

## CS319: Scientific Computing

### Week 8: Projects, Quadrature in 2D; Intro to Classes

Projects!

Dr Niall Madden

9am and 4pm, 28 February, 2024



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

# Outline

- 1 Projects!
- 2 Quadrature 2D
  - Trapezium Rule in 2D
- 3 Lab 6 preview
- 4 Encapsulation
- 5 `class`
  - Example – a stack
  - `class`
- 6 Constructors
- 7 Destructors
  - The Constructor again...

Slides and examples:

<https://www.niallmadden.ie/2324-CS319>



**Notes for this part are at:**

[https://www.niallmadden.ie/2324-CS319/  
2324-CS319-Projects.pdf](https://www.niallmadden.ie/2324-CS319/2324-CS319-Projects.pdf)

# Quadrature 2D

(These slides were part of Week 7, but I didn't get to them in class).

For the last time (in lectures) we'll look at **numerical integration**, this time of two dimensional functions.

That is, our goal is to estimate

$$\int_{a_1}^{b_1} \int_{a_2}^{b_2} f(x_1, x_2) dx_1 dx_2.$$

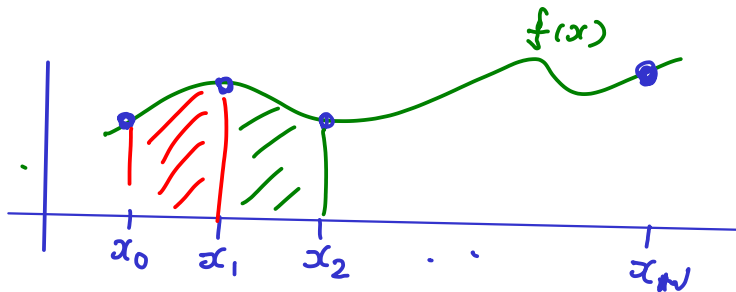
When we implement an algorithm for this, we will set

- ▶ **x1** and **x2** to be vectors of (one-dimensional) quadrature of  $N + 1$  points.
- ▶ **y** to be a **two-dimensional** array of  $(N + 1)^2$  quadrature values. That is, we will set  
`y[i][j] = f(x1[i], x2[j]);`

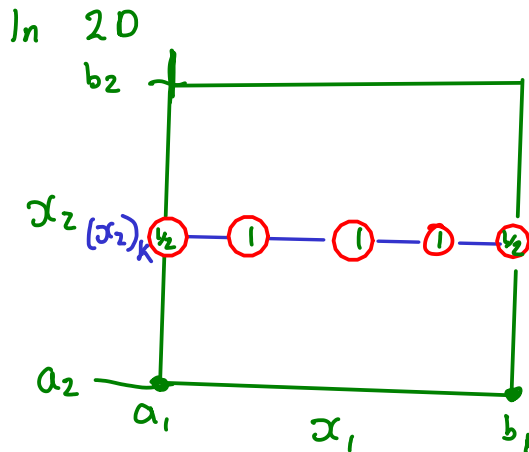
# Derivation

Recall trapezium Rule in 1D

$$\int_a^b f(x) dx \approx (b-a) \left( \frac{1}{2} f(x_0) + \sum_{i=1}^N f(x_i) + \frac{1}{2} f(x_N) \right)$$



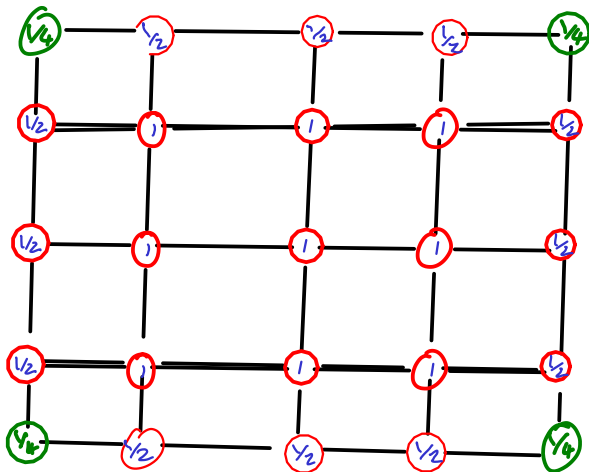
## Derivation



$$\int f(x_1, (x_2)_k) dx_1$$

$$\approx (b_1 - a_1) \left( \frac{1}{2} f((x_1)_0, (x_2)_k) + \sum_{i=1}^N f((x_1)_i, (x_2)_k) + \frac{1}{2} f((x_1)_N, (x_2)_k) \right)$$

