Annotated slides from Wednesday

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

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

Wed
nes
day

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.

2. Arrays

- **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};

2. Arrays

Consider the following piece of code:

00Array.cpp

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

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

The output I get looks like

```
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].

2. Arrays

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

For me, this gives

```
    vals=0x7ffc932b960c } confirming vals stores
&vals[0]=0x7ffc932b960c }  memory address of vals[0]
&vals[1]=0x7ffc932b9610 }
&vals[2]=0x7ffc932b9614 } difference of 4 bytes
```

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

2. Arrays

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.

Base Address

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 kth 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.

3. Timing Code

- 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";
```

3. Timing Code

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";
```

3. Timing Code

Notes:

3. Timing Code

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