MOTIVATION

Did you see *Inception* this weekend?

float movieRating1 = 1.0;
float movieRating2 = 3.5;
float movieRating3 = 1.0;
float movieRating4 = 4.0;
float movieRating5 = 4.0;
float movieRating6 = 3.0;
... 
float movieRating25 = 2.5;

What’s the average rating?

float averageRating = (movieRating1 + movieRating2 + ... + movieRating25) / 25.0;

What’s the lowest rating?

float minRating = 4.0;
if(movieRating1 < minRating)
minRating = movieRating1;
if(movieRating2 < minRating)
minRating = movieRating2;
...
if(movieRating25 < minRating)
minRating = movieRating25;

It is a common scenario in a program to have a collection of data that we need to do something to.

Could store as separate variables:

An *array* is a more practical way of storing similar data:

Arrays let us store and access data collections of a certain type.

- Arrays: Store and access data collections of a certain type
- Declaration, access
- As function parameters
- Applications
- Lab 5 preview:
  - Storing vertices
  - Visualization: LEDA
  - Makefile
- todaysOutline

- Arrays
- Lab 5 preview:
  - Storing vertices
  - Visualization: LEDA
  - Makefile

TODAY'S OUTLINE

Arrays
Declaration, access
As function parameters
Applications
Lab 5 preview:
Storing vertices
Visualization: LEDA
Makefile
Each value is identified by an index

numRatings = 3  // an int variable
movieRatings[] = 3.0, 2.5, 1.5  // an array of floats
counts[] = 3, 1, 0  // an array of ints
classNames[] = "EE", "CS", "Math"  // an array of Strings

CREATING ARRAYS

float movieRatings[];  // declares an array of 3 floats
const int numRatings = 3;
float movieRatings[numRatings];  // an equivalent declaration

ARRAY DECLARATION SYNTAX

- movieRatings is of type array of float with size 3
- movieRatings[0], movieRatings[1], movieRatings[2] are the elements of the array — each is a variable of type float
- 0, 1, 2 are the indices of the array (a.k.a. subscripts)
- bounds: the lowest and highest values of the subscripts (here: 0 and 2)

ACCESSING ARRAY ELEMENTS

float movieRatings[];  // declares an array of 3 floats
movieRatings[0] = 3.0;            // assign values to elements
movieRatings[1] = 2.5;
movieRatings[2] = 1.5;

INITIALIZING ARRAY ELEMENTS

Typically want to assign default values before doing anything:

float movieRatings[];  // declares an array of 3 floats
movieRatings[0] = 0.0;            // initialize array elements
movieRatings[1] = 0.0;
movieRatings[2] = 0.0;

float movieRatings[] = {0.0, 0.0, 0.0};

STORING AND RETRIEVING DATA

An array is a collection of variables

Each element can be used wherever a simple variable of that type is allowed: assignment, expressions, input/output, etc.

```cpp
movieRatings[2] = 1.5;
cout << movieRatings[2] << endl;
float myRating = movieRatings[0];
cout << myRating << endl;
if (movieRatings[1] == movieRatings[2]) {
    cout << "rating 1 equals rating 2" << endl;
}
```

What's wrong with this code?

```cpp
float movieRatings[3]; //declares an array of 3 floats
movieRatings[0] = 3.0; //assign values to elements
movieRatings[1] = 2.5;
movieRatings[2] = 1.5;
```

What's wrong with this code?

```cpp
float movieRatings[3]; //declares an array of 3 floats
movieRatings[0] = 3.0; //assign values to elements
movieRatings[1] = 2.5;
movieRatings[2] = 1.5;
```

What's wrong with this code?

```cpp
float movieRatings[3]; //declares an array of 3 floats
movieRatings[0] = 3.0; //assign values to elements
movieRatings[1] = 2.5;
movieRatings[2] = 1.5;
```

What's wrong with this code?

```cpp
float movieRatings[3]; //declares an array of 3 floats
movieRatings[0] = 3.0; //assign values to elements
movieRatings[1] = 2.5;
movieRatings[2] = 1.5;
```

What's wrong with this code?
What’s wrong with this code?

```cpp
float movieRatings[3]; // declares an array of 3 floats
movieRatings[0] = 3.0; // assign values to elements
movieRatings[1] = 2.5;
movieRatings[2] = 1.5;
movieRatings[3] = 3.0;
```

Error: Array out of bounds!

INDEX RULE

If an array has \( n \) elements, then an array index must evaluate to an int between 0 and \( n-1 \)

```cpp
float movieRatings[3]; // declares an array of 3 floats

movieRatings[(i+3*k)] = 3.0; // OR if 0 <= (i+3*k) <= 2
movieRatings[i] = 3.0; // the index may be simple or complex
movieRatings[(int)(abs(sin(2.0*PI*sqrt(29.067))))] = 3.0;
```

No exceptions!

