CSCI 1061U
Programming Workshop 2

Arrays
Learning Objectives

• Introduction to Arrays
  • Declaring and referencing arrays
  • For-loops and arrays
  • Arrays in memory

• Arrays in Functions
  • Arrays as function arguments, return values

• Programming with Arrays
  • Partially Filled Arrays, searching, sorting

• Multidimensional Arrays
Introduction to Arrays

• Array is a collection of data of same type
  • An array of integers aggregates multiple integers
  • An array of doubles aggregates multiple doubles

• Used for lists of like items
  • Test scores, temperatures, names, etc.
  • Avoids declaring multiple simple variables
  • Can manipulate "list" as one entity
Declaring Arrays

• Declare the array (allocates memory)
  ```c
  int score[5];
  ```
  • Declares array of 5 integers named "score"
  • Similar to declaring five variables:
    ```c
    int score[0], score[1], score[2], score[3], score[4]
    ```

• Individual parts called many things:
  • Indexed or subscripted variables
  • "Elements" of the array
  • Value in brackets called index or subscript
    • Numbered from 0 to size - 1
Accessing Arrays

• Access using index/subscript
  • `cout << score[3];`

• Note two uses of brackets:
  • In declaration, specifies SIZE of array
  • Anywhere else, specifies a subscript

• Size, subscript need not be literal
  • `int score[MAX_SCORES];`
  • `score[n+1] = 99;`
    • If n is 2, identical to: `score[3]`
Array Usage

• Powerful storage mechanism

• Can issue command like:
  • "Do this to i\textsuperscript{th} indexed variable" where i is computed by program
  • "Display all elements of array score"
  • "Fill elements of array score from user input"
  • "Find highest value in array score"
  • "Find lowest value in array score"
Array Program Example:

```cpp
#include <iostream>
using namespace std;

int main() {
    int i, score[5], max;
    cout << "Enter 5 scores:\n";
    cin >> score[0];
    max = score[0];
    for (i = 1; i < 5; i++)
    {
        cin >> score[i];
        if (score[i] > max)
            max = score[i];
        // max is the largest of the values score[0],..., score[i].
    }
    cout << "The highest score is " << max << endl;
    cout << "The scores and their\n";
    cout << "differences from the highest are:\n";
    for (i = 0; i < 5; i++)
    {
        cout << score[i] << " off by " << (max - score[i]) << endl;
    }
    return 0;
}
```

Sample output

Enter 5 scores:  
5 9 2 10 6  
The highest score is 10  
The scores and their differences from the highest are:  
5 off by 5  
9 off by 1  
2 off by 8  
10 off by 0  
6 off by 4  

for-loops with Arrays

• Natural counting loop
  • Naturally works well "counting through" elements of an array

• Example:
  ```cpp
  for (idx = 0; idx<5; idx++)
  {
      cout << score[idx] << " off by "
           << max - score[idx] << endl;
  }
  • Loop control variable (idx) counts from 0 – 5
Major Array Pitfall

• Array indexes always start with zero!
• Zero is "first" number to computer scientists
• C++ will "let" you go beyond range
  • Unpredictable results
  • Compiler will not detect these errors!
• Up to programmer to "stay in range"
Major Array Pitfall Example

• Indexes range from 0 to (array_size – 1)
  • Example:
    ```
    double temperature[24];  // 24 is array size
    // Declares array of 24 double values called temperature
    • They are indexed as:
      temperature[0], temperature[1] ... temperature[23]
    • Common mistake:
      temperature[24] = 5;
      • Index 24 is "out of range"!
      • No warning, possibly disastrous results
Defined Constant as Array Size

• Always use defined/named constant for array size

• Example:
  ```
  const int NUMBER_OF_STUDENTS = 5;
  int score[NUMBER_OF_STUDENTS];
  ```

• Improves readability

• Improves versatility

• Improves maintainability
Uses of Defined Constant

• Use everywhere size of array is needed
  • In for-loop for traversal:
    ```
    for (idx = 0; idx < NUMBER_OF_STUDENTS; idx++)
    {
      // Manipulate array
    }
    ```
  • In calculations involving size:
    ```
    lastIndex = (NUMBER_OF_STUDENTS – 1);
    ```
  • When passing array to functions (later)

• If size changes → requires only ONE change in program!
Ranged-Based For Loop

• The C++11 ranged-based for loop makes it easy to iterate over each element in a loop

• Format

