

# Arrays:

*A sequential collection of data storage locations, each of which holds the same data-type*

# Contents

- What is an array?
- Array declaration.
- Array Initialization.
- How to access array?
- Range-Based for loop.
- Array assignment.
- Parallel array.
- Passing array to function.
- N-dimensional array.

# What is data structure?

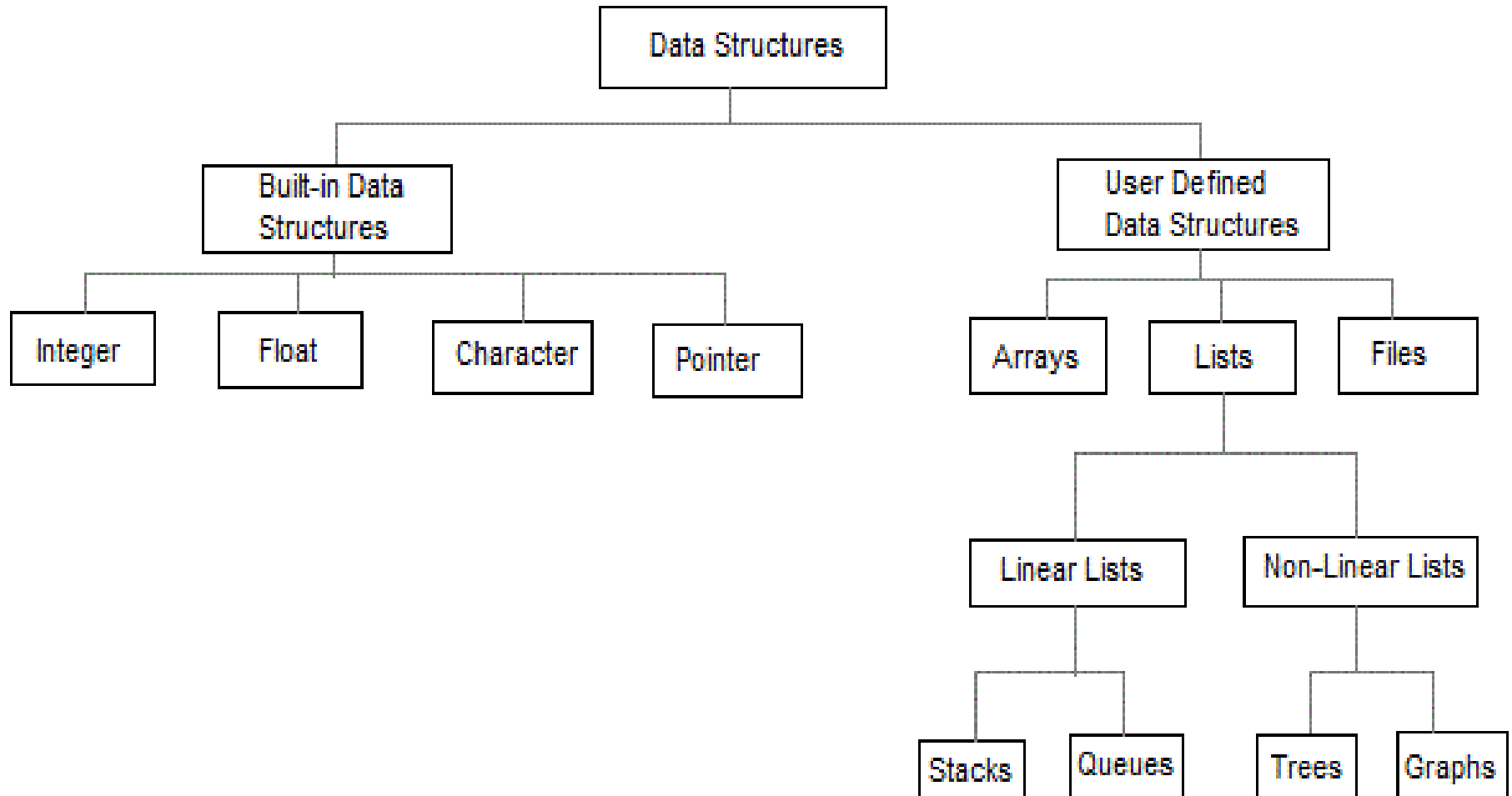
- In computer science, a data structure is a
  - data organization,
  - management, and
  - storage format
- that enables efficient access and modification.
- Precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data

# Examples

- An array
  - is a number of elements in a specific order, Typical implementations allocate contiguous memory words for the elements of arrays. Arrays may be fixed-length or resizable.
- A linked list
  - is a linear collection of data elements of any type, called nodes, where each node has itself a value, and points to the next node in the linked list.
- A record (also called tuple or struct)
  - is an aggregate data structure.

- anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as Primitive Data Structures.
- Some complex Data Structures, which are used to store large and connected data. E.g example of ADT are :
  - Linked List, Tree Graph Stack, Queue etc..

# Introduction to Data Structure



<u>Characteristic</u>	<u>Description</u>
Linear	Data items are arranged in a linear sequence. Example: <b>Array</b>
Non-Linear	Data items are not in sequence. Example: <b>Tree, Graph</b>
Homogeneous	the elements are of same type. Example: <b>Array</b>
Non-Homogeneous	The elements may or may not be of the same type. Example: <b>Structures</b>
Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: <b>Array</b>
Dynamic	Those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: <b>Linked List created using pointers</b>

# 7.1 Arrays Hold Multiple Values

- An array is a sequential collection of data storage locations, each of which holds the same data-type.
- 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.
- The size of an array is (number of elements) \* (size of each element)

# Array Terminology

- The size of an array is:

- the total number of bytes allocated for it
- $(\text{number of elements}) * (\text{number of bytes for each element})$

- Examples:

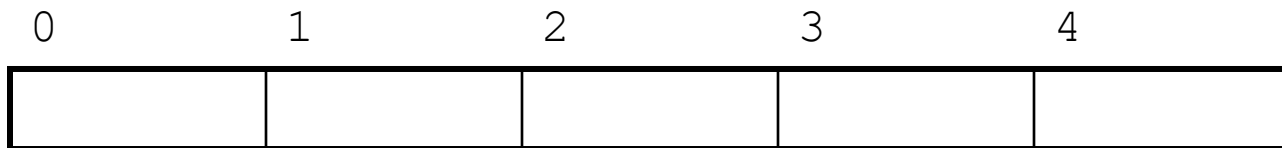
`int tests[5]` is an array of 20 bytes,  
assuming 4 bytes for an `int`

`long double measures[10]` is an array of  
80 bytes, assuming 8 bytes for a `long double`

## 7.2 Accessing Array Elements

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

subscripts:



# 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
```

# Accessing Array Elements in Program 7-1

## Program 7-1

```
1 // This program asks for the number of hours worked
2 // by six employees. It stores the values in an array.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     const int NUM_EMPLOYEES = 6;
9     int hours[NUM_EMPLOYEES];
10
11     // Get the hours worked by each employee.
12     cout << "Enter the hours worked by "
13          << NUM_EMPLOYEES << " employees: ";
14     cin >> hours[0];
15     cin >> hours[1];
16     cin >> hours[2];
17     cin >> hours[3];
18     cin >> hours[4];
19     cin >> hours[5];
20
```

*(Program Continues)*

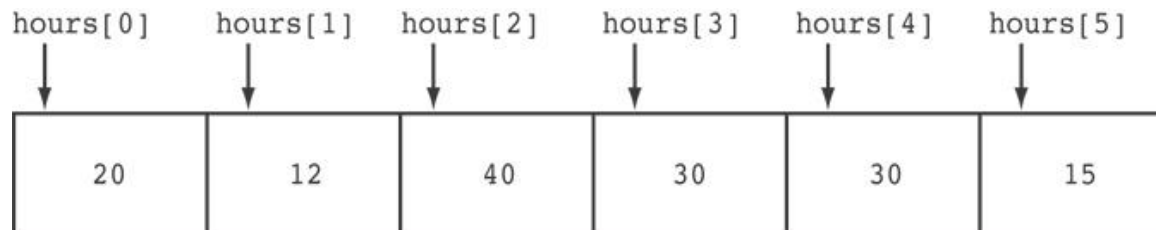
# Accessing Array Elements in Program 7-1

