Arrays

Ajit Rajwade

Refer: Chapter 14 of the book by Abhiram Ranade

What is an array?

- Suppose for every day between 1st Jan 2022 and 31st December 2022 (say), you wanted to record the value of the US dollar against the Indian rupee, i.e. 1 USD = x INR.
- If you wanted to store this value x for one day, a single variable would suffice.
- But suppose you want to store the values for all 365 days of the year, or maybe for each day over the last 5 years.
- In such a case, you need a **contiguous chunk of memory** that contains these 365 variables - all of the same type (float in this case).
- Such a structure is called an **array**.

Array declarations

- An array is declared to be of a certain data type and is given an identifier following the same rules for creating identifiers.
- Examples:
  - `float usd_to_inr[365];`
  - `double a[100], c[50];`
  - `int abc[250];`
  - `char pqr[100];`
- The number in the square brackets used while declaring the array gives the **size** (also called **length**) of the array.
- The `[]` is an operator that is also used to access elements of the array.
- `a[0]` accesses the first element of the array, `a[1]` the second, `a[2]` the third and so on.
- The **indices** (also called **subscripts**) are numbered from **`0`** (not 1) onwards till **`n-1`** (not `n`) if the array size is `n`.

Array declarations

- Array declarations can also be mixed with declarations of other scalar elements of the same type.
- Examples:
  - `double xyz[100], a, b, pqr[500];`
- The size of the array must always be declared to be a constant.
- Sometimes this is done via the `#define` operators as in the example below.

```
#include<simplecpp>

#define MAXSIZE 100

int main (){

    int a[MAXSIZE]; double b[MAXSIZE/2];

}
```

Array declarations

- Array declarations can also contain an **initialization**.
- Example:
  - `double xyz[] = {1.0, 2.5, 3.1, -9.6};`
  - `double xyz[4] = {1.0, 2.5, 3.1, -9.6};`
- In many cases, such initializations are not very convenient, and it is better to do the value assignment to different elements inside a piece of code.

Array element operations

- The `(i+1)`-th element of an array `a` is accessed using `a[i]`. Note that the first element is accessed as `a[0]`.
- `a[i]` can feature in an expression on the LHS or RHS or can be read into directly via a keyboard input.
- Examples: (for an array `a`)
  - `cin >> a[2];`
  - `a[4] = a[5] + a[6]*10*a[7];`
  - `a[4] = 3;`
  - `int b = 7*a[0];`
  - `int i = 1; a[i+1] = a[5]*a[6]+1;`
- Single elements of an array `a` behave just like scalars and can be passed to functions as arguments. For example, referring to earlier slides we could use `gcd(a[0][,a[1])` where `a` is an integer array.

