Date: 31 Jan 2024
// Week 04: CS319 - Scientific Computing
#include <iostream>
#include <cmath>   // For exp()

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

f is the function we will integrate.

02QuadratureV01.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.
```

TR = Trapezium Rule

$$\int_0^1 e^x dx = e^x \Big|_0^1 = e^1 - e^0 \text{ "="} \texttt{ exp(1.0) - 1.0;}$$

Numerical Integration

The code

02QuadratureV01.cpp (main continued)

```
22    double h=(b-a)/double(N);
      double Int_f_TR = (h/2.0)*f(0.0);
24    for (int i=1; i<N; i++)
        Int_f_TR += h*f(a+i*h);
26    Int_f_TR += (h/2.0)*f(b);

28    double error = fabs(Int_f_true - Int_f_TR);

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

f(a) ~ $\frac{h}{2} f(x_0)$

22: N is an int, so we write (b-a)/double(N) so that it is temporarily converted to a floating point number. "Casting".

02QuadratureV01.cpp (main continued)

```
22    double h=(b-a)/double(N);
      double Int_f_TR = (h/2.0)*f(0.0);
24    for (int i=1; i<N; i++)
        Int_f_TR += h*f(a+i*h);
26    Int_f_TR += (h/2.0)*f(b);

28    double error = fabs(Int_f_true - Int_f_TR);

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

Line 25: note that $x_i = a + ih$

So this is $\sum_{i=1}^{n-1} h f(x_i)$.

Typical output:

```
N=10, Trap Rule=1.71971, error=0.00143166
N=20, Trap Rule=1.71864, error=0.00035796
N=40, Trap Rule=1.71837, error=8.94929e-05
```

It appears that, as N increases, the error decreases.

So, it works!!

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`~~. N.
- The function implements the Trapezium Rule for these values, and the globally defined `f`.

03QuadratureV02.cpp(header)

```
   // 03QuadrateureV03.cpp: Trapezium Rule as a function
 2 // Trapezium Rule (TR) quadrature for a 1D function
   // Author: Niall Madden
 4 // Date: 31 Jan Feb 2024
   // Week 04: CS319 - Scientific Computing
 6 #include <iostream>
   #include <cmath>   // For exp()
 8 #include <iomanip> // for, e.g., std::setprecision

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

Line 10: we've combined header & definition.

03QuadratureV02.cpp(main)

```cpp
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);
```

03QuadratureV02.cpp(function)

```
double TrapRule(double a, double b, int N)
{
  double h=(b-a)/double(N);
  double QFn = (h/2.0)*f(0.0);
  for (int i=1; i<N; i++)
    QFn += h*f(a+i*h);
  QFn += (h/2.0)*f(b);
  return(QFn);
}
```

a

compare with lines 22-26 in V01.

Functions as arguments to functions

We now have a function that implements the Trapezium Rule. However, it is rather limited, in several respects. This includes that the function, `f`, is hard-coded in the `TrapRule` function. If we want to change it, we'd edit the code, and recompile it.

Fortunately, it is relatively easy to give the name of one function as an argument to another.

The following example shows how it can be done.

Functions as arguments to functions

04QuadratureV04.cpp(header)

```
// 04QuadrateureV03.cpp: Trapezium Rule as a function
// that takes a function as argument
// Week 04: CS319 - Scientific Computing
#include <iostream>
#include <cmath>   // For exp()
#include <iomanip>

double f(double x) {  return(exp(x)); } // definition
double g(double x) {  return(6*x*x); } // definition

double TrapRule(double Fn(double), double a, double b,
                int N);
```

define f and g

Here "Fn" is a place-holder.

However, the 1st argument to TrapRule():
must be a function, must take a double as input,
and must return a double.

