CS319: Scientific Computing

Arrays, Memory Allocation (and quadrature again)

Dr Niall Madden

Week 6: 11th and 13th of February, 2026

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

0. The array of topics this week:

1. Arrays
   - Basics
   - Initialization
   - Base Address
2. Timing Code
   - ctime
   - Example
   - chrono
3. Pointers
   - Pointer arithmetic
   - Warning!
4. Dynamic Memory Allocation
   - `new`
   - `delete`
5. Example: Quadrature 1
6. Quadrature 2: Simpson's Rule
7. Analysis
8. Juputer: lists and NumPpy

Wednesday

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

1. Announcements

- Lecture and lab attendance and participation (which is really intend for those of you who are *not* in class to hear it).

  CS319 is assessed by programming assignments (based on labs), a class test, and a project. There is no final exam. Therefore, it has a heavier workload during the semester than other modules. No attendance is not recorded, but is strongly advised. Later, as we complete projects, it becomes essential.

- **Lab 2**: Was due today. All done?
- **Lab 3**: Tomorrow and Friday. Must submit your work-in-progress.
- **Class test** next week. Confirm time?

2. Arrays

Much of Scientific Computing involves working with data, and often collections of data are stored as **arrays**, which are **list**-like structures that stores a collection of values all of the same type.

Array syntax

Syntax to declare (i.e., create) an array:

```
<data_type> ArrayName[NumberOfElements];
```

- `data_type` can be any valid data type (even ones you create yourself)
- *`ArrayName`* is any valid identifier.
- *`NumberOfElements`* is any non-negative integer (or a variable storing same).

Array syntax (more details)

- If we declare an array such as

```
double v[N];
```

the N elements are indexed as

```
v[0], v[1], v[2], v[3], ..., v[N-1].
```

We can treat each as an single `double` variable.

- The language actually allows you to refer to `v[i]` where i is *any* integer, even one outside of the range $[0, N]$. (The reason why will be explain presently, but you should not do this intentionally.)

int w[3]; both w[-5] and w[105], eg, are legal - but bad.

- **Initialiation** means to give a variable a value at the same time you declare it.
- Contrast:

```
int no_val;       // not initialised
int has_val=10;   // initialized
```

- Usually, we don't initialised arrays (more why later), but we can:

```
int w[2]={10,3};   // w[0]=10, w[1]=3;
int x[]={-1,2};    // sets the size of array, and x[0]=-1, x[1]=2;
int y[4]={-5,5};   // sets y[0]=-5, y[1]=5, y[2]=y[3]=0
int z[5]={};       // inits all vals to 0.
```

↳ same as z[5] = {0};

Consider the following piece of code:

00Array.cpp

```
10  float vals[]={1.1, 2.2, 3.3};

12  for (int i=0; i<3; i++)
      std::cout << "  vals[" << i << "]=" << vals[i];
14  std:: cout << std::endl;
    std::cout << "vals=" << vals << '\n';
```

The output I get looks like

```
1 vals[0]=1.1  vals[1]=2.2  vals[2]=3.3
vals=0x7ffd9ab8ec9c
```

Can we explain the last line of output?

It is the memory address of vals[0].

So now we know that, if `vals` is the name of an array, then in fact the value stored in `vals` is the memory address of `vals[0]`.

We can check this with

```
  std::cout << "vals=" << vals << '\n';
2 std::cout << "&vals[0]=" << &vals[0] << '\n';
  std::cout << "&vals[1]=" << &vals[1] << '\n';
4 std::cout << "&vals[2]=" << &vals[2] << '\n';
```

For me, this gives

```
    vals=0x7ffc932b960c
2 &vals[0]=0x7ffc932b960c
  &vals[1]=0x7ffc932b9610
4 &vals[2]=0x7ffc932b9614
```

confirming vals stores memory address of vals[0]

difference of 4 bytes

Can we explain? Recall: if x is a variable, then &x is its mem addr

And in the same piece of code, if I changed the first line from

`float vals[3];`

to

`double vals[3];`

we get something like

```
    vals=0x7ffd361abdc0
&vals[0]=0x7ffd361abdc0
&vals[1]=0x7ffd361abdc8
&vals[2]=0x7ffd361abdd0
```

Can we explain?

Yes! Each address differs by 8 bytes, which is what is required to store a double.