```cpp
for (datatype varname : array)
{
    // varname is set to each successive
    // element in the array
}
```

• Example

```cpp
int arr[] = {20, 30, 40, 50};
for (int x : arr)
    cout << x << " ";
cout << endl;
```

Output: 20 30 40 50
Arrays in Memory

• Recall simple variables:
  • Allocated memory in an "address"

• Array declarations allocate memory for entire array

• Sequentially-allocated
  • Means addresses allocated "back-to-back"
  • Allows indexing calculations
    • Simple "addition" from array beginning (index 0)
An Array in Memory

Display 5.2  An Array in Memory

```c
int a[6];
```

Address of `a[0]`

On this computer each indexed variable uses 2 bytes, so `a[3]` begins 2x3 = 6 bytes after the start of `a[0]`.

There is no indexed variable `a[6]`, but if there were one, it would be here.

There is no indexed variable `a[7]`, but if there were one, it would be here.

Some variable named `stuff`
Some variable named `moreStuff`
Initializing Arrays

• As simple variables can be initialized at declaration:
  ```
  int price = 0; // 0 is initial value
  ```

• Arrays can as well:
  ```
  int children[3] = {2, 12, 1};
  ```
  • Equivalent to following:
    ```
    int children[3];
    children[0] = 2;
    children[1] = 12;
    children[2] = 1;
    ```
Auto-Initializing Arrays

• If fewer values than size supplied:
  • Fills from beginning
  • Fills "rest" with zero of array base type

• If array-size is left out
  • Declares array with size required based on number of initialization values
  • Example:
    ```
    int b[] = {5, 12, 11};
    ```
    • Allocates array b to size 3
Arrays in Functions

• As arguments to functions
  • Indexed variables
    • An individual "element" of an array can be function parameter
  • Entire arrays
    • All array elements can be passed as "one entity"

• As return value from function
  • Can be done → chapter 10
Indexed Variables as Arguments

• Indexed variable handled same as simple variable of array base type

• Given this function declaration:

```c
void myFunction(double par1);
```

• And these declarations:

```c
int i;  double n, a[10];
```

• Can make these function calls:

```c
myFunction(i);  // i is converted to double
myFunction(a[3]);  // a[3] is double
myFunction(n);  // n is double
```
Subtlety of Indexing

• Consider:
  
  ```
  myFunction(a[i]);
  ```

  • Value of i is determined first
    • It determines which indexed variable is sent
  • myFunction(a[i*5]);
  • Perfectly legal, from compiler’s view
  • Programmer responsible for staying "in-bounds" of array
Entire Arrays as Arguments

• Formal parameter can be entire array
  • Argument then passed in function call is array name
  • Called "array parameter"

• Send size of array as well
  • Typically done as second parameter
  • Simple int type formal parameter
Entire Array as Argument Example

```cpp
#include <iostream>
using namespace std;

void fillUp(int a[], int size);
//Precondition: size is the declared size of the array a.
//The user will type in size integers.
//Postcondition: The array a is filled with size integers
//from the keyboard.

int main()
{
    int a[5], b[10];
    fillUp(a, 5);
    fillUp(b, 10);
    return 0;
}

void fillUp(int a[], int size)
{
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    cout << "The last array index used is " << (size - 1) << endl;
}
```

Slide Credit: Copyright © 2016 Pearson Inc. All rights reserved.
Entire Array as Argument Example

• Given previous example:

• In some main() function definition, consider this calls:

