CS319: Scientific Computing

Multidimensional Arrays; Introduction to Classes

Dr Niall Madden

Week 7: 26 + 28 February, 2025

Slides and examples: https://www.niallmadden.ie/2425-CS319

0. Notices

1. Sorry - still have not graded Lab 2. Soon!
2. Will also grade the Class Test by next week.
3. Grades for Lab 4: will be confirmed once demo'ed in a lab.
   You need to attend a lab to get a non-zero grade.

4: Projects - start next week.

0. Outline

Slides and examples: https://www.niallmadden.ie/2425-CS319

So far in CS319, we have worked with **one-dimensional** arrays. For example, if we wanted to store a set of **five** `int`egers, we could declare an array:

```
int v[5];
```

We could then access the five elements:
`v[0]`, `v[1]`, `v[2]`, `v[3]`, and `v[4]`.

This is a **one-dimensional array**: the array has a single index. It is similar to the idea of a **vector** in Mathematics.

However, often we will have table/rectangles of data, in a way that is similar to a **matrix**.

In C++, a two-dimensional $M \times N$ array of (say) `double`s can be declared as:

```
double A[M][N];
```

Then its members are

$$\begin{pmatrix} \text{A[0][0]} & \text{A[0][1]} & \text{A[0][2]} & \cdots & \text{A[0][N-1]} \\ \text{A[1][0]} & \text{A[1][1]} & \text{A[1][2]} & \cdots & \text{A[1][N-1]} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \text{A[M-1][0]} & \text{A[M-1][1]} & \text{A[M-1][2]} & \cdots & \text{A[M-1][N-1]} \end{pmatrix}$$

For example, `double A[3][4];`
gives a 2D array

$$\begin{pmatrix} \text{A[0][0]} & \text{A[0][1]} & \text{A[0][2]} & \text{A[0][3]} \\ \text{A[1][0]} & \text{A[1][1]} & \text{A[1][2]} & \text{A[1][3]} \\ \text{A[2][0]} & \text{A[2][1]} & \text{A[2][2]} & \text{A[2][3]} \end{pmatrix}$$

DMA

Dynamic memory allocation in 2D is a little complicated, because a 2D array is actually just an "array of arrays".

This is because, we declare, for example,
`double A[3][4];` what really happens is:

- `A` is assigned the base address of three **pointers**: `A[0]`, `A[1]`, `A[2]`.
- Each of those is a base address for a (1D) array of 4 doubles.

This approach has advantages: because of it the language can support arrays in as many dimensions as one would like.

But it makes DMA more complicated.

To use dynamic memory allocation to reserve memory for a two-dimensional $M \times N$ matrix of doubles (for example):

(i) ▶ declare a “pointer to pointer to double”

(ii) ▶ use new to assign memory for M pointers;

(iii) ▶ for each of those, assign memory for N doubles.

Code: ((double *) *) A

```
(i)   double **A;
(ii)  A = new double* [M];
(iii) for (int i=0; i<M; i++)
          A[i] = new double [N];
```

If we dynamically allocate memory for a 2D array, we need to de-allocate it too, using the `delete` operator (See Week 6).

If the array `A` as been allocated as on the previous slide, it is de-allocated as:

```
for (int i=0; i<M; i++)
      delete[] A[i];
delete[] A;
```

deallocate memory for each of A[0], A[1], A[2],

deallocate memory allocated in Step (ii)

undoing the allocation from step (iii) of the previous slide.

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

In 1D, the Trap Rule works as:

$$\int_a^b f(x)\,dx \approx h\left(\tfrac{1}{2}f(x_0) + \sum_{i=1:N} f(x_i) + \tfrac{1}{2}f(x_N)\right)$$

Extend this to 2D:

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

Derivation

We choose N and set

$$h = \frac{b_1 - a_1}{N} = \frac{b_2 - a_2}{N}$$

(for simplicity we are assuming the domain is a square, so $b_1 - a_1 = b_2 - a_2$).