So now we understand why C++ (and related languages) index their arrays from `0`:

- `vals[0]` is stored at the address in `vals`;
- `vals[1]` is stored at the address after the one in `vals`;
- `vals[k]` is stored at the *k*th address after the one in `vals`;

But there are numerous complications, not least that different data types are stored using different numbers of bytes. So the off-set depends on the data type.

To understand the subtleties, we need to know about **pointers**.

3. Timing Code

A Brief Interlude on timing code

In Scientific Computing we should be obsessed with accuracy, precision, correctness, and **efficiency**.

To be confident that our code is efficient, we need to be able to estimate how long it will take to run.

In more advanced settings we can using timing to locate bottle-necks in our code.

C++ provides several mechanisms for timing. We'll first look at the tools provided by the `ctime` library.

- Have `#include <ctime>` when other headers, such as `iostream` are loaded.
- Provides the `clock_t` data type for storing CPU times.
- The `clock()` function returns the CPU time since the program started.
- That can be converted to the time (in seconds) that have elapsed by dividing by `CLOCKS_PER_SEC`

Example

To time a snippet of code, try

```
clock_t start=clock();
// chunk of code goes here
clock_t end=clock();
double seconds=double(end - start)/CLOCKS_PER_SEC;
std::cout << "CPU time: " << seconds << " seconds\n";
```

In the code below we'll time how long it takes to declare a large array, with and without initialization:

01TimeCode.cpp

```
#include <ctime>
int main()
{
  // Timing without initialising the array
  int Runs=10000, // number of runs to time
    ArraySize=1000000;
  clock_t start = clock();     // start stop-watch
  for (int i=0; i<Runs; i++)
    double arr[ArraySize+i];     // declare the array (no init)
  double seconds = double(clock() - start)/CLOCKS_PER_SEC;
  std::cout << "Ave Time (no init): " << seconds/Runs << "s\n";

  // Timing with initialising the array
  start = clock();     // restart stop-watch
  for (int i=0; i<Runs; i++)
    double arr[ArraySize+i]={};     // declare the array (with init)
  seconds = double(clock() - start)/CLOCKS_PER_SEC;
  std::cout << "Ave Time ( init ): " << seconds/Runs << "s\n";
```

Notes:

On some systems, the results of ctime tools can be unreliable.

More modern tools are provided by the chrono library, which is a little more complicated.

```
auto start = std::chrono::high_resolution_clock::now();

// stuff you want to time goes here

auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> secs = end - start;

std::cout << "Ave Time (no init chron): "
    << secs.count()/Runs  << "s\n";
```

Finished here Wednesday

4. Pointers

To properly understand how to use arrays, we need to study **Pointers**.

- We already learned that if, say, `x` is a variable, then `&x` is its memory address.
- A **pointer** is a special type of variable that can store memory addresses. We use the `*` symbol before the variable name in the declaration.
- For example, if we declare

  ```
  int i;
  int *p
  ```

  then we can set `p=&i`.

4. Pointers

02Pointers.cpp

```
10  int a=-3, b=12;
    int *where;

    std::cout << "The variable 'a' stores " << a << std::endl;
14  std::cout << "The variable 'b' stores " << b << std::endl;
    std::cout << "'a' is stored at address " << &a << std::endl;
16  std::cout << "'b' is stored at address " << &b << std::endl;

18  where = &a;
    std::cout << "The variable 'where' stores "
20            << (void *) where << std::endl;
    std::cout << "... and that in turn stores " <<
22    *where << '\n';
```

One can actually do calculations on memory addresses. This is called **pointer arithmetic**. One can't (for example) add two addresses, or compute their product, but you can, for example, increment them.

03PointerArithmetic.cpp

```
  int vals[3];
8 vals[0]=10;  vals[1]=8;  vals[2]=-4;

10 int *p;
   p = vals;

   for (int i=0; i<3; i++)
14 {
     std::cout << "p=" << p << ", *p=" << *p << "\n";
16   p++;
   }
```

Being able to manipulate memory addresses is one of the reasons C++ is considered a very **powerful** language. It is possible to preform (low-level) operations in C++ that are impossible in, say, Python.

(For many years, the US Cybersecurity and Infrastructure Security Agency, CISA, has argued that C and C++ are inherently unsafe).

But it is also possible to write programmes that will crash, or even crash your computer, since memory addresses are not well protected.