## Derivation



Method:

$$\int_{a_1}^{b_1} \int_{a_2}^{b_2} f(x_1, x_2) dx_1 dx_2$$

$$\approx (b_1 - a_1)(b_2 - a_2)$$

$$\frac{1}{4} ("f \text{ at corners}")$$

$$+ \frac{1}{2} ("f \text{ at edges}")$$

$$+ ("f \text{ at other points"})$$

## Implementation

We'll implement this for estimating  $\int_0^1 \int_0^1 e^{x_1+x_2} dx_1 dx_2$ , with  $N$  quadrature points in each direction.

00Trap2D.cpp preamble

```
10 double f(double x1, double x2) { return(exp(x1+x2)); }  
   double ans_true = pow(exp(1.0)-1.0,2); // true value  
14 double Trap2D(double *x1, double *x2,  
                double **y, unsigned int N);
```



## 00Trap2D.cpp main()

```
16 int main(void )
17 {
18     unsigned N = pow(2,4); // Number of points in each direction
19     double a1=0.0, b1=1.0, a2=0.0, b2=1.0; // limits of int
20     double h1, h2; // step-size in x1 and x2
21     double *x1, *x2, **y; // quadrature points and values.
22
23     x1 = new double[N+1];
24     x2 = new double[N+1];
25
26     h1 = (b1-a1)/double(N);
27     h2 = (b2-a2)/double(N);
28     for(unsigned i = 0; i < N+1; i++)
29     {
30         x1[i] = a1+i*h1;
31         x2[i] = a2+i*h2;
32     }
```

00Trap2D.cpp main() continued

```
34  y = new double * [N+1];  
    for(unsigned i = 0; i < N+1; i++)  
36      y[i] = new double[N+1];  
  
38  for (unsigned i=0; i<N+1; i++)  
    for (unsigned j=0; j<N+1; j++)  
40      y[i][j] = f(x1[i], x2[j]);  
  
42  double est1 = Trap2D(x1, x2, y, N);  
    double error1 = fabs(ans_true - est1);  
  
46  std::cout << "N=" << N << " | est=" << est1  
      << " | error = " << error1 << std::endl;
```

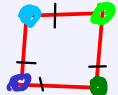
20 - DMA.

## 00Trap2D.cpp Trap2D()

```

50 double Trap2D(double *x1, double *x2, double **y,
51               unsigned N)
52 {
53     double Q, h1 = (x1[N]-x1[0])/double(N),
54             h2 = (x2[N]-x2[0])/double(N);
55
56     Q = 0.25*(f(x1[0],x2[0]) + f(x1[N],x2[0]) // 4 corners
57             + f(x1[0],x2[N]) + f(x1[N],x2[N]));
58
59     for (unsigned k=1; k<N; k++) // 4 edges (not including corners)
60         Q += 0.5*(f(x1[k],x2[0]) + f(x1[k],x2[N])
61                 + f(x1[0],x2[k]) + f(x1[N],x2[k]));
62
63     for (unsigned i=1; i<N; i++) // All the points in the interior
64         for (unsigned j=1; j<N; j++)
65             Q += f(x1[i],x2[j]);
66
67     Q *= h1*h2;
68     return(Q);

```



## Lab 6 preview

- ▶ Implement Simpson's Rule in 1D and 2D;
- ▶ Verify convergence using Python/NumPy/Jupyter.
- ▶ Compare with Monte Carlo(?)

Finished here at 10am

## Encapsulation

**Idea:** create a single entity in a program that combines data with the program code (i.e., functions) that manipulate that data. In C++, a description/definition of such entities is called a **class**, and an instance of such an entity is called an **object**.

That is, like a variable is a single instance for a **float** (for example), then an object is a single instance of a **class**.

A class should be thought of as an **Abstract Data Type** (ADT): a specialised type of variable that the user can define.