```c
int score[5], numberOfScores = 5;
fillup(score, numberOfScores);
```

  • 1st argument is entire array
  • 2nd argument is integer value

• Note no brackets in array argument!
Array as Argument: How?

• What’s really passed?

• Think of array as 3 "pieces"
  • Address of first indexed variable (arrName[0])
  • Array base type
  • Size of array

• Only 1st piece is passed!
  • Just the beginning address of array
  • Very similar to "pass-by-reference"
Array Parameters

• May seem strange
  • No brackets in array argument
  • Must send size separately

• One nice property:
  • Can use SAME function to fill any size array!
  • Exemplifies "re-use" properties of functions
  • Example:
    ```
    int score[5], time[10];
    fillUp(score, 5);
    fillUp(time, 10);
    ```
The const Parameter Modifier

• Recall: array parameter actually passes address of 1st element
  • Similar to pass-by-reference

• Function can then modify array!
  • Often desirable, sometimes not!

• Protect array contents from modification
  • Use "const" modifier before array parameter
    • Called "constant array parameter"
    • Tells compiler to "not allow" modifications
Functions that Return an Array

• Functions cannot return arrays same way simple types are returned
• Requires use of a "pointer"
• Will be discussed in chapter 10...
Programming with Arrays

• Plenty of uses
  • Partially-filled arrays
    • Must be declared some "max size"
  • Sorting
  • Searching
Partially-filled Arrays

• Difficult to know exact array size needed

• Must declare to be largest possible size
  • Must then keep "track" of valid data in array
  • Additional "tracking" variable needed
    • int numberUsed;
    • Tracks current number of elements in array
**Partially-filled Arrays Example:**

**Display 5.5**  Partially Filled Array (1 of 5)

```cpp
1 // Shows the difference between each of a list of golf scores and their average.
2 #include <iostream>
3 using namespace std;
4 const int MAX_NUMBER_SCORES = 10;

5 void fillArray(int a[], int size, int& numberUsed);
6 // Precondition: size is the declared size of the array a.
7 // Postcondition: numberUsed is the number of values stored in a.
8 // a[0] through a[numberUsed-1] have been filled with
9 // nonnegative integers read from the keyboard.

10 double computeAverage(const int a[], int numberUsed);
11 // Precondition: a[0] through a[numberUsed-1] have values; numberUsed > 0.
12 // Returns the average of numbers a[0] through a[numberUsed-1].

13 void showDifference(const int a[], int numberUsed);
14 // Precondition: The first numberUsed indexed variables of a have values.
15 // Postcondition: Gives screen output showing how much each of the first
16 // numberUsed elements of the array a differs from their average.

(continued)
```
Display 5.5  Partially Filled Array

```cpp
int main() {
    int score[MAX_NUMBER_SCORES], numberUsed;

    cout << "This program reads golf scores and shows\n" << "how much each differs from the average.\n";

    cout << "Enter golf scores:\n";
    fillArray(score, MAX_NUMBER_SCORES, numberUsed);
    showDifference(score, numberUsed);

    return 0;
}
```
Partially-filled Arrays Example:
Display 5.5 Partially Filled Array (3 of 5)

```cpp
27  void fillArray(int a[], int size, int& numberUsed)
28  {
29      cout << "Enter up to " << size << " nonnegative whole numbers.\n"
30         << "Mark the end of the list with a negative number.\n"
31      int next, index = 0;
32      cin >> next;
33      while ((next >= 0) && (index < size))
34          {
35              a[index] = next;
36              index++;
37              cin >> next;
38          }
39      numberUsed = index;
40  }
```
Partially-filled Arrays Example:

Display 5.5 Partially Filled Array (4 of 5)

```cpp
41   double computeAverage(const int a[], int numberUsed)
42   {
43       double total = 0;
44       for (int index = 0; index < numberUsed; index++)
45           total = total + a[index];
46       if (numberUsed > 0)
47           {
48               return (total/numberUsed);
49           }
50       else
51           {
52               cout << "ERROR: number of elements is 0 in computeAverage.\n"
53                   << "computeAverage returns 0.\n"
54               return 0;
55           }
56   }
```
Display 5.5  Partially Filled Array

```c
void showDifference(const int a[], int numberUsed)
{
    double average = computeAverage(a, numberUsed);
    cout << "Average of the " << numberUsed << " scores = " << average << endl;
    cout << "The scores are:
";
    for (int index = 0; index < numberUsed; index++)
        cout << a[index] << " differs from average by " << (a[index] - average) << endl;
}
```

**SAMPLE DIALOGUE**

This program reads golf scores and shows how much each differs from the average.
Enter golf scores:
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.