5. Dynamic Memory Allocation

In all examples we've had so far, we've specified the size of an array at the time it is defined.

In many practical cases, we don't have that information. For example, we might need to read data from a file, but not know the file size in advance.

It would be useful if, on the fly, we could set the size of an array.

Furthermore, for efficiency, we may want to free up memory allocated.

To add this functionality, we will use two new (to us) C++ operators for dynamic memory allocation and deallocation:

- `new` and
- `delete`.

(There are also functions `malloc()`, `calloc()` and `free()` inherited from C, but we won't use them).

The `new` operator is used in C++ to allocate memory. The basic form is

```
var = new type
```

where *type* is the specifier of the object for which you want to allocate memory and *var* is a pointer to that type.

If insufficient memory is available then `new` will return a *NULL* pointer or generate an exception.

To dynamically allocate an array:

- First declare a pointer of the right type:
  ```
  int *data;
  ```
- Then use `new`
  ```
  data = new int [MAXSIZE];
  ```

When it is no longer needed, the operator `delete` releases the memory allocated to an object.

To "delete" an array we use a slightly different syntax:

```
delete [] array;
```

where *array* is a pointer to an array allocated with `new`.

6. Example: Quadrature 1

Previously, we introduced the idea of **numerical integration** or **quadrature**.

We computed estimates for $\int_a^b f(x)dx$ by applying the Trapezium Rule:

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

[Take notes for the next few slides]

6. Example: Quadrature 1

6. Example: Quadrature 1

04TrapeziumRule.cpp

```
4 #include <iostream>
  #include <cmath>  // For exp()
6 #include <iomanip>

8 double f(double x) {  return(exp(x)); } // definition
  double ans_true = exp(1.0)-1.0; // true value of integral

  double Quad1(double *x, double *y, unsigned int N);
```

6. Example: Quadrature 1

Next we skip to the function code...

04TrapeziumRule.cpp

```
double Quad1(double *x, double *y, unsigned int N)
{
  double h = (x[N]-x[0])/double(N);
  double Q = 0.5*(y[0] + y[N]);
  for (unsigned int i=1; i<N; i++)
    Q += y[i];
  Q *= h;
  return(Q);
}
```

Source of confusion: * is used in two very different contexts here.

6. Example: Quadrature 1

Back to the main function: declare the pointers, input N, and allocate memory.

04TrapeziumRule.cpp

```
int main(void )
{
  unsigned int N;
  double a=0.0, b=1.0; // limits of integration
  double *x; // quadrature points
  double *y; // quadrature values

  std::cout << "Enter the number of intervals: ";
  std::cin >> N; // not doing input checking

  x = new double[N+1];
  y = new double[N+1];
```

6. Example: Quadrature 1

Initialise the arrays, compute the estimates, and output the error.

04TrapeziumRule.cpp

```
  double h = (b-a)/double(N);
26 for (unsigned int i=0; i<=N; i++)
  {
28   x[i] = a+i*h;
     y[i] = f(x[i]);
30 }
  double Est1 = Quad1(x, y, N);
32 double error = fabs(ans_true - Est1);
  std::cout << "N=" << N << ", Trap Rule="
34          << std::setprecision(6) << Est1
            << ", error=" <<  std::scientific
36          << error << std::endl;
```

6. Example: Quadrature 1

Finish by de-allocating memory (optional, in this instance).

04TrapeziumRule.cpp

```
38    delete [] x;
      delete [] y;
40    return(0);
}
```

6. Example: Quadrature 1

Although this was presented as an application of using arrays in C++, some questions arise...

1. What value of N should we pick the ensure the error is less than, say, 10^{-6}?
2. How could we predict that value if we didn't know the true solution?
3. What is the smallest error that can be achieved in practice? Why?
4. How does the time required depend on N? What would happen if we tried computing in two or more dimensions?
5. **Are there any better methods? (And what does "better" mean?)**

6. Example: Quadrature 1

Some answers to some of those questions

7. Quadrature 2: Simpson's Rule

Simpson's Rule is an improvement on the Trapezium Rule.