Derivation

Then our quadrature points are

In x_1: $(x_1)_0 = a_1$, $(x_1)_1 = a_1 + h$

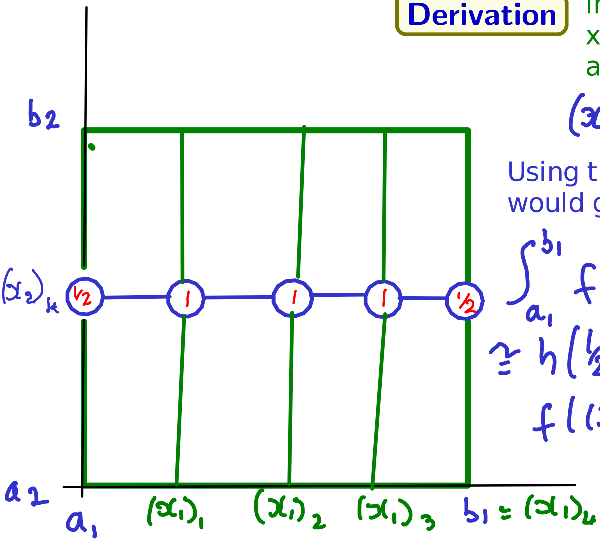
$(x_1)_2 = a_1 + 2h$, $\dots$ $(x_1)_N = a_1 + Nh = b_1$

Similarly in x_2 we have points $(x_2)_k = a_2 + kh$.

Derivation

Suppose first we wanted to integrate f(x1,x2) in the x-direciton only, along a single line at the point :

$(x_2)_k$.



Using the rule from earlier, we would get

$$\int_{a_1}^{b_1} f(x_1, (x_2)_k)\, dx_1$$
$$\approx h\Big(\tfrac{1}{2} f((x_1)_0, (x_2)_k) + f((x_1)_1, (x_2)_k) + f((x_1)_2, (x_2)_k) + f((x_1)_3, (x_2)_k) + \tfrac{1}{2} f((x_1)_4, (x_2)_k)\Big)$$

Derivation

More generally

$$\int_{a_1}^{b_1} f(x_1, (x_2)_k)\, dx \approx$$

$$h\left(\frac{1}{2} f(x_1, (x_2)_0) + \sum_{k=1:N-1} f(x_1, (x_2)_k) + \frac{1}{2} f(x_1, (x_2)_N)\right)$$

Derivation

Now apply the trapezium Rule in the x_2 direction:

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

$$h\cdot\Big(\frac{1}{2}\int_{a_1}^{b_1} f(x_1, (x_2)_0)\,dx_1 +$$

$$\sum_{k=1:N-1}\int_{a_1}^{b_1} f(x_1, (x_2)_k)\,dx_1 + \frac{1}{2}\int_{a_1}^{b_1} f(x_1, (x_2)_N)\,dx_1\Big)$$

Derivation

Applying the TR to each integral we finally get:

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

$$h^2 \Big[\tfrac{1}{4} f((x_1)_0, (x_2)_0) + \tfrac{1}{2} \sum_{k=1:N-1} f((x_1)_k, (x_2)_0) + \tfrac{1}{4} f((x_1)_N, (x_2)_N)$$

$$+ \tfrac{1}{2} f((x_1)_0, (x_2)_1) + \sum_{k=1:N-1} f((x_1)_k, (x_2)_1) + \dots \Big]$$

Derivation

Often expressed as

h^2 [$\frac{1}{4}$ ("f at corners") +

$\frac{1}{2}$ ("f at edges but not corners")

+ "f at all points in the interior].

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

```
double f(double x1, double x2) {  return(exp(x1+x2)); }
double ans_true = pow(exp(1.0)-1.0,2); // true value

double Trap2D(double *x1, double *x2,
              double **y, unsigned int N);
```

$(e^1-1)^2$

00Trap2D.cpp main()

```
int main(void )
{
  unsigned N = pow(2,4);        // Number of points in each direction
  double a1=0.0, b1=1.0, a2=0.0, b2=1.0;          // limits of int
  double h1, h2;                // step-size in x1 and x2
  double *x1, *x2, **y;         // quadrature points and values

  x1 = new double[N+1];
  x2 = new double[N+1];

  h1 = (b1-a1)/double(N);
  h2 = (b2-a2)/double(N);
  for(unsigned i = 0; i < N+1; i++)
  {
    x1[i] = a1+i*h1;
    x2[i] = a2+i*h2;
  }
```

Pointers, because we'll use DMA.

$x1[i) = a_1 + i h_i$ eg $x1[0]=0$, $x1[N]= b_1$

00Trap2D.cpp main() continued

```
y = new double * [N+1];
for(unsigned i = 0; i < N+1; i++)
  y[i] = new double[N+1];

for (unsigned i=0; i<N+1; i++)
  for (unsigned j=0; j<N+1; j++)
    y[i][j] = f(x1[i], x2[j]);

double est1 = Trap2D(x1, x2, y, N);
double error1 = fabs(ans_true - est1);

std::cout << "N=" <<  N << " | est=" << est1
          << " | error = " << error1 << std::endl;
```

memory allocation for y.

assign values for y.