There are many important examples of “built-in” C++ classes, such as **string**, and objects, such as **cin** and **cout**. But we’ll leave those until later, and first study how to make our own.

# Encapsulation

*The next bit is really important: not just to C++, but for writing robust scientific computing code.*

Within an object, code and data may be either

- ▶ **Private:** accessible only to another part of that object, or
- ▶ **Public:** other parts of the program can access it even though it belongs to a particular object. The public parts of an object provide an **interface** to the object for other parts of the program.

It is referred to a **“data hiding”**, an important concept in software design.

In C++, *encapsulation* is implemented using the `class` keyword. The example we'll consider is a **stack** – a *LIFO* (Last In First Out) queue.

.....

*There is already a C++ implementation of a **stack**. It is part of the **Standard Template Library (STL)**. We reinvent the wheel here only because it is a nice example that includes most of the key concepts associated with classes in C++. We will study the STL later in CS319 (maybe ...)*



The name of our class will be `MyStack`. It will permit two primary operations:

- ▶ an item may be added to the top of the stack: `push()` ;
- ▶ an item may be removed from the top of the stack: `pop()`.

These then are our interfaces to the stack. Hence these will be **public**.



For the stack itself, the following must be maintained:

- ▶ an array containing the items in the contents;
- ▶ a counter/index to the top of the stack.

These are *private* to the class.

We choose this example because it is obvious that

- ▶ *push()* and *pop()* are the interfaces to the object—they are declared as *public*;
- ▶ the contents of the stack, and the counter of the number of objects in it, need only be visible to the object itself; hence they are *private*.

In our example there is also a public function to initialise the stack.

The basic syntax for defining a class:

```
class class-name {  
private:  
    ...    // private functions and variables  
public:  
    ...    // public functions and variables  
};
```

*class-name* becomes a new object type—one can now declare objects to be of type *class-name*.

This is only a declaration. Therefore,

- ▶ functions are not defined, though the prototype is given,
- ▶ variables are declared but are not initialised,
- ▶ the declaration block is delineated by `{` and `}`, and terminated with a semicolon.

As mentioned our class has two private members

- ▶ `contents`: a `char` array of length `MAX_STACK` the array containing the stacked items.
- ▶ `top`: an `int` that stores the number of items on the stack.

It has three public member functions: (= "methods")

- (a) `init()` sets the stack counter to 0. No arguments or return value.
- (b) `push()` adds an item to the stack. One argument: the character to be added.
- (c) `pop()` takes no argument but returns the removed item.

```
class MyStack {  
private:  
    char contents[MAX_STACK];  
    int top;  
public:  
    void init(void );  
    void push(char c);  
    char pop(void );  
};
```

} we could include the  
code for these functions,  
here, but will do that

outside of the  
class definition (this time).

push(c) will add "c" to the top of  
the stack

x = pop() will remove the top item from  
the stack and store it in x.

To define the functions associated with a particular class we use

1. the name of the class, followed by
2. the *scope resolution operator* `::` , followed by
3. the name of the function.

We now define the three (public) functions: `init()`, `push()` and `pop()`.

The `init()` is required only to set the value of `top` to zero:

```
void MyStack::init(void)
{
    top=0;
}
```

General  
classname `::` member()

Note that we didn't have to declare the (private) variable `top`.

The `push()` function takes as its only argument a single character. It adds the character to the stack and increments the index to the top of the stack.

```
void MyStack::push(char c) {  
    contents[top]=c;  
    top++;  
}
```

*ie top = top + 1 .*

The `pop()` function doesn't take any arguments ( `void`). It removes the item from the stack by returning the top entry and decrementing `top`.

```
char MyStack::pop(void){  
    top--;  
    return(contents[top]);  
}
```

The first item in the stack is at position `0`, the second is a position `1`, the 3rd is at position `2`, etc. So when `top=n` then there are `n` items in the stack but the top one is actually located in `contents[n-1]`.

Now that our class `MyStack` has been declared, and its functions defined, we can declare objects to be of type `MyStack`, e.g.,

```
MyStack s1, s2;
```

We can refer to the functions `s1.pop()` and `s2.push(c)`, say, because these are public members of the class. We cannot refer to `s1.top` as this variable is private to the class and is hidden from the rest of the program.

