Design and Computing 1

Introduction to Scientific Computing

Daniel Poole
Department of Aerospace Engineering
University of Bristol
d.j.poole@bristol.ac.uk

2017

LECTURE 6

Intro to Scientific Computing

- Have used programs to perform simple calculations
- Conditionals can be used to branch off during a program
- Loops are used to perform calculations multiple times
- Can split a problem down and program using divide-and conquer using functions
- Can read and write with files and use scientific plotting software

TODAY

- Storing data in arrays/matrices

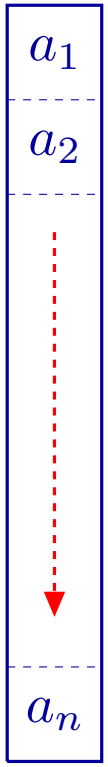
Arrays

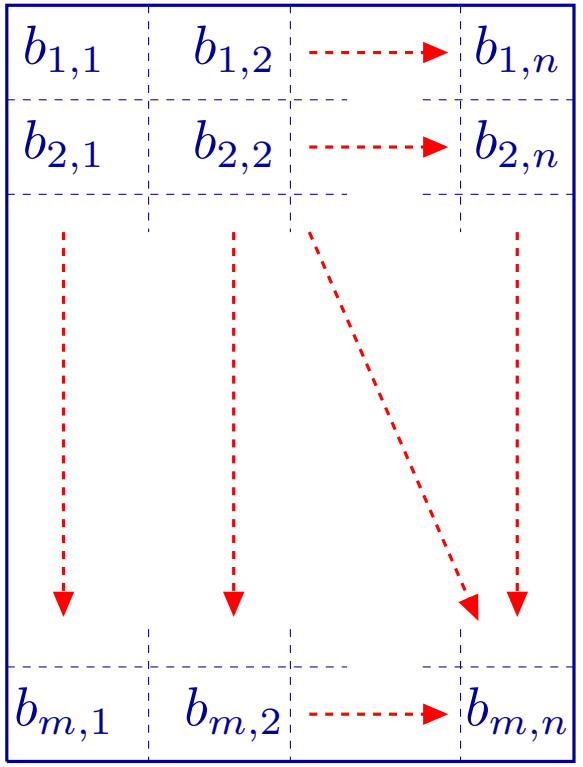
So far, we have been storing data in individual variables, where one piece of data is assigned to one variable. However, as should be obvious, problems start to arise when we want to store vast amounts of data within our program. An example of this is the solution of a computational fluid dynamics (CFD) code where the flowfield variables (such as pressure and density) need to be stored at a large number of places in the volume of interest. Furthermore, up until now, we have known exactly the amount of data we need to store, but what happens if we know we need to store some data, but as at the time of writing your code, you don't exactly know how much data it will be.

This week we will consider storing data in arrays (i.e. matrices of data). This allows us to start storing much greater quantities of data that we can then manipulate. We will also consider the idea of allocating the correct size of arrays, for those scenarios where we do not know the amount of data we will need to store at the time of writing.

Arrays

An array is just a general way of saying a vector (column of data) or matrix (columns and rows and sometimes extra dimensions):





Array Syntax

If we know how large we want our array, then we declare it as:

```
int array1[3];
double array2[5];
```

Which means "make space for an array called `array1` of three integers" and "make space for an array called `array2` of five doubles".

To assign or access an entry in an array we do:

```
array1[0]=2;
b=array2[4];
```

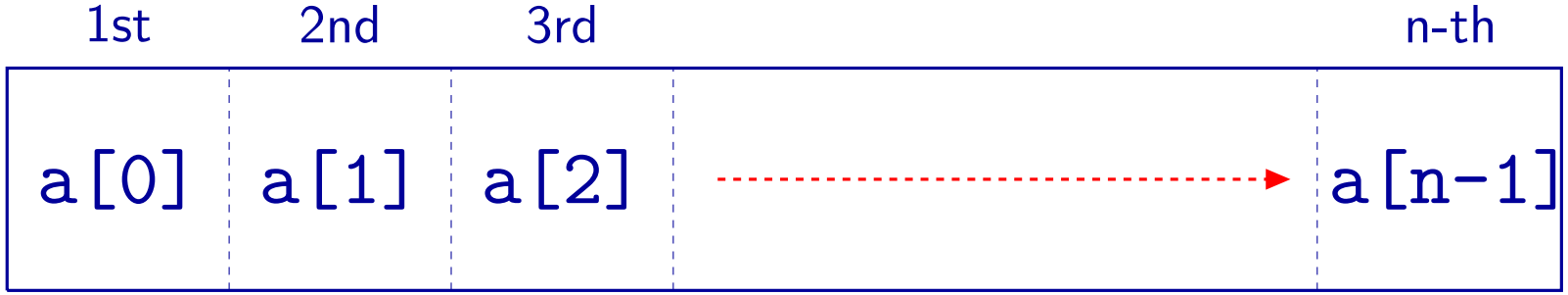
Which means "assign the **first** entry of `array1` a value of two" and "assign to variable `b` the value of the **fifth** entry of `array2`".

