Arrays
Arrays

- Introduction
- Fundamentals
  - Declaring arrays
  - Indexing arrays
  - Initialzing arrays
- Arrays and functions
- Multidimensional arrays
- Sorting and algorithm efficiency
Arrays
Array Introduction

- An array is a sequence of values of the same type
  - Like a list of values
  - Each value is a single element of the list
- An array is a collection of values
  - The entire collection can be referenced, or
  - Its individual elements
Let’s say we want to write a program to input grades for a course

- Assume that 44 students are registered
- Input each grade, one at a time, and store it

Once the grades are input we can use them to calculate the average grade
In a world without arrays (or similar structures) we need a separate variable for each grade:

```java
double grade1 = 0;
double grade2 = 0;
double grade3 = 0;
...
double grade42 = 0;
double grade43 = 0;
double grade44 = 0;
```

Let's assume that these are global variables declared outside the main function.
We need to input the data into each variable
We also need to make sure that grades are entered in the correct variables

```cpp
if (count == 1)
    cin >> grade1;
else if (count == 2)
    cin >> grade2;
...
else if (count == 44)
    cin >> grade44;
```

Assume that input is from the keyboard
This big, ugly IF statement should go in an input function
And could be written as a switch statement
Assume that all of the grades have been entered
  - We can now calculate the average

Let’s make a separate function to do this
  - `double getAverage( ... )`

Ideally, functions should be self contained
  - Input is passed using a parameter list
  - And the desired output is returned
  - But we would need 44 parameters for this function!
    - So, let’s just use the previously defined global variables
    - Which is bad, because global variables
double getAverage (){ 
    double sum = 0;
    sum += grade1;
    sum += grade2;
    sum += grade3;
    sum += grade4;
    sum += grade5;
    sum += grade6;
    sum += grade7;
    sum += grade8;
    sum += grade9;
}

This is what the getAverage function would look like ...
Average Grades ...

// ...
sum += grade10;
sum += grade11;
sum += grade12;
sum += grade13;
sum += grade14;
sum += grade15;
sum += grade16;
sum += grade17;
sum += grade18;
sum += grade19;

You can’t put this in a loop because the variable that is added is different each time
Average Grades ...

// ...
sum += grade20;
sum += grade21;
sum += grade22;
sum += grade23;
sum += grade24;
sum += grade25;
sum += grade26;
sum += grade27;
sum += grade28;
sum += grade29;

Good thing it’s not a class with 250 students in it or this would be really boring
Average Grades ...

```cpp
// ...
sum += grade30;
sum += grade31;
sum += grade32;
sum += grade33;
sum += grade34;
sum += grade35;
sum += grade36;
sum += grade37;
sum += grade38;
sum += grade39;
```
Average Grades ...

// ...

```cpp
    sum += grade40;
    sum += grade41;
    sum += grade42;
    sum += grade43;
    sum += grade44;
    return sum / 44;
```
Here is the same function, this time assuming that the grade data was stored in an array.

```c
double getAverage(double arr[], int size){
    double sum = 0;
    for (int i = 0; i < size; ++i){
        sum += arr[i];
    }
    return sum / size;
}
```
What Is An Array?

- An array variable is a collection of other variables
  - You can think of an array as something that contains variables
  - This is important because an integer array is *not* an integer, it is a *collection* of integers
- The items stored in an array (*elements*) are stored sequentially in main memory
  - This is an important implementation issue
Declaring Arrays

- An array is declared with a type, and []s to indicate that the variable is an array
  - The type is the type of the contents of the array

```c
int score[10];
```

- type of the data stored in the array
- brackets declare the variable as an array
- size of the array
Array Indexing

- The elements of the array are accessed using an *index*
  - Indexes are the addresses of the elements
    - The first element always has an index of 0
    - The last index is always *array size* – 1
- Array indexes follow the name of the array and are enclosed in [ ]s
  - Individual array elements are used in exactly the same way as variables
Using an Array

```c
int arr[4];
int x;
for(int i = 0; i < 4; ++i){
    arr[i] = i + 1;
}
x = arr[2]; //access 3rd. element
```