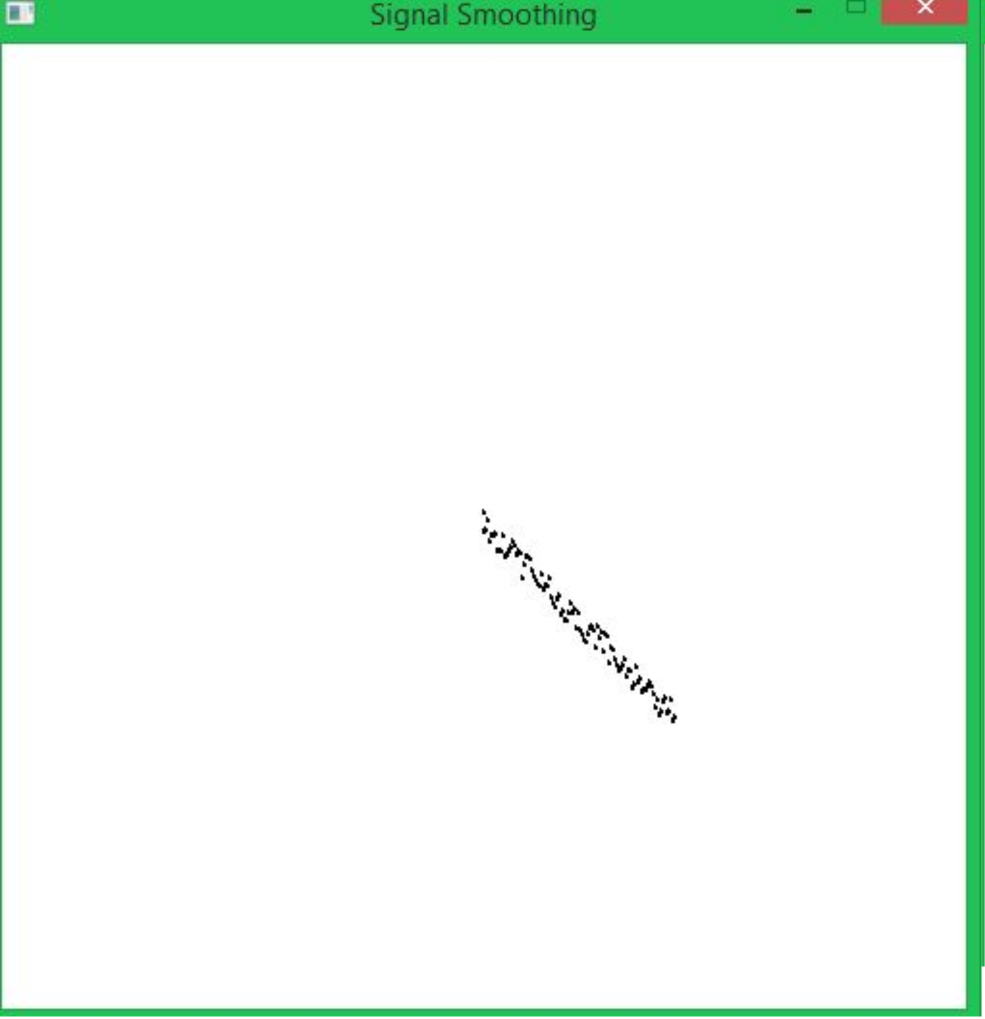
Range for an index

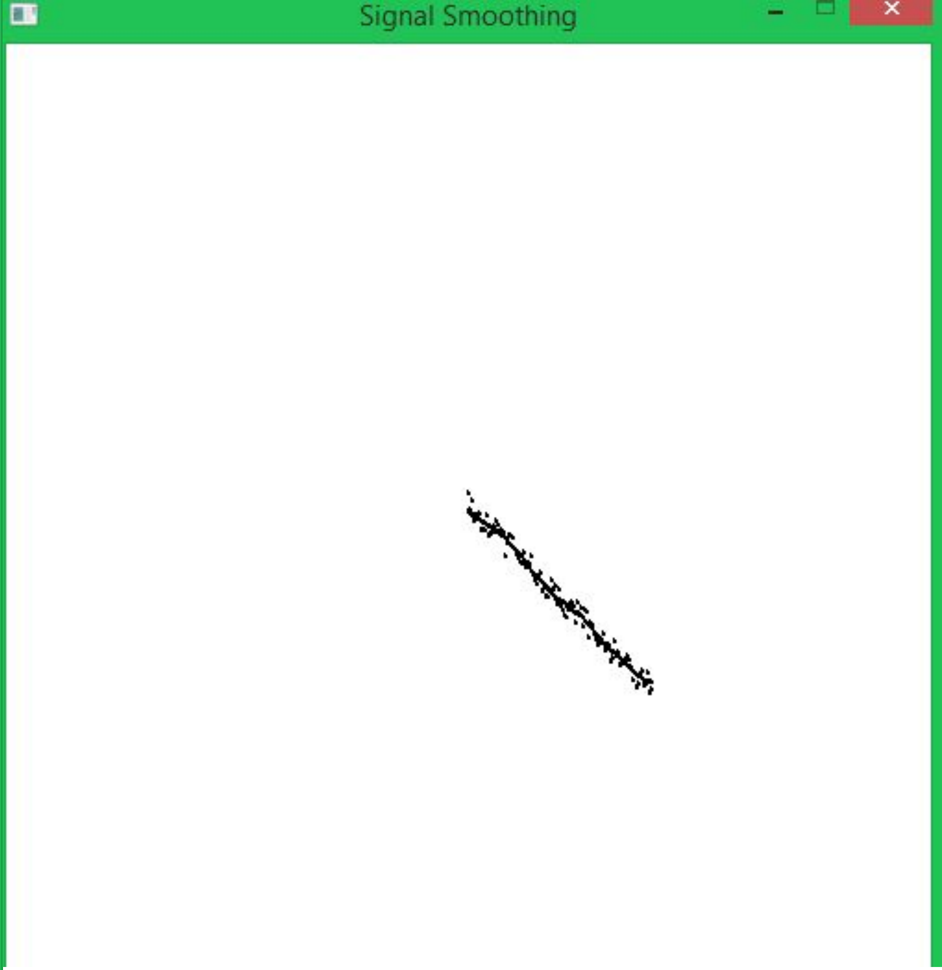
- Consider an array declared as `int a [100];`
- It is the job of the programmer to ensure that array indices for `a` will always lie in the range from 0 to 100-1 (both inclusive).
- If you attempt to access an array element with index outside this range, you may get **runtime** errors or (even worse!) **strange behaviour** in your program with unexpected and unpredictable outputs.
- For example: use of elements `a[-1]` or `a[100]` or `a[101]` will not cause syntax errors, but will cause runtime errors.

