CSCE 121
Introduction to Program Design & Concepts
Array Processing

Dr. Tim McGuire

Grateful acknowledgment to Dr. Philip Ritchey and Dr. Michael Moore for some of the material on which these slides are based.

Arrays Hold Multiple Values

- Array: variable that can store multiple values of the same type
- Values are stored in adjacent memory locations
- Declared using `[]` operator:

```
int tests[5];
```

Array Terminology

In the definition `int tests[5];`

- `int` is the data type of the array elements
- `tests` is the name of the array
- `5,` in `[5],` is the size declarator. It shows the number of elements in the array.

Size Declarators

- Named constants are commonly used as size declarators.

```
const int SIZE = 5;
int tests[SIZE];
```

- This eases program maintenance when the size of the array needs to be changed.

Accessing Array Elements

- Array elements can be used as regular variables:

```
tests[0] = 79;
cout << tests[0];
cin >> tests[1];
tests[4] = tests[0] + tests[1];
```

- Arrays must be accessed via individual elements:

```
cout << tests; // not legal
```

Un-initialized Arrays

- If no values are listed in the array declaration, some compilers will initialize each element to zero.
  - DO NOT DEPEND ON THIS!

Array Initialization

- Arrays can be initialized with an initialization list:

```
int tests[] = {79,82,91,77,84};
```

- The values are stored in the array in the order in which they appear in the list.
- The size of the list is determined by the number of elements in the initializer list.

No Bounds Checking in C++

- When you use a value as an array subscript, C++ does not check it to make sure it is a *valid* subscript.

- In other words, you can use subscripts that are beyond the bounds of the array.

No Bounds Checking in C++

- Be careful not to use invalid subscripts.
- Doing so can corrupt other memory locations, crash program, or lock up computer, and cause elusive bugs.

No Bounds Checking in C++

```
// This program unsafely accesses an area of memory
// by writing values beyond an array's boundary.
// WARNING: If you compile/run this program, it could crash.
#include <iostream>
using namespace std;

int main()
{
   const int SIZE = 3;  // Constant for the array size
   int values[SIZE];    // An array of 3 integers
   int moveVals[SIZE];
   int count;           // Loop counter variable

   // Attempt to store 5 numbers in the three-element
array.
   cout << "I will store 5 numbers in a 3 element
array!\n";
 for (count = 0; count < 5; count++)
      values[count] = count * 10;

   // If the program is still running, display the numbers.
   cout << "If you see this message, it means the program\n";
   cout << "has not crashed! Here are the numbers:\n";
   for (count = 0; count < 5; count++)
      cout << values[count] << endl;
   return 0;
}
```

Off-By-One Errors

- An off-by-one error happens when you use array subscripts that are off by one.
- This can happen when you start subscripts at 1 rather than 0:

```
// This code has an off-by-one error.
const int SIZE = 100;
int numbers[SIZE];
for (int count = 1; count <= SIZE; count++)
   numbers[count] = 0;
```

The Range-Based `for` Loop

- C++17 provides a specialized version of the `for` loop that, in many circumstances, simplifies array processing.
- *The range-based* `for` *loop is a loop that iterates once for each element in an array.*
- *Each time the loop iterates, it copies an element from the array to a built-in variable, known as the range variable.*
- The range-based `for` loop automatically knows the number of elements in an array.
  - You do not have to use a counter variable.
  - You do not have to worry about stepping outside the bounds of the array.

Range-Based For Loops

- Here is the general format of the range-based for loop:

```
for (datatype varname : array)
{
        // varname is successively set to each
        // element in the array
        statement;
}
```

- ***datatype*** is the data type of the range variable.
- ***varname*** is the name of the range variable. This variable will receive the value of a different array element during each loop iteration.
- ***array*** is the name of an array on which you wish the loop to operate.
- ***statement*** is a statement that executes during a loop iteration. If you need to execute more than one statement in the loop, enclose the statements in a set of braces.