<table>
<thead>
<tr>
<th></th>
<th>2048</th>
<th>2052</th>
<th>2056</th>
<th>2060</th>
<th>2064</th>
<th>2068</th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[0]</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>arr[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>arr[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

represents stack memory, where each cell represents 4 bytes

result of the statements shown above
Initializing Arrays

- Array elements have to be given values individually
  - Although this can be performed in a loop
- There is a special shorthand notation for initializing arrays when they are declared
  - `int arr[] = {1, 3, 7, 9};`
  - This shorthand notation is only allowed on the same line as the declaration
Array Size

- Array size must be specified when the array is declared
  - It must be a literal (a number) or
  - A constant
  - The size can *not* be given using a variable
- Therefore an array's size cannot change during the life of a program
  - It is specified at *compile time*
Arrays are often processed using loops

- Each element in an array is accessed in turn
- The loop control variable is used as an index into the array
- The loop terminates once every element of the array has been processed
  - When the loop control variable is equal to the size of the array
// Assume an int array named arr,
// and a constant called ARR_SIZE
// The loop prints the contents of arr

for (int i = 0; i < ARR_SIZE; i++){
    cout << arr[i] << endl;
}

The condition is \( i < ARR\_SIZE \) because the last legal index is \( ARR\_SIZE - 1 \)
// Assume an int array named arr, 
// and a constant called ARR_SIZE 
// The loop prints the contents of arr

for (int x : arr){
    cout << x << endl;
}

This alternate form of the for loop is designed to iterate through containers

Introduced in C++ 11, so may require a flag to compile

Only works with static arrays – i.e. arrays discussed to date in this presentation
// Assume an int array named arr, 
// and a constant called ARR_SIZE 
// The loop prints the contents of arr

int i = 0;
while (i < ARR_SIZE){
    cout << arr[i] << endl;
    i++;
}
Size as a Constant

- It is much better to define array size as a constant than to use a literal value.
- If the programmer wants to change the array size this only needs to be done once.
  - Use the constant whenever the array size is referenced, and
  - Avoid using magic numbers!
What happens if a program attempts to access a non-legal index?

Some sort of error occurs, either

- An illegal attempt to access a memory location has been made (… stack is corrupted …), or
- Something less predictable

Always ensure that an index is legal

- Between 0 and array size – 1
Aside – Using Arrays

- Make sure that you distinguish between an array and its contents
  - `int arr[4];`
    - `arr` is the entire array
    - `arr[1]` is one element of the array
- The array is a container of integers, one element is an integer
  - They are not interchangeable
It is often important to know how much of an array is used

- Unassigned elements shouldn't be processed
  - e.g. summing or calculating the average of an array

Consider what input a function requires

- The array, and its maximum size
- The array, and current size
  - i.e. the number of elements actually used
- The array, its maximum size, and its current size
Arrays and Functions
Arrays arguments can be passed to functions
  - So parameter lists can include arrays
  - The array type, and the fact that it is an array must be specified in the parameter list
    - An array is specified just like any declaration

A function header for an array sum function
  - `int sumArray(int arr[], int n){`
// Sums the array, returning the sum
int sumArray(int arr[], int n){
    int sum = 0;
    for (int i = 0; i < n; i++){
        sum += arr[i];
    }
    return sum;
}
A common array task is to search an array for a particular value.

```c
int search(int arr[], int n, int x){
    for (int i = 0; i < n; i++){
        if (arr[i] == x)
            return i;
    }
    return -1;
}
```

Return the index of the target as soon as it is found.
Return -1 if the target is not found, as it is not a valid array index.
What happens if you pass an array to a function that changes the array parameter?

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

The function doubles each of the values stored in the array parameter.
const int ARR_SIZE = 3;
int main()
{
  int arr[] = {1, 2, 3};
  doubleArray(arr, ARR_SIZE);
  for(int i = 0; i < 3; ++i){
    cout << arr[i] << endl;
  }
  return 0;
}
To understand what is going on in the previous example we need to know more

- It's easy to think of an array as a container, like a bookcase
- But a bookcase is a structure in its own right

- An array is just a collection of values
  - All of the same type, and
  - Stored in sequence in main memory
More About Arrays

- An array is a sequence of bytes in main memory reserved for the array contents
  - e.g. `int arr[10];`
    - Reserves 40 contiguous bytes (4 byte `ints`, 4 * 10 = 40)
    - Each element can be referenced using indexes
- And `arr` stores the address of the first array element
  - These two statements print the same address
    - `cout << arr << endl;`
    - `cout << &arr[0] << endl;`
Passing an Array to a Function

