Arrays Hold Multiple Values

• When you have lots of data to store, naming variables is a problem:
  ```java
double length1, length2, length3;
```
• What if we have 1000 variables?
  ```java
  We can name them individually:
  double length1, length2, ..., length999;
  ```
  What if we have 1000 variables?
  ```java
  double length1, length2, length3;
  ```
• Naming variables is a problem.
  ```java
  When you have lots of data to store, you can use an array.
  ```

Arrays Hold Multiple Values

• An array allows you to store and work with multiple values of the same data type.
• Values are stored in adjacent memory locations.
• Declared using the syntax:
  ```java
type name[quantity];
```
• Quantity must be a constant int.
• Good style is to not use numeric constants in code, so we would use the second approach.
  ```java
  const int size = 50;
  int altitude[size];
  ```
• The definition: 
  ```java
  altitude[size];
  ```
• Allocate the following memory:
  ```java
  altitudes[5];
  ```

Array Terminology

In the definition
  ```java
  int tests[5];
  ```
• Tests is the name of the array.
• 5, in [5], is the size declarator. It shows how many values of the type specified are stored. It specifies the size of the array.
• Good style is to not use numeric constants in code, so we would use the second approach.
  ```java
  tests[size];
  ```
Array Terminology

• The size of an array is:
  – the total number of bytes allocated for it
  – (number of elements) * (number of bytes for each element)

• Examples:
  int tests[5] is an array of 20 bytes,
  assuming 4 bytes for an int
  long double measures[10] is an array of
  80 bytes, assuming 8 bytes for a long double

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.

7.2 Accessing Array Elements

• Now we have a group of variables. How do we access each one?
• We use the name of the array and a subscript.
• The subscript is a number indicating which element in the array you referring to.

Accessing Array Elements

• Each element in an array is assigned a unique subscript.
• Subscripts start at 0

Accessing Array Elements

• The last element’s subscript is $n-1$ where $n$ is the number of elements in the array.

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
Example

Here are the contents of the `hours` array, with the values entered by the user in the example output:

Using a Loop to Step Through an Array

Example - The following code defines an array, `numbers`, and assigns 99 to each element:

```cpp
const int ARRAY_SIZE = 5;
int numbers[ARRAY_SIZE];
for (int count = 0; count < ARRAY_SIZE; count++)
    numbers[count] = 99;
```

A Closer Look At the Loop

- Default: Local array - all elements initialized to 0 by default
- By default: Global array - all elements initialized to 0 by default

Default Initialization

- Can access element with a constant or literal subscript:
  ```cpp
  cout << tests[3] << endl;
  ```
- Can use integer expression as subscript:
  ```cpp
  int i = 5;
  cout << tests[i] << endl;
  ```

Literal Subscript:
Can access element with a constant or literal subscript

Program Continues...

Example
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.

• The compiler does not keep track of whether you are in bounds or out of bounds.

• .NET has a Run-Time Check that tries to let you know when this has happened. Many environments don’t check.

7.4 Array Initialization

• Arrays can be initialized with an initialization list:

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

7.5 Off-By-One Errors

• An off-by-one error happens when you use array subscripts that are off by one.

• Doing so can corrupt other memory locations, crash your program, or lock up your computer.

• Be careful not to use invalid subscripts.

What the Code Does

Code From Program 7-5

```cpp
for (int count = 0; count < SIZE; count++)
    numbers[count] = 0;
```
7.5 Processing Array Contents

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

Partial Array Initialization

- If array is initialized with fewer initial values than the size declarator, the remaining elements will be set to 0.

Example

- This creates a char array of length 13.
  
- char address[]="Hello World!"

Implicit Array Sizing

- Can determine array size by the size of the initialization list:

  ```
  int quizzes[][4] = {{12,17,15,11}};
  ```

- Must use either array size declarator or initialization list:

  ```
  int quizzes[4][4];
  ```

Initializing With a String

- Character array can be initialized by enclosing string in " ":

  ```
  const int SIZE = 6;
  char fName[SIZE] = "Henry";
  ```

  - Must leave room for \0 at end of array
  
  ```
  const int SIZE = 6;
  char fName[SIZE] = "Henry\0";
  ```

  - Character array can be initialized by "enduring string in " " :

  ```
  char fName[SIZE] = { 'H', 'e', 'n', 'r', 'y', '\0' };
  ```

- If initializing character-by-character, must add in \0 explicitly:

  ```
  char address[SIZE] = "Hello World!\0"
  ```

- Must use either array size declarator or initialization list:

  ```
  int quizzes[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
  ```

7.5 Processing Array Contents

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

Example

- This creates a char array of length 13.
  
- char address[]="Hello World!"

Implicit Array Sizing

- Can determine array size by the size of the initialization list:

  ```
  int quizzes[][4] = {{12,17,15,11}};
  ```

- Must use either array size declarator or initialization list:

  ```
  int quizzes[4][4];
  ```

Initializing With a String

- Character array can be initialized by enclosing string in " ":

  ```
  const int SIZE = 6;
  char fName[SIZE] = "Henry";
  ```

  - Must leave room for \0 at end of array
  
  ```
  const int SIZE = 6;
  char fName[SIZE] = "Henry\0";
  ```

  - Character array can be initialized by "enduring string in " " :

  ```
  char fName[SIZE] = { 'H', 'e', 'n', 'r', 'y', '\0' };
  ```

- If initializing character-by-character, must add in \0 explicitly:

  ```
  char fName[SIZE] = { 'H', 'e', 'n', 'r', 'y', '\0' };
  ```

- This creates a char array of length 13.