Example 1: Signal Smoothing

- Imagine you recorded a person speaking via a digital recorder.
- Let us say the recorder collects `n` samples every second, and does so for `t` seconds, for a total of `m = tn` samples.
- These audio samples can be stored in an **array** of `m` floating point numbers.
- There is ambient noise (wind, fan noise, etc.) due to which the recording is noisy.
- So you can smooth this signal, for which you replace the noisy sample at index `i` by the average of all values from `leftindex = max(0,i-5)` to `rightindex = min(m-1,i+5).`
- You store the smoothed signal in **another** array of the **same** size.

```
#include<simplecpp>
#include<cstdlib>
#define SIGLENGTH 100
main_program{
// array declarations for the original, noisy, restored signals
float signal[SIGLENGTH],noisy_signal[SIGLENGTH], restored_signal[SIGLENGTH],noiseval;
int i, leftindex, rightindex, j; const int windowRadius = 7;

initCanvas ("Signal Smoothing",500,500);// inbuilt function to create a new window
Circle pt(0,0,0);

for(i = 0; i < SIGLENGTH; i++)
{
    signal[i] = i; // simple signal created
    noiseval = 15*float(rand()-RAND_MAX/2)/RAND_MAX;
    noisy_signal[i] = signal[i] + noiseval;
    pt.reset(i+250,noisy_signal[i]+250,1);// draw a small circle around that point
    pt.imprint();
}
wait(2);

// to be continued on next slide
for(i = 0; i < SIGLENGTH; i++)
{
    leftindex = max(0,i-windowRadius);
    rightindex = min(SIGLENGTH-1,i+windowRadius);

    // code for the averaging operation
    float avg = 0.0;
    for (j=leftindex; j <= rightindex; j++)
    {
        avg += noisy_signal[j];
    }
    avg /= (rightindex-leftindex+1);
    restored_signal[i] = avg;

    pt.reset(i+250,restored_signal[i]+250,1);// draw a small circle around that point
    pt.imprint();
}

wait(10);
}
```


Signal Smoothing


Signal Smoothing

Example 1: Signal Smoothing

- For simplicity, we created a signal of the form f(x) = x.
- We added random values (noise) to it to make it look wiggly.
- Then we smoothed the signal by performing averaging.
- Notice that we maintained **three separate arrays**: one each for the original, noisy and restored signals.
- Notice how close the restored signal is to the original signal (though not identical to it).
- You may have heard of convolutional neural networks (CNNs) in the media.
- What you just implemented is a simple convolution operation, i.e. you implemented the convolution of the noisy signal with a so-called “moving average filter”. Hurray! :-)