```
21    // Display the values in the array.
22    cout << "The hours you entered are:";
23    cout << " " << hours[0];
24    cout << " " << hours[1];
25    cout << " " << hours[2];
26    cout << " " << hours[3];
27    cout << " " << hours[4];
28    cout << " " << hours[5] << endl;
29    return 0;
30 }
```

## Program Output with Example Input Shown in Bold

Enter the hours worked by 6 employees: **20 12 40 30 30 15** [Enter]  
The hours you entered are: 20 12 40 30 30 15

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:

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

# Array Initialization

- Arrays can be initialized with an initialization list:

```
const int SIZE = 5;  
int tests[SIZE] = {79, 82, 91, 77, 84};
```

- The values are stored in the array in the order in which they appear in the list.
- The initialization list cannot exceed the array size.



# Code From Program 7-3

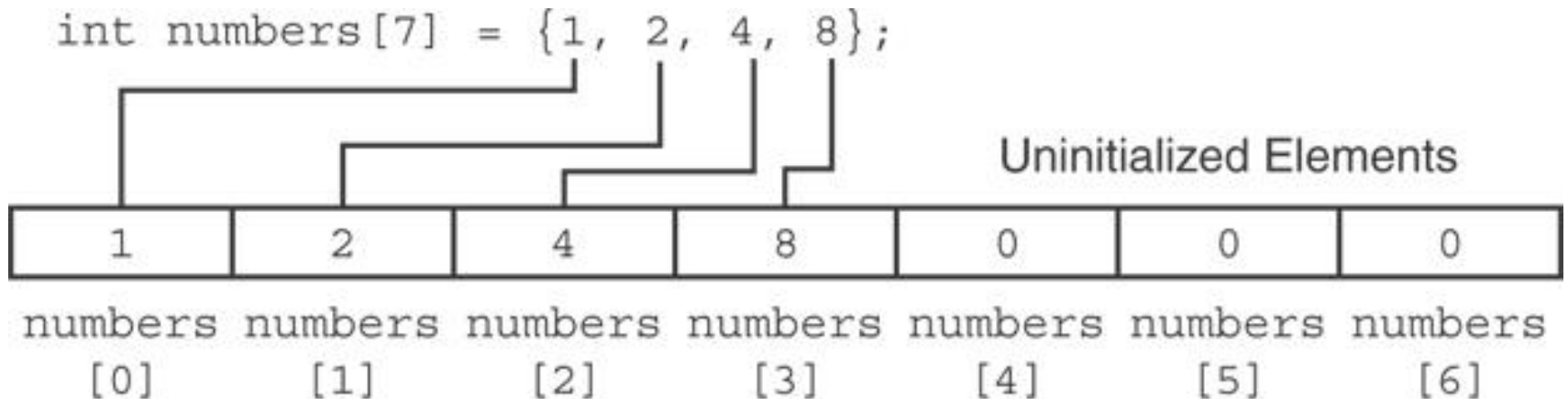
```
7      const int MONTHS = 12;
8      int days[MONTHS] = { 31, 28, 31, 30,
9                          31, 30, 31, 31,
10                         30, 31, 30, 31};
11
12      for (int count = 0; count < MONTHS; count++)
13      {
14          cout << "Month " << (count + 1) << " has ";
15          cout << days[count] << " days.\n";
16      }
```

## Program Output

```
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

# Partial Array Initialization

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



# Implicit Array Sizing

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

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

12	17	15	11
----	----	----	----

- Must use either array size declarator or initialization list at array definition

## 7.3 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.

# Code From Program 7-9

- The following code defines a three-element array, and then writes five values to it!