Privacy Policy

- This creates a char array of length 13.

Privacy Policy
Processing Array Contents

Finding the Highest Value in an Array

Finding the Lowest Value in an Array

Array Assignment

Printing the Contents of an Array

Summing and Averaging Array Elements

- Use a simple loop to add together array elements.
- Once summed, can compute average:
  
  ```
  average = sum / SIZE;
  ```

- Use this element-by-element:

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

Finding the Highest Value in the Numbers Array

Finding the Lowest Value in the Numbers Array

Array Assignment

Processing Array Contents

Finding the Highest Value in an Array

Finding the Lowest Value in an Array

Array Assignment

Printing the Contents of an Array

Summing and Averaging Array Elements

- Use a simple loop to add together array elements.
- Once summed, can compute average:

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

- Use this element-by-element:

```
for (i = 0; i < ARRAY_SIZE; i++)
    cout << tests[i] << endl;
```
Partially-Filled Arrays

• It is common to only use a portion of an array. This is allowed. You simply need to keep track of how many elements are currently being used.

Then instead of looping from zero to one less than the size of the array you loop from zero to one less than how many elements are currently being used.

```cpp
const int SIZE = 100;
in array[SIZE];
for (int i=0;i<50;i++)
cin>>array[i];
```

Comparing Arrays

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

```cpp
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";
```

We can re-write it like this:

```cpp
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
for (int i=0;i<SIZE;i++)
{
    if (firstArray[i] != secondArray[i])
        {arraysEqual = false;
        break;}
}
if (arraysEqual)
    cout << "The arrays are equal.\n";
else
    cout << "The arrays are not equal.\n";
```

Using Parallel Arrays

• We often want to keep a variety of data about one thing. For example, you may want to keep student information about a student:
  – Student ID
  – Student age
  – Student GPA
  – Etc.

• Problem: Arrays must contain data of the same type.

• Solution: Use multiple arrays.

• Parallel arrays: two or more arrays that contain related data. A subscript is used to relate arrays.

```cpp
element = array[student ID] // course grade
student ID // course average
course average // course grade
```

Example 1

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

Arrays may be of different types. Elements at the same subscript are related. A subscript is used to relate arrays. Parallel arrays: two or more arrays that contain related data.

7.6 Using Parallel Arrays

Comparing Arrays

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

```cpp
\text{compare two arrays, you must compare element-by-element:}
\text{compare two arrays, you must compare element-by-element:}
\text{compare two arrays, you must compare element-by-element:}
```
7.7 Arrays as Function Argument

Arrays as Function Argument

- Arrays as Function Argument
- To pass an array to a function, just use the array name:

```
void showScores(int tests[]); // function prototype
```

- To define a function that takes an array as an argument, use empty brackets for the array argument:

```
void showScores(int tests[]); // function prototype
```

- When passing an array to a function, it is common to pass array size so that function knows how many elements to process:

```
void showScores(int tests[], int size); // function prototype
```

- Array size must also be reflected in prototype.

Example 2

```c
void displayValue(int); // prototype
void displayValue(int age[10]); // call
void displayValue(int) // definition
{
    cout << "The value is " << value << endl;
}
```

Example 2

```c
void displayValue(int[]); // function prototype
void displayValue(int tests[]); // function header
```

Example 2

```c
void displayValue(int[]); // function prototype
void displayValue(int tests[], int size) // function header
```
7.8 Two-Dimensional Arrays

A two-dimensional array is like several identical arrays put together. It is useful for storing multiple sets of data.

- Like a table in a spreadsheet
- Multiple sets of data
- A two-dimensional array is like several identical arrays put together. It is useful for storing
- The number of columns is number of rows. Second is
- The first declarator is number of rows; second is
- Use two size declarators in definition:
- Like a table in a spreadsheet
- Use two subscripts to access element:
  - exams[2][2] = 86;
  - The row index is first, it starts at 0
  - The column index is second, it starts at 0
- A two-dimensional array is like several identical
- The function has complete access to the elements in the array
- This is called Pass-by-pointer in the lab manual
- Seems like Pass-by-reference
- Actually the address is passed by value
- The function can actually make changes to the array
- The contents of the array
- The function actually makes changes to the array
- Actually being passed is
- The address of where the array begins is
- Two-Dimensional Arrays
- Example

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

// Example
```
Two-Dimensional Arrays

How do we loop through a 2-D array?

Example

```java
int number = 1;
for (int row=0; row<ROWS; row++)
    test[row][4] = number++;

Example

<table>
<thead>
<tr>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

Two-Dimensional Arrays

Example

```java
int number = 1;
for (int col=0; col<COLS; col++)
    test[2][col] = number++;

Example

<table>
<thead>
<tr>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

Two-Dimensional Arrays

Example