- An array variable stores the address of the array's first element
  - Referred to as a *pointer* to the array
  - The index operator ([][]) allows access to the element of the array stored at the given index
- Passing an array variable to the function gives the function the address of the array
  - In a similar way to pass by reference
    - Although the mechanism is somewhat different
Consider this assignment statement:
- \texttt{arr[8] = 23;}

To find this array element
- Look up the address stored in \texttt{arr}
- Multiply type size (4 for an \textit{int}) by the index
- Add this to the address to find the element
- Known as an \textit{offset calculation}
Be careful when specifying loop conditions
- Particularly when working with arrays
- Being off by one with an index value may result in a runtime error
  - Or worse ...

Where possible use `<`, `>`, `<=`, or `>=` as conditions rather than `==` or `!=`
- As an example of this we will look at a function to reverse the contents of an array
// Function to print a small integer array on one line
void printArray(int arr[], int n)
{
    cout << "\{";
    for(int i=0; i < n; i++)
    {
        cout << arr[i];
        if(i != n){
            cout << ",";
        }
    }
    cout << "}";
}

Note that i != n is not used here

Commas separate array elements
Now let's write the function to reverse an array

```c
// Function to reverse the contents of an array
void reverse(int arr[], int size)
{
    int lo, hi;
    for(lo=0, hi=size-1; lo!=hi; lo++, hi--){
        int temp = arr[lo];
        arr[lo] = arr[hi];
        arr[hi] = temp;
    }
}
```

This version is broken!
Comma operator
Testing Reverse

```c
int arr[] = {1,2,3,4,5,6,7};
int arr_size = sizeof(arr)/sizeof(int);
printArray(arr, arr_size);
reverse(arr, arr_size);
cout << endl << endl;
printArray(arr, arr_size);
```

so far, so good ...

```c
int arr[] = {1,2,3,4,5,6,7,8};
int arr_size = sizeof(arr)/sizeof(int);
printArray(arr, arr_size);
reverse(arr, arr_size);
cout << endl << endl;
printArray(arr, arr_size);
```

they didn`t ...

the for condition should have been lo < hi
Array Size Revisited

- As noted previously array size must be given a constant value
  - And the size of an array cannot be changed while the program is running (during run-time)
  - This is to support the organization of stack memory
- It's possible to allocate memory at run-time
  - From a free store of memory, called dynamic memory

more on this later ...
Arrays can be returned from a function

- However the obvious way of specifying the return type is *illegal*
  - `int[]` `getAnArray()` {
    ...
  }

- In addition, returning an array raises the question of how big it is

  This doesn't work

  more on this later ...
Multi-Dimensional Arrays
Multi Dimensional Arrays

- Assume we want to store monthly rainfall in Vancouver for five years
  - So sixty data points, in an array of size 60
  - What is the index for September of year 3?
- It would be more convenient to store the data in a table
  - With years as columns and months as rows
- This is possible using a multidimensional array
To declare a multidimensional array you declare an array of arrays

- `float rainfall[5][12];`
  - It is an array of 5 arrays of size 12
  - Where `rainfall[0]` is the first array of size 12

To access elements of the inner array, use two indices

- So September of year 3 is `rainfall[2][8]`
  - Remember 0 based indexing
To initialize a multidimensional array use nested for loops

- Or the same initialization syntax as regular arrays

- Note that multidimensional arrays are not limited to two dimensions
```cpp
void testMatrix()
{
    int matrix[3][5] =
    {
        {1,2,3,4,5},
        {11,12,13,14,15},
        {21,22,23,24,25}
    };

    int row, column;

    for(row = 0; row < 3; ++row)
    {
        for(column = 0; column < 5; ++column)
        {
            cout << setw(4) << matrix[row][column];
        }
        cout << endl;
    }
}
```

don’t need the inner brackets – but they make it easier to read
Sorting and Efficiency
It is sometimes useful that a function changes an array passed to it
- A common example of this is *sorting* an array
- There are many different sorting algorithms
  - Selection sort, Insertion sort, Bubble sort, ...
  - Quicksort, Mergesort, Radix sort, ...
- The algorithms differ in how fast they are
  - And how much main memory space they require
  - And how they are affected by the input organization
Selection sort is a simple sorting algorithm that repeatedly finds the smallest item.