NOTE: C array counting starts from 0

Array Syntax

NOTE: C array counting starts from 0

So an array of size `n` is indexed as:



You can only assign or retrieve one item at a time, which also applies to printing our array, so we would need a loop:

```
for(ii=0; ii<n; ii++){
   printf("%i \n",a[ii]);
}
```

Worked Example: Bubble Sort

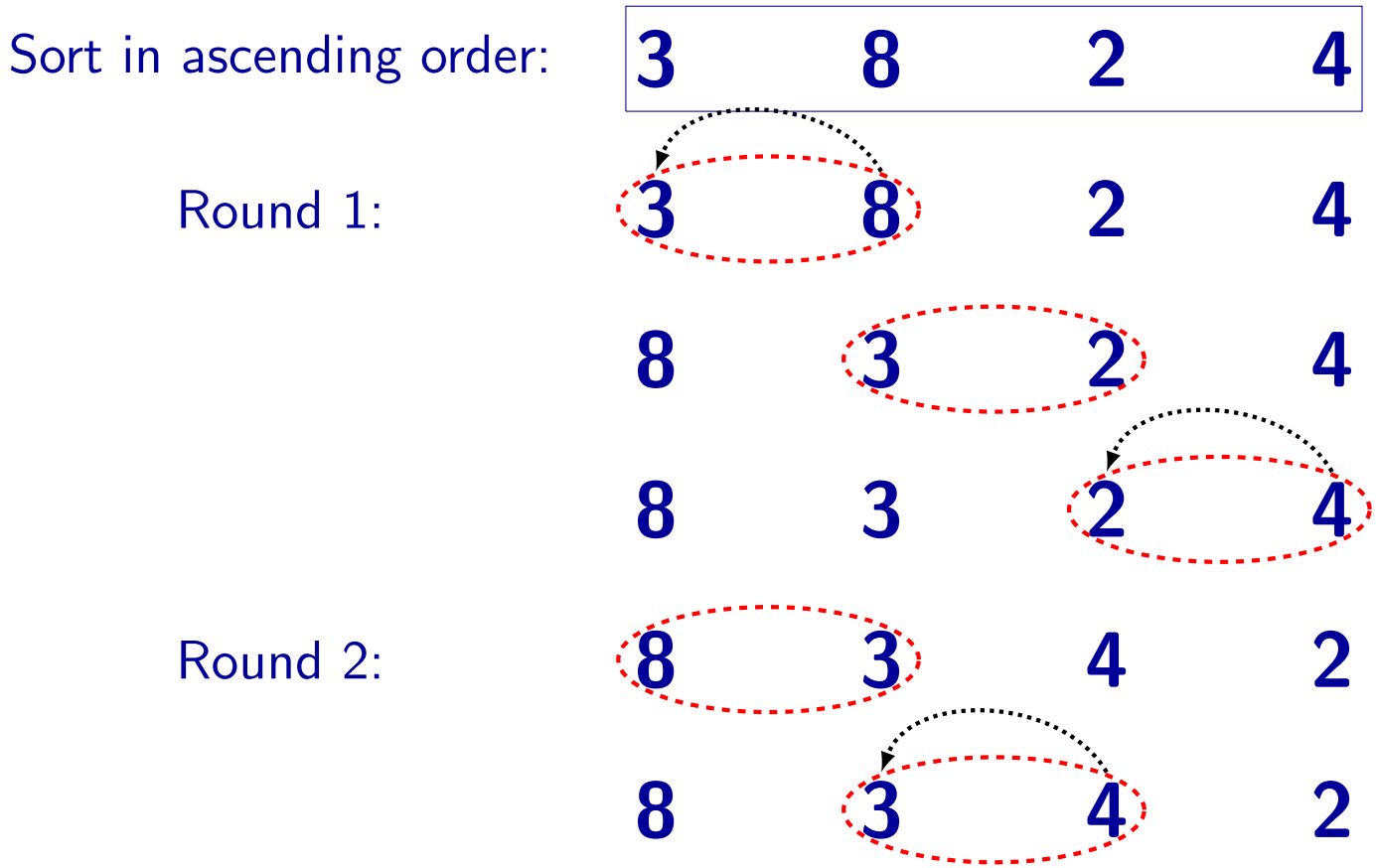
The bubble sort is the simplest of algorithms that will sort of series of numbers into either ascending or descending order. The algorithm works by comparing two adjacent numbers, and swapping them if one is larger. This keeps looping through all of the entries over and over until no more swapping has occurred e.g.