```java
int number = 1;
for (int row=0; row<ROWS; row++)
    for (int col=0; col<COLS; col++)
        test[row][col] = number++;

Example

<table>
<thead>
<tr>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>
Two-Dimensional Array Initialization

- Two-dimensional arrays are initialized row-by-row:
  
  ```
  const int ROWS = 2, COLS = 2;
  int exams[ROWS][COLS] = { {84, 78},
                             {92, 97} };  
  ```

- Can omit inner { }, some initial values in a row – array elements without initial values will be set to 0

Example

```
{  
  84  
  78 
}
```

Notice

- Pay attention to which size you are passing

```
void getExams(int exams[][COLS], int cols)  
```
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}};

  // 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}};

  // 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}};

  // Get the row elements sum.
  for (int row = 0; row < NUM_ROWS; row++){
    // For each row.
    for (int col = 0; col < NUM_SCORES; col++)
      total += numbers[row][col];
    // Display the row.
    cout << "Row " << row + 1 << " sum is " << total << endl;
    total = 0;
  }

Example
Summing the Columns of a Two-Dimensional Array

```cpp
// 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;
}
```

### Example

```
cout << "can use [ ] to access elements definition needed - no need to determine size at
automatically adds space as more is
Vector<short, scores>:
Can hold values of any type:
Chapter 16
Template Library (covered more in
A data type defined in the Standard

Program shown (continued)
```

### 7.12 Introduction to the STL Vector

- Can define arrays with any number of
- 7.10 n-d arrays
  - Can define arrays with any number of
  ```cpp
  short rectSolid[2][3][5];
  double timeGrid[3][4][3][4];
  ```
  - When used as parameter, specify all but
    1st dimension in prototype, reading:
    ```cpp
    void getRectSolid(short [[3][5]]);
    double timeGrid[3][4][3][4];
    ```
  - Dimensions:
    ```cpp
    short rectSolid[2][3][5];
    ```

### 7.9 Array of C-Strings

- Use a two-dimensional array of characters as an
  array of strings:
  ```cpp
  const int NAMES = 3, SIZE = 10;
  char students[NAMES][SIZE] = {
      { "Ann", "Bill", "Cindy" },
      { "John", "Mike", "Sue" },
      { "Jane", "Alice", "Bob" }
  };
  ```
  - Each row contains one string
  - Can use row subscript to reference the string in
    a particular row:
    ```cpp
    cout << students[i];
    ```

### Example

```
int main()
{
    char students[3][10] = {
        { "Ann", "Bill", "Cindy" },
        { "John", "Mike", "Sue" },
        { "Jane", "Alice", "Bob" }
    };

    // Display the class average.
    for (col = 0; col < NUM_SCORES; col++)
    {
        // Display the class average.
        for (int row = 0; row < NUM_STUDENTS; row++)
            total += scores[row][col];
        // Display the class average.
        average = total / NUM_STUDENTS;
        // Display the class average.
        cout << "Class average for test " << (col + 1)
            << " is " << average << endl;
    }
}
```
Declaring Vectors

- You must #include<vector>
- Declare a vector to hold int element:
  ```cpp
  vector<int> scores;
  ```
- Declare a vector with initial size 30:
  ```cpp
  vector<int> scores(30);
  ```
- Declare a vector and initialize all elements to 0:
  ```cpp
  vector<int> scores(30, 0);
  ```
- Declare a vector initialized to size and contents of another vector:
  ```cpp
  vector<int> finals(scores);
  ```

Adding Elements to a Vector

- Use push_back member function to add element to a full array or to an array that had no defined size:
  ```cpp
  scores.push_back(75);
  ```
- Use size member function to determine size of a vector:
  ```cpp
  int howbig = scores.size();
  ```

Removing Vector Elements

- Use pop_back member function to remove last element from vector:
  ```cpp
  scores.pop_back();
  ```
- To remove all contents of vector, use clear member function:
  ```cpp
  scores.clear();
  ```
- To determine if vector is empty, use empty member function:
  ```cpp
  while (!scores.empty()) ...
  ```

Other Useful Member Functions

- Exchange the contents of two vectors:
  ```cpp
  vec1.swap(vec2);
  ```
- Add elements to a vector, optionally initializes them:
  ```cpp
  vec1.resize(5,0);
  ```
- Reverse the order of the elements in a vector:
  ```cpp
  vec1.reverse();
  ```
- Returns the maximum number of elements a vector can store:
  ```cpp
  maxelts = vec1.capacity();
  ```
- Returns the value of the element at position `elt` in the vector:
  ```cpp
  cout << vec1.at(i);
  ```

Declaring Vectors

- Use clear, push_back to add elements to a vector:
  ```cpp
  scores.push_back(75);
  ```
- Declare a vector initialized to size and contains elements of another vector:
  ```cpp
  vector<int> scores(30, 0);
  ```
- Declare a vector with initial size and contains elements of another vector:
  ```cpp
  vector<int> scores(30);
  ```
- Declare a vector to hold the element that you must include:
  ```cpp
  vector<int> scores;
  ```
- You must include<vector>