Functions as arguments to functions

04QuadratureV04.cpp (part of main())

```
  std::cout << "Which shall we integrate: \n"
            << "\t 1. f(x)=exp(x) \n\t 2. g(x)=6*x^2?\n";
  int choice;
  std::cin >> choice;
  while ( !(choice == 1 || choice  == 2) )
  {
    std::cout << "You entered " << choice
              <<". Please enter 1 or 2: ";
    std::cin >> choice;
  }
  double Int_TR=-1; // good place-holder
  if (choice == 1)
    Int_TR = TrapRule(f,a,b,10);
  else
    Int_TR = TrapRule(g,a,b,10);

  std::cout << "N=10" << ", Trap Rule="
            << std::setprecision(6) << Int_TR  << std::endl
  return(0);
```

\t = "tab"

not

OR

N=10;

Functions as arguments to functions

04QuadratureV04.cpp (TrapRule())

```
double TrapRule(double Fn(double), double a,
                double b, int N)
{
  double h=(b-a)/double(N); a
  double QFn = (h/2.0)*Fn(~~0.0~~);
  for (int i=1; i<N; i++)
    QFn += h*Fn(a+i*h);
  QFn += (h/2.0)*Fn(b);

  return(QFn);
}
```

Functions with default arguments

In our previous example, we wrote a function with the header

```
double TrapRule(double Fn(double), double a, double
b, int N);
```

And then we called it as

N
↓

```
Int_TR = TrapRule(f,a,b,10);
```

That is, when we were not particularly interested in the value of `N`, we took it to be 10.

It is easy to adjust the definition of the function so that, for example, if we called the function as

```
Int_TR = TrapRule(f,a,b);
```

it would just be assumed that $N = 10$. All we have to do is adjust the first ~~link~~ line in the function ~~definition~~ header/prototype.

Functions with default arguments

To do, this we specify the value of N in the **function prototype**. You can see this in `05QuadratureV04.cpp`. In particular, note Line 10:

`05QuadratureV04.cpp` (line 10)

```
10 double TrapRule(double Fn(double), double a,
                    double b, int N=10);
```

This means that, if the user does not specify a value of N, ~~ten~~ then it is taken that $N = 10$.

More precisely, if I don't specify the 4th argument, it is taken to be 10. (The name of that value, N, is just a place-holder, but is required).

Functions with default arguments

Important: Finished here **31/1/2024** at 5pm

- You can specify default values for as many arguments as you like. For example:

```
double TrapRule(double Fn(double), double a=0.0,
          double b=1.0, int N=10);
```

✓

- If you specify a default value for an argument, you must specify it for any following arguments. For example, the following would cause an error.

```
double TrapRule(double Fn(double), double a=0.0,
          double b=1.0, int N);
```

✗

Since a & b have default values, so too must N.

Pass-by-value

(We'll leave quadrature behind for a while, but will return later.)

In C++ we need to distinguish between

- a variable's (unique) memory address
- a variable's identifier (might not be unique) item the value stored in the variable.

The classic example is function that

- takes two `int`eger inputs, `a` and `b`;
- after calling the function, the values of `a` and `b` are swapped.

Pass-by-value

To understand this example, it is important to understand the difference between a

1. **local variable**, which belongs only to the function (or block) in which it is defined;
2. **global variable**, which belongs to the whole programme, and can be accessed in any function (or block).

(Global variables are very uncommon, but we'll have a look at them in some lab exercises).

Pass-by-value

06SwapByValue.cpp

```cpp
#include <iostream>
void Swap(int a, int b);

int main(void )
{
  int a, b;

  std::cout << "Enter two integers: ";
  std::cin >> a >> b;

  std::cout << "Before Swap: a=" << a << ", b=" << b
            << std::endl;
  Swap(a,b);
  std::cout << "After Swap: a=" << a << ", b=" << b
            << std::endl;

  return(0);
}
```

Pass-by-value

```
void Swap(int x, int y)
{
  int tmp;

  tmp=x;
  x=y;
  y=tmp;
}
```

This won't work.

We have passed only the *values stored in the variables a and b*. In the swap function these values are copied to local variables *x* and *y*. Although the local variables are swapped, they remained unchanged in the calling function.

What we really wanted to do here was to use **Pass-By-Reference** where we modify the contents of the memory space referred to by a and b. This is easily done...

Pass-by-value

...we just change the declaration and prototype from

```
void Swap(int x, int y)  // Pass by value
```

to

```
void Swap(int &x, int &y)  // Pass by Reference
```

the pass-by-reference is used.

Function overloading

C++ has certain features of **polymorphism**: where a single identifier can refer to two (or more) different things. A classic example is when two different functions can have the same name, but different argument lists.