```
69 74 68 -1
```

Average of the 3 scores = 70.3333
The scores are:
69 differs from average by -1.33333
74 differs from average by 3.66667
68 differs from average by -2.33333
Global Constants vs. Parameters

• Constants typically made "global"
  • Declared above main()

• Functions then have scope to array size constant
  • No need to send as parameter then?
    • Technically yes
  • Why should we anyway?
    • Function definition might be in separate file
    • Function might be used by other programs!
Searching an Array

• Very typical use of arrays
• Display 5.6 next slide
Display 5.6
Searching an Array (1 of 4)

Display 5.6  Searching an Array

1  //Searches a partially filled array of nonnegative integers.
2  #include <iostream>
3  using namespace std;
4  const int DECLARED_SIZE = 20;
5  
6  void fillArray(int a[], int size, int& numberUsed);
7  //Precondition: size is the declared size of the array a.
8  //Postcondition: numberUsed is the number of values stored in a.
9  //a[0] through a[numberUsed-1] have been filled with
10  //nonnegative integers read from the keyboard.
11  
12  int search(const int a[], int numberUsed, int target);
13  //Precondition: numberUsed is <= the declared size of a.
14  //Also, a[0] through a[numberUsed-1] have values.
15  //Returns the first index such that a[index] == target,
16  //provided there is such an index; otherwise, returns -1.
Display 5.6
Searching an Array (2 of 4)

```cpp
15 int main( )
16 {
17     int arr[DECLARED_SIZE], listSize, target;
18     fillArray(arr, DECLARED_SIZE, listSize);
19     char ans;
20     int result;
21     do
22     {
23         cout << "Enter a number to search for: ";
24         cin >> target;
25         result = search(arr, listSize, target);
26         if (result == -1)
27             cout << target << " is not on the list.\n";
28         else
29             cout << target << " is stored in array position "
30                 << result << endl
31                 << "(Remember: The first position is 0.)\n";
```
Display 5.6
Searching an Array (3 of 4)

```cpp
Display 5.6  Searching an Array

32    cout << "Search again?(y/n followed by Return): ";
33    cin >> ans;
34    } while ((ans != 'n') && (ans != 'N'));
35    cout << "End of program.\n";
36    return 0;
37
38    void fillArray(int a[], int size, int& numberUsed)
39    {<The rest of the definition of fillArray is given in Display 5.5>
40    int search(const int a[], int numberUsed, int target)
41    {
        int index = 0;
42        bool found = false;
43        while (!found && (index < numberUsed))
44        if (target == a[index])
45            found = true;
46        else
47            index++;
```
Display 5.6
Searching an Array (4 of 4)

```java
49    if (found)
50        return index;
51    else
52        return -1;
53    }
```

**Sample Dialogue**

Enter up to 20 nonnegative whole numbers. 
Mark the end of the list with a negative number.

10 20 30 40 50 60 70 80 -1
Enter a number to search for: 10
10 is stored in array position 0
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 40
40 is stored in array position 3
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 42
42 is not on the list.
Search again?(y/n followed by Return): n
End of program.
Sorting an Array:

**Display 5.7 Selection Short**

- **Selection Sort Algorithm**

---

**Display 5.7 Selection Sort**

\[
\begin{array}{cccccccccc}
\text{a[0]} & \text{a[1]} & \text{a[2]} & \text{a[3]} & \text{a[4]} & \text{a[5]} & \text{a[6]} & \text{a[7]} & \text{a[8]} & \text{a[9]} \\
8 & 6 & 10 & 2 & 16 & 4 & 18 & 14 & 12 & 20 \\
\end{array}
\]
Sorting an Array Example:

Display 5.8 Sorting an Array (1 of 4)

```c++
1 //Tests the procedure sort.
2 #include <iostream>
3 using namespace std;

4 void fillArray(int a[], int size, int& numberUsed);
5 //Precondition: size is the declared size of the array a.
6 //Postcondition: numberUsed is the number of values stored in a.
7 //a[0] through a[numberUsed - 1] have been filled with
8 //nonnegative integers read from the keyboard.
9 void sort(int a[], int numberUsed);
10 //Precondition: numberUsed <= declared size of the array a.
```

(continued)
Sorting an Array Example:
Display 5.8  Sorting an Array (2 of 4)