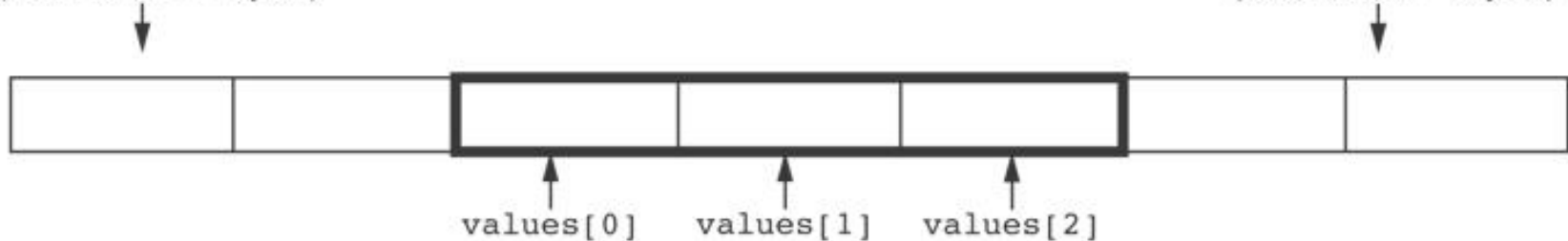
```
9      const int SIZE = 3;    // Constant for the array size
10     int values[SIZE];      // An array of 3 integers
11     int count;             // Loop counter variable
12
13     // Attempt to store five numbers in the 3-element array.
14     cout << "I will store 5 numbers in a 3-element array!\n";
15     for (count = 0; count < 5; count++)
16         values[count] = 100;
```

# What the Code Does

The way the `values` array is set up in memory.  
The outlined area represents the array.

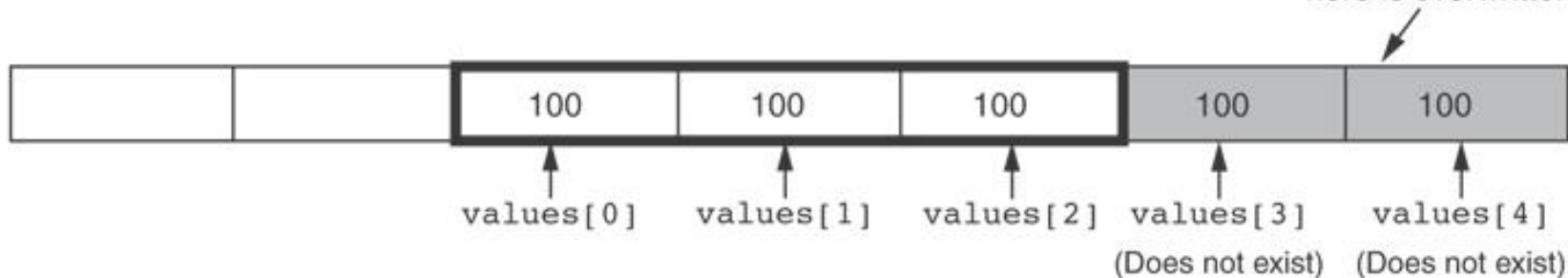
Memory outside the array  
(Each block = 4 bytes)

Memory outside the array  
(Each block = 4 bytes)



How the numbers assigned to the array overflow the array's boundaries.  
The shaded area is the section of memory illegally written to.

Anything previously stored  
here is overwritten.



# 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.

# 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;
```



# 7.4 The Range-Based `for` Loop

- C++ 11 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.

# The Range-Based `for` Loop

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

```
for (dataType rangeVariable : array)  
    statement;
```

- dataType*** is the data type of the range variable.
- rangeVariable*** 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.

# The range-based `for` loop

```
1  // This program demonstrates the range-based for loop.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      // Define an array of integers.
8      int numbers[] = { 10, 20, 30, 40, 50 };
9
10     // Display the values in the array.
11     for (int val : numbers)
12         cout << val << endl;
13
14     return 0;
15 }
```

# Modifying an Array with a Range-Based `for` Loop

- As the range-based `for` loop executes, its range variable contains only a copy of an array element.
- You cannot use a range-based `for` loop to modify the contents of an array unless you declare the range variable as a reference.
- To declare the range variable as a reference variable, simply write an ampersand (&) in front of its name in the loop header.

# Modifying an Array with a Range-Based for Loop

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

// Get values for the array.
for (int &val : numbers)
{
    cout << "Enter an integer value: ";
    cin >> val;
}

// Display the values in the array.
cout << "Here are the values you entered:\n";
for (int val : numbers)
    cout << val << endl;
```

# Modifying an Array with a Range-Based for Loop

You can use the `auto` key word with a reference range variable. For example, the code in lines 12 through 16 in Program 7-12 could have been written like this:

```
for (auto &val : numbers)
{
    cout << "Enter an integer value: ";
    cin >> val;
}
```