Here is a rough idea of how it works:

7. Quadrature 2: Simpson's Rule

Simpson's Rule

- Choose an **EVEN** number of intervals N, and set $h = (b - a)/N$.
- Define the quadrature points $x_0 = a$, $x_1 = a + h$, $\ldots x_N = b$. In general, $x_i = a + ih$.
- Set $y_i = f(x_i)$ for $i = 0, 1, \ldots, N$.
- Compute

$$Q_2(f) := \frac{h}{3}\Big(y_0 + \sum_{i=1,3,\ldots,N-1} 4y_i + \sum_{i=2,4,\ldots,N-2} 2y_i + y_N\Big).$$

7. Quadrature 2: Simpson's Rule

The program 01CompareRules.cpp implements both methods and compares the results for a given N. Here we just show the code for the implementation of Simpson's Rule.

05CompareRules.cpp

```
double Quad2(double *x, double *y, unsigned int N)
{
  double h = (x[N]-x[0])/double(N);
  double Q = y[0]+y[N];
  for (unsigned int i=1; i<=N-1; i+=2)
    Q += 4*y[i];
  for (unsigned int i=2; i<=N-2; i+=2)
    Q += 2*y[i];
  Q *= h/3.0;
  return(Q);
```

7. Quadrature 2: Simpson's Rule

When we run `05CompareRules.cpp`, and h test both methods attempts at estimating

$$\int_0^1 e^x dx,$$

we get output like:

N	Trapezium Error	Simpson's Error
8	2.236764e-03	2.326241e-06
16	5.593001e-04	1.455928e-07
32	1.398319e-04	9.102726e-09
64	3.495839e-05	5.689702e-10

From this we can quickly observe the Simpson's Rule to give smaller errors than the Trapezium Rule, for the same effort.

Can we quantify this?

8. Analysis

Next we want to analyse, experimentally, the results given by these program.

We'll do the calculations, in detail, for the Trapezium Rule.

In Lab 5, you will redo this for Simpson's Rule.

. . . . . .

Let $E_N = |\int_a^b f(x)dx - Q_1(f)|$ where $Q_1(\cdot)$ is implemented for a given N.

We'll speculate that

$$E_N \approx CN^{-q},$$

for some positive constants C and q. If this was a numerical analysis module (like MA378) we'd determine C and p from theory. In CS319 we do this **experimentally**.

8. Analysis

The idea:

8. Analysis

To implement this, we need some data. That can be generated, for the Trapezium Rule, by the following programme.

Notice that we use dynamic memory allocation. That is because the size of the arrays, `x` and `y` change while the programme.

06CheckConvergence.cpp

```
18 int main(void )
   {
20   unsigned K = 8;   // number of cases to check
     unsigned Ns[K];    // Number of intervals
22   double Errors[K];
     double a=0.0, b=1.0; // limits of integration
24   double *x, *y; // quadrature points and values.
```

8. Analysis

06CheckConvergence.cpp

```
26  for (unsigned k=0; k<K; k++)
    {
28    unsigned N = pow(2,k+2);
      Ns[k] = N;
30    x = new double[N+1];
      y = new double[N+1];
32    double h = (b-a)/double(N);
      for (unsigned int i=0; i<=N; i++)
34    {
        x[i] = a+i*h;
36      y[i] = f(x[i]);
      }
38    double Est1 = Quad1(x,y,N);
      Errors[k] = fabs(ans_true - Est1);
40    delete [] x; delete [] y;
    }
```

8. Analysis

Our program outputs the results in the form of two `numpy` arrays. We'll have two different functions (with the same name!), since one is an array of `int`s and the other `double`s.

Here is the code for creating outputting `numpy` array of doubles. The one for `int`s is similar.

06CheckConvergence.cpp

```
void print_nparray(double *x, int n, std::string str)
{
  std::cout << str << "=np.array([";
  std::cout << std::scientific << std::setprecision(6);
  std::cout << x[0];
  for (int i=1; i<n; i++)
    std::cout << ", " << x[i];
  std::cout << "])" << std::endl;
```

9. Juputer: lists and NumPpy

- The next set of slides are in the Jupyter Notebook: `CS319-Week05-notebook.ipynb`. (NOT POSTED YET!!)
- Can be downloaded from https://www.niallmadden.ie/2526-CS319
- Can try that out on `https://cloudjupyter.universityofgalway.ie`
- Tip: on that server, try: `File... Open from URL...` add `https://www.niallmadden.ie/2526-CS319/Week05/CS319-Week05-notebook.ipynb`