Range-Based For Loop Example

- The following code outputs 2 3 5 7 11 13 17

```
int array[] = {2, 3, 5, 7, 11, 13, 17};
for (int n : array)
     cout << n << " ";
cout << endl;
```

- Note that your array must be completely filled with valid values because this prints every element in the array.

The Range-Based `for` Loop versus the Regular `for` Loop

- The range-based for loop can be used in any situation where you need to step through the elements of an array, and you do not need to use the element subscripts.

- If you need the element subscript for some purpose, use the regular `for` loop.

Processing Array Contents

- Array elements can be treated as ordinary variables of the same type as the array

- When using ++, −− operators, don't confuse the element with the subscript:

```
tests[i]++; // add 1 to tests[i]
tests[i++]; // increment i, no
            // effect on tests
```

Array Assignment

To copy one array to another,

- Don’t try to assign one array to the other:

```
newTests = tests;  // Won't work
```

- Instead, assign element-by-element:

```
for (i = 0; i < ARRAY_SIZE; i++)
    newTests[i] = tests[i];
```

Printing the Contents of an Array

- You can display the contents of a *character* array by sending its name to cout:

```
char fName[] = "Henry";
 cout << fName << endl;
```

But, this ONLY works with character arrays!

Printing the Contents of an Array

- For other types of arrays, you must print element-by-element:

```
for (i = 0; i < ARRAY_SIZE; i++)
    cout << numbers[i] << endl;
```

Printing the Contents of an Array

- In C++17 you can use the range-based `for` loop to display an array's contents, as shown here:

```
for (int val : numbers)
    cout << val << endl;
```

- If we wanted to print the array in reverse order, we would use the traditional for loop :

```
for (i = ARRAY_SIZE-1; i >= 0; --i)
  cout << numbers[i] << endl;
```

See file-reverse.cpp for an example of using arrays with files.

Summing and Averaging Array Elements

- Use a simple loop to add together array elements:

```
int tnum;
double average, sum = 0;
for(tnum = 0; tnum < SIZE; tnum++)
        sum += tests[tnum];
```

- Once summed, can compute average:

```
average = sum / SIZE;
```

Summing and Averaging Array Elements

- In C++17 you can use the range-based `for` loop, as shown here:

```
double total = 0;  // Initialize accumulator
double average;    // Will hold the average
for (int val : scores)
    total += val;
average = total / NUM_SCORES;
```

Finding the Highest Value in an Array

```
int count;
int highest;
highest = numbers[0];
for (count = 1; count < SIZE; count++)
{
   if (numbers[count] > highest)
      highest = numbers[count];
}
```

When this code is finished, the `highest` variable will contains the highest value in the `numbers` array.

Finding the Lowest Value in an Array

```
int count;
int lowest;
lowest = numbers[0];
for (count = 1; count < SIZE; count++)
{
   if (numbers[count] < lowest)
      lowest = numbers[count];
}
```

When this code is finished, the `lowest` variable will contains the lowest value in the `numbers` array.

Comparing Arrays

- To compare two arrays, you must compare element-by-element:

```
const int SIZE = 5;
int firstArray[SIZE] = { 5, 10, 15, 20, 25 };
int secondArray[SIZE] = { 5, 10, 15, 20, 25 };
bool arraysEqual = true; // Flag variable
int count = 0;           // Loop counter variable
// Compare the two arrays.
while (arraysEqual && count < SIZE)
{
   if (firstArray[count] != secondArray[count])
      arraysEqual = false;
   count++;
}
if (arraysEqual)
   cout << "The arrays are equal.\n";
else
   cout << "The arrays are not equal.\n";
```