# 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.

# 7.5 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 << tests[i] << endl;
```

# Printing the Contents of an Array

- In C++ 11 you can use the range-based `for` loop to display an array's contents, as shown here:

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

# 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++ 11 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 contain 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 contain the lowest value in the `numbers` array.



# 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.

# 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";
```

## 7.6 Using Parallel Arrays

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

# Parallel Array Example

```
const int SIZE = 5;    // Array size
int id[SIZE];          // student ID
double average[SIZE]; // course 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;
}
```

# Parallel Arrays in Program 7-15

## Program 7-15

```
1  // This program uses two parallel arrays: one for hours
2  // worked and one for pay rate.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9      const int NUM_EMPLOYEES = 5;    // Number of employees
10     int hours[NUM_EMPLOYEES];        // Holds hours worked
11     double payRate[NUM_EMPLOYEES];   // Holds pay rates
12
13     // Input the hours worked and the hourly pay rate.
14     cout << "Enter the hours worked by " << NUM_EMPLOYEES
15          << " employees and their\n"
16          << "hourly pay rates.\n";
17     for (int index = 0; index < NUM_EMPLOYEES; index++)
18     {
19         cout << "Hours worked by employee #" << (index+1) << ": ";
20         cin >> hours[index];
21         cout << "Hourly pay rate for employee #" << (index+1) << ": ";
22         cin >> payRate[index];
23     }
24
```

*(Program Continues)*

# Parallel Arrays

```
25 // Display each employee's gross pay.
26 cout << "Here is the gross pay for each employee:\n";
27 cout << fixed << showpoint << setprecision(2);
28 for (int index = 0; index < NUM_EMPLOYEES; index++)
29 {
30     double grossPay = hours[index] * payRate[index];
31     cout << "Employee #" << (index + 1);
32     cout << ": $" << grossPay << endl;
33 }
34 return 0;
35 }
```

## Program Output with Example Input Shown in Bold

Enter the hours worked by 5 employees and their hourly pay rates.

Hours worked by employee #1: **10**

Hourly pay rate for employee #1: **9.75**

Hours worked by employee #2: **15**

Hourly pay rate for employee #2: **8.62**

Hours worked by employee #3: **20**

Hourly pay rate for employee #3: **10.50**

Hours worked by employee #4: **40**

Hourly pay rate for employee #4: **18.75**

Hours worked by employee #5: **40**

Hourly pay rate for employee #5: **15.65**

Here is the gross pay for each employee:

Employee #1: \$97.50

Employee #2: \$129.30

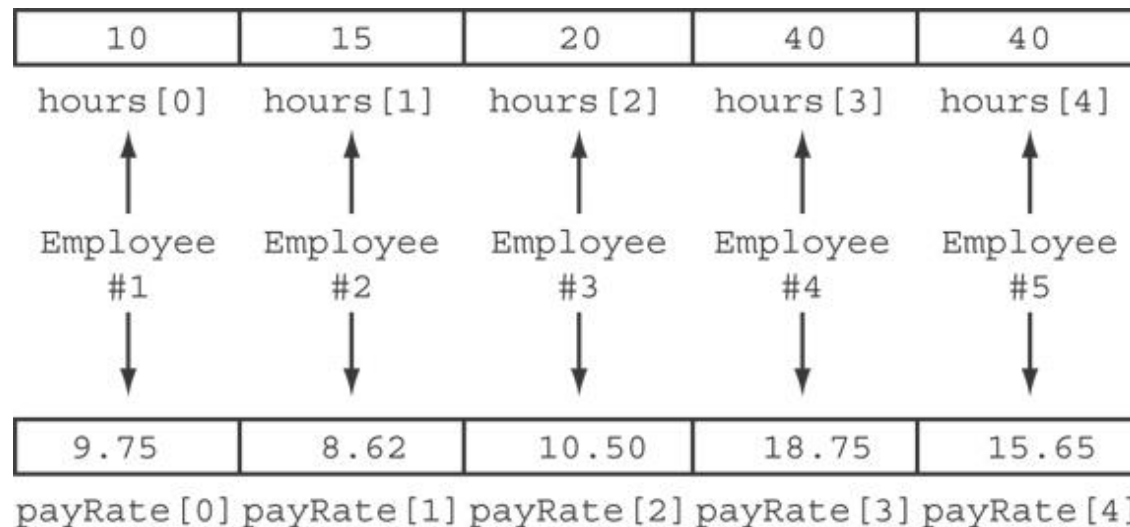
Employee #3: \$210.00

Employee #4: \$750.00

Employee #5: \$626.00

# Parallel Arrays in Program 7-15

The `hours` and `payRate` arrays are related through their subscripts:



