Chapter 7: Arrays

7.1 Arrays Hold Multiple Values

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**:
  
  ```c
  int tests[5];
  ```

Array - Memory Layout

- **The definition**:
  
  ```c
  int tests[5];
  ```

  **allocates the following memory**:

- **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
    - (number of elements) * (number of bytes for each element)

  - **Examples**:
    
    ```c
    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.
  ```cpp
  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

Accessing Array Elements

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

<table>
<thead>
<tr>
<th>Subscripts</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>

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:
  ```cpp
  tests[0] = 79;
  cout << tests[0];
  cin >> tests[1];
  tests[4] = tests[0] + tests[1];
  ```
- Arrays must be accessed via individual elements:
  ```cpp
  cout << tests; // not legal
  ```

Accessing Array Contents

- 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;
  ```
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

The variable count starts at 0, which is the first valid subscript value. The loop ends when the variable count reaches 5, which is the first invalid subscript value.

Default Initialization

- Global array → all elements initialized to 0 by default
- Local array → all elements **uninitialized** by default

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 7-5

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

```cpp
const int SIZE = 3; // Constant for the array size
int values[SIZE]; // An array of 3 integers
int count; // Loop counter variable

// Attempt to store five numbers in the three-element array
for (count = 0; count < 5; count++)
    values[count] = 100;
```
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:

```cpp
// 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;
```

Array Initialization

- Arrays can be initialized with an initialization list:

```cpp
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.

Partial Array Initialization

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

```cpp
int numbers[7] = {1, 2, 4, 8};
```
Implicit Array Sizing

- Can determine array size by the size of the initialization list:
  ```cpp
  int quizzes[]={12,17,15,11};
  ```

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

---

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.

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

---

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.
- Program 7-12 demonstrates...
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;

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:

  ```cpp
  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:

  ```cpp
  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:

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

Summing and Averaging Array Elements

- Use a simple loop to add together array elements:

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

Summing and Averaging Array Elements

- In C++ 11 you can use the range-based for loop, as shown here:

  ```cpp
  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

- In C++ 11 you can use the range-based for loop, as shown here:

  ```cpp
  int count;
  int highest;
  highest = numbers[0];
  for (count = 1; count < SIZE; count++)
      if (numbers[count] > highest) // If the next number is greater than 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

```c
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.

Comparing Arrays

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

```c
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
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";
```

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

```c
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;
}
```

7.7 Using Parallel Arrays

7.8 Arrays as Function Arguments
Arrays as Function Arguments

- To pass an array to a function, just use the array name:
  ```cpp
  showScores(tests);
  ```
- To define a function that takes an array parameter, use empty `[]` for array argument:
  ```cpp
  void showScores(int[]);
  // function prototype
  ```
- `void` showScores(int tests[])
  // function header

When passing an array to a function, it is common to pass array size so that function knows how many elements to process:
```cpp
showScores(tests, ARRAY_SIZE);
``` Array size must also be reflected in prototype, header:
```cpp
void showScores(int[], int);
// function prototype
```    
```cpp
void showScores(int tests[], int size)
// function header
``` 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 Arrays
Two-Dimensional Arrays

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

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

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

2D Array Initialization

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

  ```c
  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 or NULL

Example – The showArray Function from Program 7-22

```c
void showArray(int array[][COLS], int rows)
{
    for (int x = 0; x < rows; x++)
    {
        for (int y = 0; y < COLS; y++)
        {
            cout << array[x][y] << " ";
        }
        cout << endl;
    }
}
```

Two-Dimensional Array Representation

- Use two subscripts to access element:

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

Two-Dimensional Array as Parameter, Argument

- Use array name as argument in function call:

  ```c
  getExams(exams, 2);
  ```

- Use empty [] for row, size declarator for column inprototype, header:

  ```c
  const int COLS = 2;
  // Prototype
  void getExams(int [][COLS], int);
  // Header
  void getExams(int exams[][COLS], int rows)
  ```

Example – The showArray Function from Program 7-22

```c
int table1[TABLE_SIZE][COLS] = {{1, 2, 3, 4},
                                 {5, 6, 7, 8},
                                 {9, 10, 11, 12}};
int table2[TABLE_SIZE][COLS] = {{19, 20, 30, 40},
                                 {50, 60, 70, 80},
                                 {90, 100, 110, 120},
                                 {130, 140, 150, 160}};
```

How showArray is Called

```c
cout << "The contents of table1 are:\n";
showArray(table1, TABLE_SIZE);
cout << "The contents of table2 are:\n";
showArray(table2, TABLE_SIZE);
```
Summing All the Elements in a Two-Dimensional Array

- Given the following definitions:

```c
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, 8},
  {9, 9, 8, 3, 1},
  {6, 2, 7, 4, 1}};
```

```c
// 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;
```

Arrays with Three or More Dimensions

- Can define arrays with any number of dimensions:
  ```c
  short rectSolid[2][3][5];
  double timeGrid[3][4][3][4];
  ```
- When used as parameter, specify all but 1\textsuperscript{st} dimension in prototype, heading:
  ```c
  void getRectSolid(short [][][3][5]);
  ```