STORING AND RETRIEVING DATA

An array is a collection of variables

Each element can be used wherever a simple variable of that type is allowed: assignment, expressions, input/output, etc.

An entire array can’t be treated as a single variable in C++

  - Can’t assign or compare entire arrays using `=`, `==`, `<`, ...
  - Can’t use cout or cin on an entire array

But, you can do these things one element at a time

TRAVERSING AN ARRAY

Loops are useful for traversing an array, or accessing each of its elements in sequence

```cpp
float movieRatings[3];

for (int i = 0; i < 3; i++)
{
    cout << "Rating " << i << " = " << movieRatings[i] << endl;
}
```

RAINFALL DATA

General task: Read daily rainfall amounts and print some interesting information about them

Input data: Zero or more measurements, followed by a sentinel

Example input data:

```
1.0 0.2 0.0 0.0 1.4 0.1 0.0 -1.0
```

Empty input sequence:

```
-1.0
```

A named, ordered collection of variables of identical type

Rainfall for 1 week:

Name the collection (rain); number the elements (0 to 6)
Example declaration:
float rain[7];

Example access:
rain[0] has value 1.0
rain[6] has value 0.0
2*rain[4] has value 2.8

Since an array has to be declared a fixed size, you often declare it bigger than you think you will really need.

const int MAX_RAIN_DAYS=365;
float rain[MAX_RAIN_DAYS];

How do you know which elements in the array actually hold data, and which are unused extras?

Problem:
Calculate and print number of days for which rainfall is above average.

Sample output:
Please enter rainfall data for this week:
1.0 0.2 0.0 1.4 0.1 0.0 -1.0
Average: 0.386
There were 2 days with above average rain fall.

Algorithm:

const int ARRAY_SIZE=10
double average ( int a[], int num)  {
int i, total = 0 ;
for ( i = 0 ;  i < num ;  i++ ) {
total = total + a[i]  ;
}
return ((double) total / (double) num) ;
}

int main ()
{...
float rain_avg = average ( rain, numRainDays ) ;
}...
ARRAYS AS PARAMETERS

Array parameters (entire arrays) do not work like simple variables:

- An array is never copied
- The array name is always treated as a pointer

In C++, arrays do not contain information about their size, so the size often needs to be passed as an additional parameter

DATA STRUCTURES

An array is an important data structure

- Functions give us a way to organize programs

Data structures are used to organize data, especially:

- Large amounts of data
- Variable amounts of data
- Sets of data where the individual pieces are related to one another

LAB 5 PREVIEW: N-SIDED POLYGONS

Use arrays to store vertices
**ARRAYS OF VERTICES**

Two arrays of size MAX_NUM_VERTICES

```cpp
const int MAX_NUM_VERTICES = 24;
int x[MAX_NUM_VERTICES];
int y[MAX_NUM_VERTICES];
```

For each garden: iterate over number of vertices
Read x and y coordinate
Store in the x[ ] and y[ ] arrays

---

**VISUALIZATION: LEDA**

LEDA: a platform for combinatorial and geometric computing

We will use only the graphics functionalities

Need to tell the shell how/where to find LEDA

Type at the command line:

```
use leda
```

---

**VISUALIZATION: LEDA**

Functions provided in `visualize.cpp`:

```cpp
window_open();
window_close();
wait (int seconds);
draw_trapezoid (int xi, int xf, int yi, int yf)
```

---

**MAKEFILE**

Compilation and linking getting more complicated

Multiple source files: `poly.cpp` and `visualize.cpp`

External libraries

Use a Makefile:

Describes how to compile all the code files, what to link together and where to find external dependencies

---

**MAKEFILE DESCRIBES DEPENDENCIES**

![Diagram of makefile dependencies]

Source code (text) Object code (binary)

---

**COMPIRING AND LINKING**

To compile and link, type at the command line:

```
make
```

(Note: g++ `poly.cpp` will not work!)

What happens:

```
g++ -o poly -I/usr/cots/leda-6.1/incl poly.cpp
visualize.cpp
-L/usr/cots/leda-6.1 -lleda -L/usr/X11 -lx11 -lm
```
**SUMMARY**

Array data structure
- Groups data of same type
- Individual elements behave like simple variables
- Can be a function parameter
- Use loops

Lab 5
LEDA
make and Makefile

**MULTIDIMENSIONAL ARRAYS**

Use more than one index to access array elements

2D: an "array of arrays"

Loop through rows and columns

**Digital images can be stored as 2D arrays**

**Images Processing**

What happens if we look at the difference between two images?

A - B = ?

**Images Processing**

A - B = D
Frames
Motion squared absolute differences between sequential frames
Foreground differences from a background frame (smoothed)