# 7.7 Arrays as Function Arguments

- To pass an array to a function, just use the array name:

```
showScores(tests);
```

- To define a function that takes an array parameter, use empty `[]` for array argument:

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

```
// function header  
void showScores(int tests[])
```



# Arrays as Function Arguments

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

```
showScores(tests, ARRAY_SIZE);
```

- Array size must also be reflected in prototype, header:

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

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

# Modifying Arrays in Functions

- Array names in functions are like reference variables – changes made to array in a function are reflected in actual array in calling function
- Need to exercise caution that array is not inadvertently changed by a function

# 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;
```

# A Two-dimensional Array

## Program 7-21

```
1  // This program demonstrates a two-dimensional array.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  int main()
7  {
8      const int NUM_DIVS = 3;           // Number of divisions
9      const int NUM_QTRS = 4;           // Number of quarters
10     double sales[NUM_DIVS][NUM_QTRS]; // Array with 3 rows and 4 columns.
11     double totalSales = 0;             // To hold the total sales.
12     int div, qtr;                      // Loop counters.
13
14     cout << "This program will calculate the total sales of\n";
15     cout << "all the company's divisions.\n";
16     cout << "Enter the following sales information:\n\n";
17
```

*(program continues)*

# A Two-dimensional Array

## **Program 7-21** *(continued)*

```
18     // Nested loops to fill the array with quarterly
19     // sales figures for each division.
20     for (div = 0; div < NUM_DIVS; div++)
21     {
22         for (qtr = 0; qtr < NUM_QTRS; qtr++)
23         {
24             cout << "Division " << (div + 1);
25             cout << ", Quarter " << (qtr + 1) << ": $";
26             cin >> sales[div][qtr];
27         }
28         cout << endl; // Print blank line.
29     }
30
31     // Nested loops used to add all the elements.
32     for (div = 0; div < NUM_DIVS; div++)
33     {
34         for (qtr = 0; qtr < NUM_QTRS; qtr++)
35             totalSales += sales[div][qtr];
36     }
37
38     cout << fixed << showpoint << setprecision(2);
39     cout << "The total sales for the company are: $";
40     cout << totalSales << endl;
41     return 0;
42 }
```

# A Two-dimensional Array in Program 7-21

## Program Output with Example Input Shown in Bold

This program will calculate the total sales of all the company's divisions.

Enter the following sales data:

Division 1, Quarter 1: \$**31569.45** [Enter]

Division 1, Quarter 2: \$**29654.23** [Enter]

Division 1, Quarter 3: \$**32982.54** [Enter]

Division 1, Quarter 4: \$**39651.21** [Enter]

Division 2, Quarter 1: \$**56321.02** [Enter]

Division 2, Quarter 2: \$**54128.63** [Enter]

Division 2, Quarter 3: \$**41235.85** [Enter]

Division 2, Quarter 4: \$**54652.33** [Enter]

Division 3, Quarter 1: \$**29654.35** [Enter]

Division 3, Quarter 2: \$**28963.32** [Enter]

Division 3, Quarter 3: \$**25353.55** [Enter]

Division 3, Quarter 4: \$**32615.88** [Enter]

The total sales for the company are: \$456782.34

# 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

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

# 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}};
```



# 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;
}
```

# 7.9 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];
```

- When used as parameter, specify all but 1<sup>st</sup> dimension in prototype, heading:

```
void getRectSolid(short[][3][5]);
```

# Summary

- What is an array?
- Array declaration.
- Array Initialization.
- How to access array?
- Range-Based for loop.
- Array assignment.
- Parallel array.
- N-dimensional array