00Trap2D.cpp main() last part

```
  // De-allocate memory
  delete [] x1;
  delete [] x2;
  for(unsigned i = 0; i < N+1; i++)
    delete [] y[i];
  delete [] y;

  return(0);
}
```

`00Trap2D.cpp` Trap2D()

```
double Trap2D(double *x1, double *x2, double **y,
              unsigned N)
{
  double Q, h1 = (x1[N]-x1[0])/double(N),
    h2 = (x2[N]-x2[0])/double(N);

  Q = 0.25*(y[0][0] + y[N][0]   // 4 corners
          + y[0][N] + y[N][N]);

  for (unsigned k=1; k<N; k++)   // 4 edges (not including corners)
    Q += 0.5*(y[k][0] + y[k][N]
            + y[0][k] + y[N][k]);

  for (unsigned i=1; i<N; i++) // All the points in the interior
    for (unsigned j=1; j<N; j++)
      Q += y[i][j];

  Q *= h1*h2;
  return(Q);
}
```

$Q = (h_1)(h_2)Q$

Q for Quadrature

3. Preview of Labs 5 and 6

- **Lab 5** (this week)
  - Implement Simpson's Rule and Boole's Rule in 1D;
  - Verify convergence using Python/NumPy/Jupyter.
- **Lab 6** (next week)
  - Extend Simpson's Rule to 2D;
  - Compare with Monte Carlo.

Finished here Wed at 5pm

Encapsulation (aka Object Oriented Programming)

Idea: create a single entity in a program that combines data with the program code (i.e., functions) which 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.

In **object oriented programming** the classes we define have two types of components:

- **data members** (also called *fields*, *attributes* or *properties*). These are usually variables of arrays.
- **function members**, which are also called **methods**. These are used to manipulate the data members.

Note: In CS319 when we say "**method**" we always mean a function which is a member of a class.

4. Encapsulation

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

Within an object, methods and data members 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.*

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 methods 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 methods and variables
public:
    ...      // public methods and variables
};
```

Class_name is an identifier, and follows the usual rules for names.

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:

(a) `init()` sets the stack counter to 0. No arguments or return value. (Later will be replaced by a constructor).

(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.

`01MyStack.cpp` preamble

```
#define MAX_STACK 10

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

places c onto the stack

pops a character off the stack & returns it.

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:

`01MyStack.cpp` : `init()`

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

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.

01MyStack.cpp : push()

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

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

01MyStack.cpp : pop()

```
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 : main()

```
int main(void ) {
  MyStack s;

  s.init();

  s.push('C');
  s.push('S');
  s.push('3');
  s.push('1');
  s.push('9');

  std::cout << "Popping ... " << std::endl;

  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;
  std::cout << s.pop() << std::endl;

  return (0);
}
```

6. 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 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:
  MyStack(void); // Constructor. No return type
  void push(char c);
  char pop(void);
private:
  char contents[MAX_STACK];
  int top;
};
```

instead of init()

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

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

Note that the constructor as no explicit return type.

Now whenever an objects of type `MyStack` is created, e.g., with
`MyStack s;`
the function `s.MyStack()` is called automatically – and *s.top* is set to zero.

Note: this and subsequent slides are somewhat different from those presented in class.

Complementing the idea of a **constructor**, we have a **destructor**: that is a function that is automatically called when ...

- 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.

However, for the versions of the `MyStack` class we've seen so far, there is nothing for a destructor to do. So, we'll have one final version of this class.

7. MyStack (Final version)

MyStack: final version

We now make some final modifications to the MyStack class. The main idea is to allow for the stack to have a maximum size that is chosen by the user, which makes it more flexible (previously it was of size MAX_STACK=10, which is very limiting.

To do that, we'll have to use **Dynamic Memory Allocation**.

Moreover, this allows us to demonstrate two things:

(a) We can have multiple constructors. In this case, one will be of default size, and the other user-chosen;

(b) A destructor, which is responsible for deallocation of memory.

For full implementation, see 02MyStackConstructor.cpp.

Here is the new version of the class definition. Note:

- `contents` is now a pointer;
- there are two `MyStack()` methods
- the new `~MyStack` method

```
class MyStack {
private:
  char *contents;
  int top;
public:
  MyStack(void );    // default constructor
  MyStack (unsigned int StackSize);
  ~MyStack(void );   // destructor
  void push(char c);
  char pop();
};
```

Code for the constructors[1]

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

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

[1] This is for illustration. Better again: use one constructor, but with a default argument value.

7. MyStack (Final version) — The destructor

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