Example 2: Histograms

- Consider we have with us the scores of `n` students in a course.
- We want to determine how many students scored above 90, how many scored more than 80 and less than or equal to 90, how many scored above 70 but less than or equal to 80 and so on.
- Such a computation is called a **histogram**, and it is a very basic structure in statistics.
- We will write a program to perform this task. We will maintain two arrays: one for the histogram, and the other for the marks for each student.

```
for(i=0;i<10;i++) hist[i] = 0; // histogram with 10 “bins” or intervals

for(i=0;i<n;i++){

if (scores[i] <= 10) hist[0]++;

else if (scores[i] <= 20) hist[1]++; else if (scores[i] <= 30) hist[2]++;

else if (scores[i] <= 40) hist[3]++; else if (scores[i] <= 50) hist[4]++;

else if (scores[i] <= 60) hist[5]++; else if (scores[i] <= 70) hist[6]++;

else if (scores[i] <= 80) hist[7]++; else if (scores[i] <= 90) hist[8]++;

else hist[9]++;

}
```

In this example, all bins had equal width, but you can easily modify this code to handle unequal bin sizes as well.

Accessing array elements

- Consider an array `int a[100];` – integer array of 100 elements
- For each element of `a`, there are 4 bytes allocated since an `int` occupies 4 bytes.
- These bytes are all allocated in a contiguous chunk.
- Let us say that the first element (`a[0]`) is at address `X`.
- The next element (`a[1]`) will be located at byte address `X+4`, element `a[2]` at byte address `X+8` and so on.
- When the term of the form `a[expression]` is encountered, `expression` is first evaluated.
- If its value is `v`, then `a[expression]` lies at `X+4v` (or `X+kv` for a datatype needing `k` bytes of memory per element).
- Due to these calculations, accessing array elements takes a bit more time than accessing a single variable. But the access time is not dependent on the array size.

Arrays and Pointers

- Array elements are accessed as `a[i]`.
- An alternative expression for this is: `*(a+i)`
- Here `a` refers to the base address of the array, `a+i` refers to the address of the `(i+1)-th` element, and `*(a+i)` fetches the values of the `(i+1)-th` element.
- Consider the following: `int a[10]; int *b = a;`
- Here `b` is a pointer to an integer and it is assigned the base address of the array `a`.
- Hence using `b[3]` has the same effect as using `a[3]`.
- Note however that you can never change the base address of an array after declaration - it will result in compiler errors.
- For example, the following is not allowed:

```
int a[10], b[10];

a = b; // not allowed - compiler error
```

Arrays and Functions

- Arrays can be passed as arguments to a function.
- Note that arrays are never passed by value, but only by reference.
- If they were passed by value, it would have been very costly in terms of both memory and time, especially for large sized arrays that would need to copied from the calling function to the one being called!
- Consider the following function which takes an array `a` and its size `n` as input, and doubles the value of each element.

```
void double_array_values (int *a, int n){
    for (int i = 0; i < n; i++) a[i] *= 2;
}
```

Arrays and Functions

- Consider two more functions which takes an array `a` and its size `n` as input. One prints the value of each element. The other takes input into array elements.

```
void print_array_values (int *a, int n){

    for (int i = 0; i < n; i++) cout << a[i];

}

void input_array_values (int *a, int n){

    for (int i = 0; i < n; i++) cin >> a[i];

}
```

Arrays and Functions

- Alternative syntax for these function declarations:

```
void print_array_values (int a [], int n){
    for (int i = 0; i < n; i++) cout << a[i];
}
void input_array_values (int a [], int n){
    for (int i = 0; i < n; i++) cin >> a[i];
}
void double_array_values (int a [], int n){
    for (int i = 0; i < n; i++) a[i] *= 2;
}
```

Arrays and Functions

- These functions are called in the following manner:

```
int a[10];

// code to assign values to elements of a

input_array_values(a,10);

// as arrays are passed by reference, the values in ‘a’ are

// doubled. The change is retained even after the function exits

double_array_values (a, 10);

// code to print array values

print_array_values(a,10);
```

- When an array is passed as argument to a function, its length also needs to be passed.
- When you just refer to array `a`, you refer to its base address, but there is no knowledge of its length unless you specify it!

Sorting

- It is the process of arranging the elements of an integer or floating point array in strictly non-decreasing or non-increasing order.
- **Example 1:** while assigning letter grades, a course instructor sorts the records of all the students for the course in non-increasing order of scores.
- **Example 2:** When you search for a product on Amazon, you get a listing in decreasing order of popularity or increasing order of price.
- Sorting is a very fundamental operation on an array.
- It is also a fundamental task in computer science.
- There are many algorithms to perform sorting, and we will study the simplest one called **selection sort**.

