CS319: Scientific Computing

Multidimensional Arrays

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**.

0. Outline

1. Two-dimensional arrays
   - Recall: 1D
   - 2D arrays
   - 2D DMA
   - Deallocation
2. Quadrature in 2D
   - Trapezium Rule in 2D
3. Preview of Labs 5 and 6
4. Encapsulation
5. `class`
   - Example – a stack
   - `class`
6. Constructors
7. Destructors
8. The Constructor again...

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

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

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

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

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):

- declare a "pointer to pointer to double"
- use `new` to assign memory for M pointers;
- for each of those, assign memory for N doubles.

Code:

```
double **A;
A = new double* [M];
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;
```

2. Quadrature in 2D

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

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

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

00Trap2D.cpp main()

```
16 int main(void )
   {
18   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
20   double h1, h2;               // step-size in x1 and x2
     double *x1, *x2, **y;        // quadrature points and values

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

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

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

00Trap2D.cpp main() last part

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

      return(0);
56  }
```

00Trap2D.cpp Trap2D()

```
58 double Trap2D(double *x1, double *x2, double **y,
                unsigned N)
60 {
     double Q, h1 = (x1[N]-x1[0])/double(N),
62     h2 = (x2[N]-x2[0])/double(N);

64   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)
68     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
72     for (unsigned j=1; j<N; j++)
         Q += y[i][j];

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

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.

4. Encapsulation

Encapsulation

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.

4. 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.*

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:

(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.

01MyStack.cpp preamble

```
12 class MyStack {
   private:
14   char contents[MAX_STACK];
     int top;
16 public:
     void init(void );
18   void push(char c);
     char pop(void );
20 };
```

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()

```
22 void MyStack::init(void) {
     top=0;
24 }
```

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()

```
26 void MyStack::push(char c) {
     contents[top]=c;
28   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()

```
32 char MyStack::pop(void) {
     top--;
     return(contents[top]);
34 }
```

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()

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

     s.init();

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

     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;

     return (0);
56 }
```

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.

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

6. Constructors

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.

6. 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 );
};
```

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

6. Constructors

Code for the constructor.

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

7. 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();
};
```

7. Destructors

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

8. The Constructor again...

The example we had earlier of a constructor was particularly basic, not least because is 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.

8. The Constructor again...

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];
}
```

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