```cpp
Display 5.8  Sorting an Array

11  //The array elements a[0] through a[numberUsed - 1] have values.
12  //Postcondition: The values of a[0] through a[numberUsed - 1] have
13  //been rearranged so that a[0] <= a[1] <= ... <= a[numberUsed - 1].
14  void swapValues(int& v1, int& v2);
15  //Interchanges the values of v1 and v2.
16  int indexOfSmallest(const int a[], int startIndex, int numberUsed);
17  //Precondition: 0 <= startIndex < numberUsed. Reference array elements
18  //have values. Returns the index i such that a[i] is the smallest of the
19  //values a[startIndex], a[startIndex + 1], ..., a[numberUsed - 1].
20  int main()  
21  { 
22    cout << "This program sorts numbers from lowest to highest.\n";
23    int sampleArray[10], numberUsed;
24    fillArray(sampleArray, 10, numberUsed);
25    sort(sampleArray, numberUsed);
26    cout << "In sorted order the numbers are:\n";
27    for (int index = 0; index < numberUsed; index++)
28      cout << sampleArray[index] << " ";
29    cout << endl;
30    return 0;
31  }
```
Sorting an Array Example:

Display 5.8 Sorting an Array (3 of 4)

```c
void fillArray(int a[], int size, int &numberUsed)
    <The rest of the definition of fillArray is given in Display 5.5.>

void sort(int a[], int numberUsed)
{
    int indexOfNextSmallest;
    for (int index = 0; index < numberUsed - 1; index++)
    { //Place the correct value in a[index]:
        indexOfNextSmallest =
            indexOfSmallest(a, index, numberUsed);
        swapValues(a[index], a[indexOfNextSmallest]);
        //a[0] <= a[1] <=...<= a[index] are the smallest of the original array
        //elements. The rest of the elements are in the remaining positions.
    }

void swapValues(int& v1, int& v2)
{
    int temp;
    temp = v1;
    v1 = v2;
}
```
Sorting an Array Example:

Display 5.8 Sorting an Array (4 of 4)

```c
Display 5.8  Sorting an Array

51        v2 = temp;
52    }
53
54 int indexOfSmallest(const int a[], int startIndex, int numberUsed)
55 {   int min = a[startIndex],
56     indexOfMin = startIndex;
57     for (int index = startIndex + 1; index < numberUsed; index++)
58       if (a[index] < min)
59         {   min = a[index];
60             indexOfMin = index;
61         //min is the smallest of a[startIndex] through a[index]
62     }
63
64     return indexOfMin;
65 } 
```

**SAMPLE DIALOGUE**

This program sorts numbers from lowest to highest. Enter up to 10 nonnegative whole numbers. Mark the end of the list with a negative number.

80 30 50 70 60 90 20 30 40 –1

In sorted order the numbers are:

20 30 30 40 50 60 70 80 90
Multidimensional Arrays

• Arrays with more than one index
  • char page[30][100];
    • Two indexes: An "array of arrays"
    • Visualize as:
      page[0][0], page[0][1], ..., page[0][99]
      page[1][0], page[1][1], ..., page[1][99]
      ...
      page[29][0], page[29][1], ..., page[29][99]

• C++ allows any number of indexes
  • Typically no more than two
Multidimensional Array Parameters

• Similar to one-dimensional array
  • 1st dimension size not given
  • Provided as second parameter
  • 2nd dimension size IS given

• Example:

```cpp
void DisplayPage(const char p[][100], int sizeDimension1) {
    for (int index1=0; index1<sizeDimension1; index1++) {
        for (int index2=0; index2 < 100; index2++)
            cout << p[index1][index2];
        cout << endl;
    }
}
```
Summary 1

- Array is collection of "same type" data
- Indexed variables of array used just like any other simple variables
- for-loop "natural" way to traverse arrays
- Programmer responsible for staying "in bounds" of array
- Array parameter is "new" kind
  - Similar to call-by-reference
Summary 2

• Array elements stored sequentially
  • "Contiguous" portion of memory
  • Only address of 1st element is passed to functions

• Partially-filled arrays → more tracking

• Constant array parameters
  • Prevent modification of array contents

• Multidimensional arrays
  • Create "array of arrays"