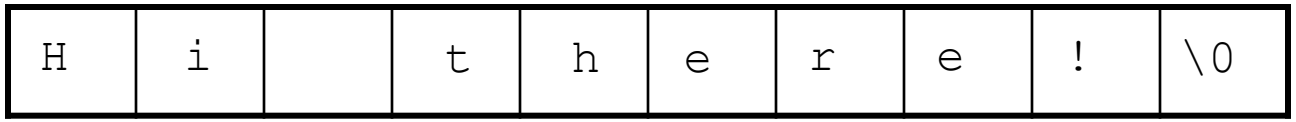
Partially-Filled Arrays

- If it is unknown how much data an array will be holding:
  - Make the array large enough to hold the largest expected number of elements.
  - Use a counter variable to keep track of the number of items stored in the array.

C-Strings

- C-string: sequence of characters stored in adjacent memory locations and terminated by `NULL` character
- String literal (string constant): sequence of characters enclosed in double quotes " " :

```
"Hi there!"
```



C-Strings

- Array of `char`s can be used to define storage for string:
```
const int SIZE = 20;
char city[SIZE];
```

- Leave room for `NULL` at end
- Can enter a value using `cin >>`
  - Input is whitespace-terminated
  - No check to see if enough space
- For input containing whitespace, and to control amount of input, use `cin.getline()`

Library Functions for Working with C-Strings

- Require the `cstring` header file

- Functions take one or more C-strings as arguments. Can use:
  - C-string name
  - pointer to C-string
  - literal string

Library Functions for Working with C-Strings

Functions:

- `strlen(str)`: returns length of C-string `str`

```
char city[SIZE] = "Missoula";
cout << strlen(city); // prints 8
```

- `strcat(str1, str2)`: appends `str2` to the end of `str1`

```
char location[SIZE] = "Missoula, ";
char state[3] = "MT";
strcat(location, state);
// location now has "Missoula, MT"
```

Library Functions for Working with C-Strings

Functions:

- `strcpy(str1, str2)`: copies `str2` to `str1`

```
const int SIZE = 20;
char fname[SIZE] = "Maureen", name[SIZE];
strcpy(name, fname);
```

Note: `strcat` and `strcpy` perform no bounds checking to determine if there is enough space in receiving character array to hold the string it is being assigned.

C-string Inside a C-string

Function:

- `strstr(str1, str2)`: finds the first occurrence of `str2` in `str1`. Returns a pointer to match, or `NULL` if no match.

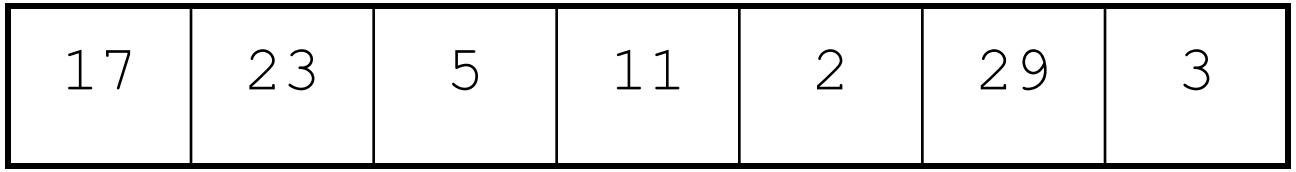
```
char river[] = "Wabash";
char word[] = "aba";
cout << strstr(state, word);
// displays "abash"
```

Searching Arrays

- A sequential search is one way to search an array for a given value
  - Look at each element from first to last to see if the target value is equal to any of the array elements
  - The index of the target value can be returned to indicate where the value was found in the array
  - A value of -1 can be returned if the value was not found

Linear Search - Example

- Array `arr` contains:



- Searching for the the value `11`, linear search examines `17, 23, 5,` and `11`
- Searching for the the value `7`, linear search examines `17, 23, 5, 11, 2, 29,` and `3`

The search Function

- The search function
  - Uses a while loop to compare array elements to the target value
  - Sets a variable of type bool to true if the target value is found, ending the loop
  - Checks the boolean variable when the loop ends to see if the target value was found
  - Returns the index of the target value if found, otherwise returns -1