- The array is divided into a sorted part and an unsorted part.
- Repeatedly swap the first unsorted item with the smallest unsorted item.
  - Starting with the element with index 0, and
  - Ending with last but one element (index $n - 1$).
## Selection Sort

<table>
<thead>
<tr>
<th>23</th>
<th>41</th>
<th>33</th>
<th>81</th>
<th>07</th>
<th>19</th>
<th>11</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>find smallest unsorted - 7 comparisons, swap with 23</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>41</td>
<td>33</td>
<td>81</td>
<td>23</td>
<td>19</td>
<td>11</td>
<td>45</td>
</tr>
<tr>
<td><strong>find smallest unsorted - 6 comparisons, swap with 41</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>11</td>
<td>33</td>
<td>81</td>
<td>23</td>
<td>19</td>
<td>41</td>
<td>45</td>
</tr>
<tr>
<td><strong>find smallest unsorted - 5 comparisons, swap with 33</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>11</td>
<td>19</td>
<td>81</td>
<td>23</td>
<td>33</td>
<td>41</td>
<td>45</td>
</tr>
<tr>
<td><strong>find smallest unsorted - 4 comparisons, swap with 81</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>11</td>
<td>19</td>
<td>23</td>
<td>81</td>
<td>33</td>
<td>41</td>
<td>45</td>
</tr>
<tr>
<td><strong>find smallest unsorted - 3 comparisons, swap with 81</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>11</td>
<td>19</td>
<td>23</td>
<td>33</td>
<td>81</td>
<td>41</td>
<td>45</td>
</tr>
<tr>
<td><strong>find smallest unsorted - 2 comparisons, swap with 81</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>11</td>
<td>19</td>
<td>23</td>
<td>33</td>
<td>41</td>
<td>81</td>
<td>45</td>
</tr>
<tr>
<td><strong>find smallest unsorted - 1 comparison , swap with 81</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>11</td>
<td>19</td>
<td>23</td>
<td>33</td>
<td>41</td>
<td>45</td>
<td><strong>81</strong></td>
</tr>
<tr>
<td><strong>done</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
void selectionSort(int arr[], int n){
    for(int i = 0; i < n - 1; ++i){
        int smallest = i;
        // Find the index of the smallest element
        for(int j = i + 1; j < n; ++j){
            if(arr[j] < arr[smallest]){
                smallest = j;
            }
        }
        // Swap the smallest with the current item
        int temp = arr[i];
        arr[i] = arr[smallest];
        arr[smallest] = temp;
    }
}

outer loop

inner loop
int getSmall(int arr[], int start, int end){
    int smallest = start;
    for(int i = start + 1; i < end; ++i){
        if(arr[i] < arr[smallest]){
            smallest = i;
        }
    }
    return smallest;
}

void swap(int arr[], int i, int j){
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

void selectionSort(int arr[], int n){
    for(int i = 0; i < n -1; ++i){
        int smallest = getSmall(arr, i, n);
        swap(arr, i, smallest);
    }
}
Algorithm Efficiency

- When designing an algorithm its efficiency is an important consideration
  - i.e. How fast does it run?
- The speed of an algorithm is affected by a number of factors
  - The computer
  - The algorithm
  - The Input
The computer that an algorithm is run on will affect its speed

- How fast is the processor?
- How much main memory does it have?
- What else is it doing at the same time?
- Does the algorithm require specialized hardware?
  - Such as a graphics card (GPU)
  - If so, how powerful is this hardware?
The input to the algorithm will also affect its speed

- What is its size?
- What kind of container is it stored in?
- How is that container implemented?
- What is its organization?
When assessing algorithm efficiency it is often useful to ignore

- The computer it is run on and
- Any one actual input size

The efficiency of an algorithm can be expressed as a formula
- A function of the input size, which is referred to by a variable, $n$
Examples

- Let’s look at some examples
  - Linear search
  - Reverse
  - Selection sort

- We'll start by running them on a computer with varying sizes of input
  - Making some preliminary observations
  - And then reasoning about them