Selection Sort

- Consider an array `A` with `n` elements.
- We first search for the largest value in `A` (let us say it was at index `k1`) and would like to place it at index `n-1`.
- However `A[n-1]` already contains a value, which we do not want to destroy. Hence we swap `A[n-1]` and `A[k1]`.
- Now, `A[n-1]` contains the **same value** that it would have contained in a **perfectly sorted** array. The rest of the array is still **unsorted**.
- Now, we find the maximum element in `A` from index 0 to index `n-2`. Let us say it lies at index `k2`.
- Then we want to swap `A[k2]` and `A[n-2]`.
- We proceed in this fashion: in the iteration with index `i`, we find the maximum element in `A` from the first `i` indices and then exchange it with the value at the `(i-1)`th index.

Selection Sort

- We will first write a function to return the index of the maximum element in an array `A` in the first `L` indices.

```
int argmax (float *scores, int L){
    int maxindex = 0;
    for(int j=1; j<L; j++){
        // update the index if a larger element is found
        if(scores[maxindex] < scores[j]) maxindex = j;
    }
    return maxindex;
}
```

```
void selection_sort (float *data, int n){

for(int i=n; i>1;i--){

int maxindex = argmax(data,i); // get the maximum of the first i
elements (from index 0 to index i-1)

myswap(&data[maxindex],&data[i-1]); // refer to earlier slides for
this myswap function!

// at this point, data[i-1] will be in the correct position for a
sorted array

}

}
```

How will you modify this program to sort the array in non-increasing order?

Selection sort animation:
https://en.wikipedia.org/wiki/Selection_sort#/media/File:Selection-Sort-Animation.gif

How long does Selection Sort take?

- There are `n-1` iterations of the `for` loop in selection sort.
- In each iteration, there is a call to `argmax` to find the index of the largest element from the first `i` indices.
- The very first call to `argmax` will require `n` operations, then `n-1`, then `n-2`, and so on till 2.
- The sum total of all these is equal to $(n+2)(n-1)/2$ which is approximately equal to $n^2/2$.
- Note that the time to perform a single swapping is considered constant per operation (independent of `n`).

Array Declarations inside functions

- In newer versions of C++, including the one you will use for this course, array sizes can be declared in the form of expressions involving parameters passed to the function.
- This is a very convenient feature!
- Example:

```
void my_function (int n){
    double A[n]; // declare an array of size n
}
```

Array declarations inside functions

- We will write a *function* which takes in an array of scores and produces a histogram.
- We will provide it the following parameters:
  - An array of student scores called `scores`
  - The number of students (length of scores): `n`
  - The number of desired histogram bins: `numbins`
  - Each bin has a starting and ending value - we will provide the ending values in a **sorted (increasing)** float array with `numbins` elements called `bin_end_values`
  - An array called `hist` which will store the histogram counts. It will have `numbins` elements. **But note that this is not actually an input but an output parameter! The output of the function will be the histogram!**

```
void create_histogram(float* scores, int n, int *hist, int numbins,
float *bin_end_values)

{

    int i;

    for(i=0;i<numbins;i++) hist[i] = 0; // histo. with numbins “bins”

    for(i=0;i<n;i++){

        for(j=0;j<numbins;j++){

        // find to which bin, scores[i] belongs

            if (scores[i] <= bin_end_values[j]) { hist[j]++; break;}

        } // close inner for loop with j

    } //close outer for loop with i

} // close function
```

Why not return an array?