The search Function

```
int search(const int a[], int size, int target)
{
    int index = 0;
    bool found = false;
    while ((!found) && (index < size)) {
        if (target == a[index])
            found = true;
        else
            index++;
    }
    if (found)
        return index;
    else
        return -1;
}
```

Linear Search - Tradeoffs

- Benefits:
  - Easy algorithm to understand
  - Array can be in any order

- Disadvantages:
  - Inefficient (slow): for array of N elements, examines N/2 elements on average for value in array, N elements for value not in array

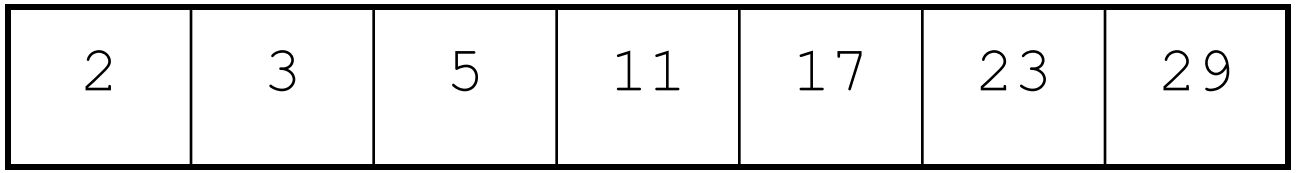
A look ahead to CSCE 221: Binary Search

Requires array elements to be in order

1. Divides the array into three sections:
   - middle element
   - elements on one side of the middle element
   - elements on the other side of the middle element
2. If the middle element is the correct value, done. Otherwise, go to step 1. using only the half of the array that may contain the correct value.
3. Continue steps 1. and 2. until either the value is found or there are no more elements to examine

Binary Search - Example

- Array `ordered` contains:



- Searching for the value `11`, binary search examines `11` and stops
- Searching for the value `7`, linear search examines `11, 3, 5,` and stops

Binary Search

Set first index to 0.
Set last index to the last subscript in the array.
Set found to false.
Set position to -1.
While found is not true and first is less than or equal to last
    Set middle to the subscript half-way between array[first] and array[last].
    If array[middle] equals the desired value
        Set found to true.
        Set position to middle.
    Else If array[middle] is greater than the desired value
        Set last to middle - 1.
    Else
        Set first to middle + 1.
    End If.
End While.
Return position.

Binary Search - Tradeoffs

- Benefits:
  - Much more efficient than linear search. For example, if an array has 1000 elements, performs at most 10 comparisons, as opposed to 1000 for linear search
  - (For 1,000,000 elements, only 20 comparisons!)

- Disadvantages:
  - Requires that array elements be sorted

Parallel Arrays

- zyBook calls them multiple arrays.
- Conceptually a set of items with multiple types of related information
  - People
    - Name
    - Age
    - Weight
  - Separate arrays for name, age and weight.
  - Each index maps to the same person.
- Alternately use an array of structs

Parallel Array Example

```
const int SIZE = 5;    // Array size
int id[SIZE];          // student ID
double average[SIZE]; // student's average
char grade[SIZE];      // course grade
...
for(int i = 0; i < SIZE; i++)
{
    cout << "Student ID: " << id[i]
         << " average: " << average[i]
         << " grade: " << grade[i]
         << endl;
}
```

See parallel-arrays.cpp

Same Example with an Array of Structs

```
const int SIZE = 5;   // Array size
struct Student {
   int id;      // student ID
   double avg; // student's average
   char grade; // course grade
};
// array of structs
struct Student st[5];
  . . .
```

```
for(int i = 0; i < SIZE; i++) {
  cout << "Student ID: " << st[i].id
       << " average: " << st[i].avg
       << " grade: " << st[i].grade
       << endl;
  }
```