This is called **function overloading**.

There are lots of reasons to do this. For example, just now we wrote a function called `Swap()` that swapped the value of two `int` variables. But suppose we wanted to write a function that swapped two `float`s, or two `string`s. Would we have to give a different name to each function? No!

Function overloading

As a simple example, we'll write two functions with the same name: one that swaps the values of a pair of `int`s, and that other that swaps a pair of `float`s. (Really this should be done with `template`s...)

07Swaps.cpp

```cpp
#include <iostream>

// We have two function prototypes!
void Swap(int &a, int &b);
void Swap(float &a, float &b);
```

Function overloading

07Swaps.cpp (continued)

```
int main(void) {
    int a, b;
    float c, d;

    std::cout << "Enter two integers: ";
    std::cin >> a >> b;
    std::cout << "Enter two floats: ";
    std::cin >> c >> d;

    std::cout << "a=" << a << ", b=" << b <<
      ", c=" << c << ", d=" << d << std::endl;
    std::cout << "Swapping ...." << std::endl;

    Swap(a,b);
    Swap(c,d);

    std::cout << "a=" << a << ", b=" << b <<
      ", c=" << c << ", d=" << d << std::endl;
    return(0);
```

Function overloading

07Swaps.cpp (continued)

```
void Swap(int &a, int &b)
{
  int tmp;

  tmp=a;
  a=b;
  b=tmp;
}

void Swap(float &a, float &b)
{
  float tmp;

  tmp=a;
  a=b;
  b=tmp;
}
```

Function overloading

What does the compiler take into account to distinguish between overloaded functions?

C++ takes the following into account:

- **Type of arguments**. So `void Sort(int, int)` is different from `void Sort(char, char)`.
- **The number of arguments**. So `int Add(int a, int b)` is different from `int Add(int a, int b, int c)`.

But not

- **Return values**. For example, we cannot have two functions `int Convert(int)` and `float Convert(int)` since they have the same argument list.
- **user-defined types** (using `typedef`) that are in fact the same. See, for example, `08OverloadedConvert.cpp`.

Detailed example

In the following example, we combine two features of C++ functions:

- Pass-by-reference,
- Overloading,

We'll write two functions, both called `Sort`:

- `Sort(int &a, int &b)` – sort two integers in ascending order.
- `Sort(int list[], int n)` – sort the elements of a list of length n.

The program will make a list of length 8 of random numbers between 0 and 39, and then sort them using **bubble sort**.

Detailed example

09Sort.cpp (i)

```
#include <iostream>
#include <stdlib.h>

const int N=8;

void Sort(int &a, int &b);
void Sort(int list[], int length);
void PrintList(int x[], int n);
```

Detailed example

09Sort.cpp (ii)

```
int main(void )
{
   int i, x[N];

   for (i=0; i<N; i++)
     x[i]=rand()%40;

   std::cout << "The list is:\t\t";
   PrintList(x, N);
   std::cout << "Sorting..." << std::endl;

   Sort(x,N);

   std::cout << "The sorted list is:\t";
   PrintList(x, N);
   return(0);
}
```

Detailed example

09Sort.cpp (iii)

```cpp
// Arguments: two integers
// return value: void
// Does: Sorts a and b so that a¡=b.
void Sort(int &a, int &b)
{
  if (a>b)
  {
    int tmp;
    tmp=a;  a=b;   b=tmp;
  }
}
```

Detailed example

09Sort.cpp (iii)

```
// Arguments: an integer array and its length
// return value: void
// Does: Sorts the 1st n elements of x
void Sort(int x[], int n)
{
  int i, k;
  for (i=n-1; i>1; i--)
    for (k=0; k<i; k++)
      Sort(x[k], x[k+1]);
}
```

Detailed example

```
void PrintList(int x[], int n)
{
  for (int i=0; i<n; i++)
    std::cout << x[i] << "  ";
  std::cout << std::endl;
}
```

Detailed example

Exercise (Simpson's Rule)

- *Find the formula for Simpson's Rule for estimating* $\int_a^b f(x)dx$.
- *Write a function that implements it.*
- *Compare the Trapezium Rule and Simpson's Rule. Which appears more accurate for a given* N.