Sort in ascending order:

3	**8**	**2**	**4**

Worked Example: Bubble Sort

The bubble sort is the simplest of algorithms that will sort of series of numbers into either ascending or descending order. The algorithm works by comparing two adjacent numbers, and swapping them if one is larger. This keeps looping through all of the entries over and over until no more swapping has occurred e.g.



Worked Example: Bubble Sort

As always, we need to split this down into a series of smaller problems (divide-and-conquer):

1. Read in numbers to be sorted and store in an array (done recursively)

2. For each pair of numbers in the list

   (a) Test if the second is larger
   (b) If it is then perform a swap
   (c) Print the new list

3. If no swapping has occurred this time then stop

4. Otherwise run through the list again

Multidimensional Arrays

We don't just have array vectors, but can have multidimensional arrays too. This is an array of arrays. For example, a multiplication table:

```
int multi[3][5], ii, jj;
for(ii=0; ii<=2; ii++) {
  for(jj=0; jj<=4; jj++) {
    multi[ii][jj]=(ii+1)*(jj+1);
  }
}
```

Which if printed would give us:

```
1  2  3  4  5
2  4  6  8 10
3  6  9 12 15
```

Arrays and Functions

We can send the whole array to a function (or obviously just one element too):

```
int myfunc(int multi[2]) {
  multi[1]=0;
  return(0);
}

int main () {
  int x[2]={5,5};
  printf("%i\n",x[1]);
  myfunc(x);
  printf("%i\n",x[1]);
  return(0);
}
```

BUT: if we change the value of an array entry in the function, the value is also changed in the main program. You can think of arrays as **GLOBAL** variables, meaning then can be changed by the main and all other functions. In this case, we have sent the whole array but modified only one element. This has then been modified in `main` too.

```
5
0
```

Arrays and Functions

Example (sum of the array elements):

```
int sumArray(int a[3]) {
  int ii, sum=0;
  for(ii=0;ii<3;ii++){
    sum+=a[ii];
  }
  return(sum);
}

int main () {
  int y, x[3]={1,2,3};
  y=sumArray(x);
  printf("Sum=%i\n",y);
  return(0);
}
```

BUT: if we change the value of an array entry in the function, the value is also changed in the main program. You can think of arrays as **GLOBAL** variables, meaning then can be changed by the main and all other functions.

```
Sun=6
```

Variable Array Sizes

As mentioned at the start of the lecture, often we do not know, at the time of writing a program, how much data we are going to store in an array (in reality, we very rarely know this fact). In these cases it is useful to set-up an array, but not to give it a size. This can be done later in the code. This is called allocating the array.

Firstly, though, we must load a new library:

```
#include<stdlib.h>
```

http://www.cplusplus.com/reference/cstdlib/

Variable Array Sizes

We declare our array using the * syntax:

```
int *myarray;
```

We can use the `calloc` function to allocate the size of our array after we have declared it:

```
myarray=(int *)calloc(10,sizeof(int));
```

Which means "make the integer array `myarray` to be ten entries big". `sizeof(<type>)` is a function that returns the size of the memory used for storing one variable of the specified `<type>`.

We can also deallocate the array if we know we no longer need it, or need it for something else:

```
free(myarray);
```

Variable Array Sizes

We can now use a variable (which must be integer) to allocate the size of our arrays:

```
#include<stdlib.h>
int main () {
  int *myarray1, n1, n2;
  double *myarray2;
  n1=5; n2=7;

  /* Allocate arrays */
  myarray1=(int *)calloc(n1, sizeof(int));
  myarray2=(double *)calloc(n2, sizeof(double));
  /* Deallocate arrays */
  free(myarray1);
  free(myarray2);

  return(0);
}
```

Variable Array Sizes

Multidimensional arrays get complicated as we have to allocate each sub-element (i.e. each column) individually. This is due to the way in which the memory is set-up in C:

```
int **myarray, nrows, ncolumns, i;
nrows=5, ncolumns=3;

myarray=(int **)calloc(nrows, sizeof(int *));
for(i = 0; i < nrows; i++){
  myarray[i] = (int *)calloc(ncolumns, sizeof(int));
}
```

Note the use of `**myarray` as we are dealing with a two dimensional array. This will become clearer when we cover the pointers in a future lecture. Essentially, the first `calloc` call makes `myarray` point to `nrows` pointers, which each point to a different vector of `ncolumns` integers (using the second `malloc` call).

Summary

- Have considered storing data in arrays
- Have introduced idea of allocating array during run-time

LAB

- Reading data into an array and manipulation of data stored in arrays