- Since arrays are always passed by reference, any changes made to the elements of the array are retained even when the function exits.
- So we passed the array hist in the earlier example as input and all updates will be available when the program returns to the calling function.
- But could we also have the written the function in the manner shown on the next slide?

```
int* create_histogram2(float* scores, int n, int numbins,
float *bin_end_values)
```

- Remember that an array can be referenced by means of a pointer! So can we create a histogram array inside the body of `create_histogram2` and then return that to the calling function?

```
int* create_histogram2(float* scores, int n, int numbins, float *bin_end_values){

    int i;

    int hist[numbins];  // histogram with numbins "bins"

    for(i=0;i<numbins;i++) hist[i] = 0;

    for(i=0;i<n;i++){

        for(j=0;j<numbins;j++){

        // find to which bin, scores[i] belongs

        if (scores[i] <= bin_end_values[j]) { hist[j]++; break;}

        } // close inner for loop with j

    } //close outer for loop with i

    return hist;

} // close function
```

Can we do this?

Why not return an array?

- It is okay to create a new array inside the function.
- The histogram will get correctly updated.
- But when you return `hist` to the calling function, there is a **serious** problem.
- The activation frame of the function `create_histogram2` is **destroyed** once it exits.
- Therefore `hist` will now point to a location which has been deleted, i.e. marked for further allocation.
- This will produce a runtime error or unpredictable outputs.
- **In general, never return a pointer to a local variable, whether or not it is an array.**

Searching within an array: Linear search

- Consider an array `A` with `n` numbers.
- Suppose you want to search for a query number `q` inside this array.
- How do you do this? Just run a `for` loop and check every element against it, in the following way. This is called **linear search**.

```
int linearSearch (int *A, int n, int q){
for (int i = 0; i < n; i++){
    if (q == A[i]) return i;
}
return  -1; // q not found - return an invalid index
}
```

Searching within an array: Linear search

- How many compare operations does linear search take?
- In the worst case, `n` operations, if the query number `q` happened to lie at the last index.
- Assuming `q` could lie at any index from 0 to `n-1` with equal probability, the average number of comparisons will be `(1+2+3+...+n-1+n)/n = n(n+1)/2n = (n+1)/2`.

Searching within a sorted array: Binary search

- Now let us assume that `A` were sorted in increasing order.
- Can you do better than searching element by element?
- The answer is yes: if your algorithm makes use of the fact that the array is sorted.
- Let us check `q` against `A[n/2]`.
- If `q == A[n/2]`, you are done!
- If `q < A[n/2]`, then `q` must necessarily lie in the first half of the array `A` from `0` to `n/2-1`. Note that `q` will be less than any element with index more than `n/2`.
- If `q > A[n/2]`, then `q` must necessarily lie in the second half of the array `A` from `n/2+1` to `n-1`. Note that `q` will be greater than any element with index less than `n/2`.

Searching within a sorted array: Binary search

- This earlier analysis “smells of” recursion :-)
- If `q < A[n/2]`, then you search within the first half of the array.
- If `q > A[n/2]`, then you search within the second half of the array.
- The recursive function for this is on the next slide.
- Note that this is the first time you are writing a recursive function with arrays!
- For every step of recursion, we will maintain a left index and a right index, with the understanding that the array is to be searched only within these limits.
- The left and right indices will be updated at every step of the recursion.

```
int rec_binsearch (int* A, int left, int right, int q){

if (left > right) return -1; // number not found

int mid = (right+left)/2;

if (A[mid] == q) return mid; //query number found

if (q < A[mid]) { cout << left << " " << mid-1 << endl;
return rec_binsearch(A,left,mid-1,q); }// first half of A

if (q > A[mid]) { cout << mid+1 << " " << right << endl;
return rec_binsearch(A,mid+1,right,q); } // second half of A

}
```

Non-recursive implementation of binary search

```
int binsearch (int* A, int left, int right, int q)

{

    while (left <= right){

        int mid = (left+right)/2;

        if (A[mid] == q) return mid;

        if (q < A[mid]) right = mid-1; // first half of A

        if (q > A[mid]) left = mid+1; // second half of A

    }

    return -1;

}
```

How long does binary search take?

- In binary search, we are dividing the sorted array into half each time.
- In the worst case, the number of comparison operations will be equal to the number of times you divide the array size into two, so as to reach 1.
- That is $\log_2(n)$.
- This is the worst case number of comparison operations.
- It turns out that the number of comparisons in the average case is also quite close to $\log_2(n)$, the derivation of which is beyond the scope of our course.
- Binary search is very similar in spirit to another algorithm we have done in this course so far. Which one?