Multi-dimensional Arrays

- Matrix calculations
- Multi-dimensional data
- An array of arrays
  - int table[10][12];
- Access
  - ary2d[i][j]

Two-Dimensional Arrays

- Can define one array for multiple sets of data
- Like a table in a spreadsheet
- Use two size declarators in definition:

```
const int ROWS = 4, COLS = 3;
int exams[ROWS][COLS];
```

- First declarator is number of rows; second is number of columns

Two-Dimensional Array Representation

```
const int ROWS = 4, COLS = 3;  int
exams[ROWS][COLS];
```

columns

rows

exams[0][0]	exams[0][1]	exams[0][2]
exams[1][0]	exams[1][1]	exams[1][2]
exams[2][0]	exams[2][1]	exams[2][2]
exams[3][0]	exams[3][1]	exams[3][2]

- Use two subscripts to access element:

```
exams[2][2] = 86;
```

2D Array Initialization

- Two-dimensional arrays are initialized row-by-row:

```
const int ROWS = 2, COLS = 2;
int exams[ROWS][COLS] = { {84, 78}, {92, 97} };
```

84	78
92	97

Summing All the Elements in a Two-Dimensional Array

- Given the following definitions:

```
const int NUM_ROWS = 5; // Number of rows
const int NUM_COLS = 5; // Number of columns
int total = 0;          // Accumulator
int numbers[NUM_ROWS][NUM_COLS] =
   {{2, 7, 9, 6, 4},
    {6, 1, 8, 9, 4},
    {4, 3, 7, 2, 9},
    {9, 9, 0, 3, 1},
    {6, 2, 7, 4, 1}};
```

2D-Array.cpp

Summing All the Elements in a Two-Dimensional Array

```
// Sum the array elements.
for (int row = 0; row < NUM_ROWS; row++)
{
   for (int col = 0; col < NUM_COLS; col++)
      total += numbers[row][col];
}

// Display the sum.
cout << "The total is " << total << endl;
```

Summing the Rows of a Two-Dimensional Array

- Given the following definitions:

```
const int NUM_STUDENTS = 3;
const int NUM_SCORES = 5;
double total;   // Accumulator
double average; // To hold average scores
double scores[NUM_STUDENTS][NUM_SCORES] =
     {{88, 97, 79, 86, 94},
      {86, 91, 78, 79, 84},
      {82, 73, 77, 82, 89}};
```

Summing the Rows of a Two-Dimensional Array

```
// Get each student's average score.
for (int row = 0; row < NUM_STUDENTS; row++)
{
   // Set the accumulator.
   total = 0;
   // Sum a row.
   for (int col = 0; col < NUM_SCORES; col++)
      total += scores[row][col];
   // Get the average
   average = total / NUM_SCORES;
   // Display the average.
   cout << "Score average for student "
        << (row + 1) << " is " << average <<endl;
}
```

Summing the Columns of a Two-Dimensional Array

- Given the following definitions:

```
const int NUM_STUDENTS = 3;
const int NUM_SCORES = 5;
double total;   // Accumulator
double average; // To hold average scores
double scores[NUM_STUDENTS][NUM_SCORES] =
     {{88, 97, 79, 86, 94},
      {86, 91, 78, 79, 84},
      {82, 73, 77, 82, 89}};
```

Summing the Columns of a Two-Dimensional Array

```
// Get the class average for each score.
for (int col = 0; col < NUM_SCORES; col++)
{
   // Reset the accumulator.
   total = 0;
   // Sum a column
   for (int row = 0; row < NUM_STUDENTS; row++)
      total += scores[row][col];
   // Get the average
   average = total / NUM_STUDENTS;
   // Display the class average.
   cout << "Class average for test " << (col + 1)
        << " is " << average << endl;
}
```

Arrays with Three or More Dimensions

- Can define arrays with any number of dimensions:

```
short rectSolid[2][3][5];
double timeGrid[3][4][3][4];
```