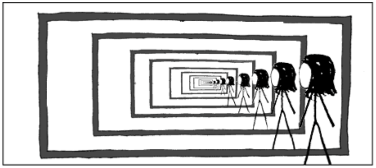
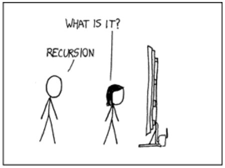
CS319: Scientific Computing

Functions and Quadrature

Dr Niall Madden

Week 4: 4th and 6th, February, 2026





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

0. Outline

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

1. Overview of this week's classes

Last week, we began studying the use of **functions** in C++.

We'll motivate some of this study with a key topic in Scientific Computing: **Quadrature**, which is also known as **Numerical Integration**. The concept was briefly introduced at the end of last Friday's class.

We'll also use this as an opportunity to study the idea of **experimental analysis** of algorithms.

. . . .

However, we are still studying how to write functions in C++. For each new concept, we'll write a new version of *`QuadratureV0x.cpp`*.

2. Numerical Integration (again)

We finished Week 3 by starting to develop an algorithm for estimating definite integrates of one-dimensional functions:

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

using one of the simplest methods: the Trapezium Rule.

Specifically, we learned that one can approximate

$$\int_{x_0}^{x_1} f(x)df \approx \frac{1}{2}(x_1 - x_0)\big(f(x_0) + f(x_1)\big).$$

The can be extended to give the following algorithm.

3. The Trapezium Rule algorithm

The Trapezium Rule

- Choose the number of intervals N, and set $h = (b - a)/N$.
- Define the quadrature points $x_0 = a$, $x_1 = a + h$, $\dots x_N = b$. In general, $x_i = a + ih$.
- Set $y_i = f(x_i)$ for $i = 0, 1, \dots, N$.
- Compute $Q_1(f) := h\big(\frac{1}{2}y_0 + \sum_{i=1}^{N-1} y_i + \frac{1}{2}y_N\big)$.

3. The Trapezium Rule algorithm

QuadratureV01.cpp (headers)

```
// QuadrateureV01.cpp:
// Trapezium Rule (TR) quadrature for a 1D function
// Author: Niall Madden
// Date: Feb 2026
// Week 04: CS319 - Scientific Computing
#include <iostream>
#include <cmath>   // For exp()

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

3. The Trapezium Rule algorithm

QuadratureV01.cpp (main)

```
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.
```

3. The Trapezium Rule algorithm

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);
}
```

Typical output:

4. V02: 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`.

4. V02: Trapezium Rule as a function

QuadratureV02.cpp (header)

```
// QuadrateureV02.cpp: Trapezium Rule as a function
// Trapezium Rule (TR) quadrature for a 1D function
// Author: Niall Madden
// Date: Feb 2026
// Week 04: 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);
```

4. V02: Trapezium Rule as a function

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);
```

4. V02: Trapezium Rule as a function

QuadratureV02.cpp (function)

```
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);
}
```

5. V03: 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.

5. V03: Functions as arguments to functions

QuadratureV03.cpp (header)

```
// QuadrateureV03.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);
```

5. V03: Functions as arguments to functions

QuadratureV03.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);
}
```

5. V03: Functions as arguments to functions

QuadratureV03.cpp (TrapRule())

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

  return(QFn);
}
```

6. V04: 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

```
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 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 function header.

6. V04: 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:

`QuadratureV04.cpp` (line 10)

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

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

6. V04: Functions with default arguments

Important:

- 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);
```

7. Pass-by-value

In C++ we need to distinguish between

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

In C++, if (say) `v` is a variable, then `&v` is the memory address of that variable.

We'll return to this at a later point, but for now we'll check the output of some lines of code that output a memory address.

7. Pass-by-value

01MemoryAddresses.cpp

```
int i=12;
std::cout << "main: Value stored in i: " << i << '\n';
std::cout << "main: address of i: " << &i << "\n\n";
Address(i);
std::cout << "main: Value stored in i: " << i << '\n';
```

Typical output might be something like:

