Arrays

- basic use of 1D arrays
- passing as parameters
Allen Telescope Array (ATA)
Wind Turbines
The Power of Wind
Why Arrays?

• Assume we need:
  – 50 integer scores
  – some data analysis on them

• OPTION 1: declare 50 int variables?
  int score1, score2, score3, … score50;

  inflexible tedious hard to access them all e.g. need 50 statements to read each one
Other Options?

- OPTION 2: store them in one array
const int NUMSCORES = 50;

int main()
{
    int scores[NUMSCORES];
}

number of elements (size) integral, constant
basetype
scores [ 0 ]

scores [ 1 ]

scores [ 49 ]

50 consecutive memory locations

index is offset from base index
Arrays

- `int scores[] = {30, 40, 39, 35, 14, 21};`

Score

<table>
<thead>
<tr>
<th>30</th>
<th>40</th>
<th>39</th>
<th>35</th>
<th>14</th>
<th>21</th>
</tr>
</thead>
</table>

SIX items           ONE name               use SUBSCRIPTS to access
How to get data into array?

```cpp
int scores[5];    // declare array

scores[0] = 42;       // assign value into it
scores[1] = scores[0] + 5;
cin >> scores[0];     // read value into it

for (int index = 0; index < 5; index++) {
    scores[index] = 2 * index;    // calculate
}
```
Range Errors

- C++ does NOT check array indices for validity
- range error:
  - using an out of range index
  - here, valid index range: 0 - 49
- range errors can:
  - overwrite data in program
  - can overwrite program
  - can even overwrite OS code
for ( count = 0; count <= NUMSCORES; count++)
    inFile >> scores[ count];

most likely: count is overwritten by the last score read, if the file contains a 51st value. (What happens then depends on what that last exam score actually was)

Remember variable declarations:
int sum = 0,
mean,
scores[NUMSCORES],
count;

ifstream inFile;
scores[0]

scores[1]

scores[49]

index is offset from base index

"scores[50]"

consecutive memory locations

count
Operations on arrays

- int arr1[5], arr2[5];

- I/O (cin >> arr1; )  NO
- Assignment (arr1 = arr2; )  NO
- Arithmetic (arr1 = arr2 + 6; )  NO
- Comparison (if (arr1 < arr2 ))  NO
- Pass as Argument  YES
  - by reference only
- Return as function result (return arr1 )  NO
Passing Arrays to functions

- C++ arrays are:
  - always passed by reference
  - saves memory space
    - array is not copied
  - saves processing time
    - array elements are not copied
the Name of an Array ...

- refers to the entire data structure
- is a pointer constant
- scores same as the memory address of the 1st array element

address operator

- scores == & scores[0]

data type const int *
Example Call (assume main is caller):

```c
readScoreList ( scores );
```

Possible Corresponding Prototypes:

```c
void readScoreList ( int scores[NUMSCORES] );
void readScoreList ( int scores [ ] );
void readScoreList ( int * scores );
```
Passing by \textit{const} reference

- passing arrays as input-only
- in lieu of true pass by value

\texttt{printResults (scores);}

\texttt{void printResults(const int scores[]);}  

- new passing method for us: argument is not copied, but callee is not allowed to alter it
Initializing Array Elements in Declaration

• Declare array with 5 elements, and initialize all 5 specifically
  \[
  \text{int numbers [ 5 ] = \{ 3, 5, 7, 9, 11 \};}
  \]

• Declare array of 3 initialized elements
  \[
  \text{int numbers [ ] = \{ 1, 3, 8 \};}
  \]

• Declare array with 100 elements, and initialize all elements to zero
  \[
  \text{int numbers [ 100 ] = \{ 0 \};}
  \]
Summary

• Arrays are very useful for set of like data
  – 0-based
  – Be careful not to write past end of array (range error)
  – ALL elements must be the same datatype

• Arrays are always passed by reference
  – Can be modified in called function
  – `void readScoreList (int scores [])`;  //proto
    `readScoreList (scores);`  //call

• Arrays may be passed by const ref – very useful
  – Cannot be modified in called function
  – `void calcMeanScore(const int scores[]);`  //proto
    `calcMeanScore (scores);`  //call