.....

To use the objects, we could have a `main()` function that behaves as follows:

- ▶ Declare and initialise a `MyStack` object `s`;
- ▶ Push the characters `'C', 'S', '3', '1', '9'` onto the stack;
- ▶ The stack's contents are popped and output to the console using `cout`.



## 01MyStack.cpp

```
int main(void ) {  
38     MyStack s;  
  
40     s.init();  
  
42     s.push('C');  
    s.push('S');  
44     s.push('3');  
    s.push('1');  
46     s.push('9');  
  
48     std::cout << "Popping ... " << std::endl;  
    std::cout << s.pop() << std::endl;  
50     std::cout << s.pop() << std::endl;  
    std::cout << s.pop() << std::endl;  
52     std::cout << s.pop() << std::endl;  
    std::cout << s.pop() << std::endl;  
54     return (0);  
}
```

# Constructors

Suppose we wanted to change the `MyStack` class so that the user can choose the maximum number of elements on the stack...

In the example above, the function `init()` is used explicitly to initialise the variable `top`. However, there is an initialisation mechanism called a **Constructor** that is built into the concept of a class.

## CONSTRUCTOR

A **Constructor** is a public member function of a class

- ▶ that shares the same name as the class, and
- ▶ is executed whenever a new instance of that class is created.

# Constructors

Constructors may contain any code you like; but it is good practice to only use them for initialization.

As an example, we'll change the declaration of the `stack` class as shown here:

```
class MyStack {  
public:    ← replaces init()  
    MyStack(void); // Constructor. No return type  
    void push(char c);  
    char pop(void );  
private:  
    char contents[MAX_STACK];  
    int top;  
};
```

# Constructors

We then replace the `init()` function with:

```
MyStack::MyStack(void )  
{  
    top=0;  
}
```

*Note that the constructor has no explicit return type.*

Now whenever an object of type `MyStack` is created, e.g., with

`MyStack s,`

the function `s.MyStack()` is called automatically – and `s.top` is set to zero.

# Constructors

We now make the following modifications to the `stack` implementation (for full implementation, see `02MyStackConstructor.cpp`)

```
class MyStack {  
private:  
    char *contents;  
    int top, maxsize;  
public:  
    MyStack (void);  
    MyStack (unsigned int StackSize);  
    void push(char c);  
    char pop(void );  
};
```

Two constructors

Here we have changed *contents* so that it is a pointer.

# Constructors

Code for the constructor (v1 - with no argument).

```
MyStack::MyStack(void)
{
    contents = new char [MAX_STACK];
    top=0;
}
```

Using  
dynamic  
memory  
allocation.

(not quite the version in the  
program: will fix later!).

# Destructors

Complementing the idea of a constructor is a **destructor**. This function is called

- ▶ for a local object – whenever it goes out of scope,
- ▶ for a global object – when the program ends.

The name of the destructor is the same as the class, but preceded by a tilde:

```
class MyStack {  
private:  
    char *contents;  
    int top;  
public:  
    MyStack(void );  
    ~MyStack(void );  
    void push(char c);  
    char pop();  
};
```

# Destructors

```
MyStack::~~MyStack()  
{  
    delete [] contents;  
}
```

This deallocates memory that had been allocated, using `new`, in the constructor.



The example we had earlier of a constructor was particularly basic, not least because its parameter list is `void`. More commonly, one passes arguments to the constructor that can be used, e.g.,

- ▶ to set the value of a data member;
- ▶ dynamically size an array using `new`.

However, one should still provide a default constructor (i.e., one with no arguments), or one with a default argument list.

1

```
class MyStack
{
private:
    char *contents;
    int top;
public:
    MyStack(void);
    MyStack(unsigned SkSize);
    void push(char c);
    char pop(void );
};
```

```
MyStack::MyStack(void)
{
    top=0;
    contents = new char[MAX_STACK];
}

MyStack::MyStack(unsigned SkSize)
{
    top=0;
    contents = new char[StackSize];
}
```

---

<sup>1</sup>This is for illustration. Better again: use one constructor, but with a default argument value.