```
main: The value stored in i  is 12
main: The address of i is 0x7ffcd1338314
```

7. Pass-by-value

When we pass a variable as an argument to a function, a new **copy** of the variable is made.

This is called **pass-by-value**.

Even if the variable has the same name in both `main()` and the function called, and the same value, they are different: the variables are **local** to the function (or block) in which they are defined.

We'll test this by writing a function that

- Takes a `int` as input;
- Displays its value and its memory address;
- Changes the value;
- Displays the new value and its memory address.

7. Pass-by-value

01MemoryAddresses.cpp

```
void Address(int i)
{
  std::cout << "Address: Value stored in i: " << i << '\n';
  std::cout << "Address: address of i: " << &i << '\n';
  i+=10; // Change value of i
  std::cout << "Address: New val stored in i: " << i << '\n';
  std::cout << "Address: address of i: " << &i << "\n\n";
}
```

7. Pass-by-value

Finally, let's call this function:

01MemoryAddresses.cpp

```
int i=12;
std::cout << "main: Value stored in i: " << i << '\n';
std::cout << "main: address of i: " << &i << "\n\n";
Address(i);
std::cout << "main: Value stored in i: " << i << '\n';
std::cout << "main: address of i: " << &i << '\n';
```

7. Pass-by-value

In many case, "pass-by-value" is a good idea: a function can change the value of a variable passed to it, without changing the data of the calling function.

But sometimes we **want** a function to be able to change the value of a variable in the calling function. (Another important case use is if that data is stored in a very large array which we don't want to duplicate).

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.

7. Pass-by-value

02SwapByValue.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); // ⟵ how to code this ??
  std::cout << "After Swap: a=" << a << ", b=" << b
            << std::endl;

  return(0);
}
```

7. 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...

7. 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.

We'll do that presently, but fist an example of the effect of using &...

7. Pass-by-value

Example

03PassByValueAndReference.cpp

```
void DoesNotChangeVar(int X);
void DoesChangeVar(int &X);

int main(void)
{
  int q=34;
  std::cout << "main: q=" << q << std::endl;
  std::cout << "main: Calling DoesNotChangeVar(q)...";
  DoesNotChangeVar(q);
  std::cout << "\t Now q=" << q << std::endl;
  std::cout << "main: Calling DoesChangeVar(q)...";
  DoesChangeVar(q);
  std::cout << "\t And now q=" << q << std::endl;
  return(0);
}

void DoesNotChangeVar(int X){  X+=101; }
void DoesChangeVar(int &X){  X+=101; }
```

7. Pass-by-value

Output

8. 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, we earlier had a function called `Swap()` that swapped the value of two `int` variables.

However, we can write a function that is also called `Swap()` to swap two `float`s, or two `string`s.

(Note: this can also be done with something called `templates`: we'll look at that in a few weeks.)

8. 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...)

`04Swaps.cpp` (headers)

```cpp
#include <iostream>

// We have two function prototypes with same name!
void Swap(int &a, int &b);   // note use of references
void Swap(float &a, float &b);
```

8. Function overloading

04Swaps.cpp (main)

```
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);
}
```

8. Function overloading

04Swaps.cpp (functions)

```
// Swap(): swap two ints
void Swap(int &a, int &b)
{
  int tmp;

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

}

// Swap(): swap two floats
void Swap(float &a, float &b)
{
  float tmp;

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

8. Function overloading

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

C++ distinguishes functions according to their **signature**. A signature is made up from:

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

Examples:

8. Function overloading

However, the following to not impact signatures:

- **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, `02OverloadedConvert.cpp`.
- **References**: we cannot have two functions
  `int MyFunction(int x)` and
  `int MyFunction(int &x)`

8. Function overloading

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**.

8. Function overloading

05Sort.cpp (headers)

```
#include <iostream>
#include <stdlib.h> // contains rand() header

const int N=8;

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

8. Function overloading

05Sort.cpp (main)

```cpp
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);
}
```

05Sort.cpp (Sort two ints)

```
// Sort(a, b)
// 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;
  }
}
```

05Sort.cpp (Sort list)

```cpp
// Sort(int [], int)
// Arguments: an integer array and its length
// return value: void
// Does: Sorts the first 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]);

}
```

8. Function overloading

Detailed example

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

9. Exercises

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?

Exercise

Change the `Address()` *function in* `01MemoryAddresses.cpp` *so that the variable* `i` *is passed by reference.*
How does the output change?