Chapter 8: Arrays

Starting Out with C++
Early Objects
Ninth Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda

Topics

8.1 Arrays Hold Multiple Values

8.2 Accessing Array Elements

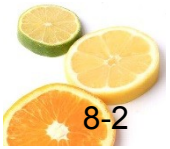
8.3 Inputting and Displaying Array Contents

8.4 Array Initialization

8.5 The Range-Based `for` loop

8.6 Processing Array Contents

8.7 Using Parallel Arrays



Topics (continued)

8.8 The `typedef` Statement

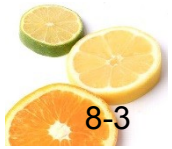
8.9 Arrays as Function Arguments

8.10 Two-Dimensional Arrays

8.11 Arrays with Three or More Dimensions

8.12 Vectors

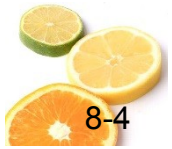
8.13 Arrays of Objects



8.1 Arrays Hold Multiple Values

- Array: a variable that can store multiple values of the same type
- The values are stored in consecutive memory locations
- It is declared using `[]` operator

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

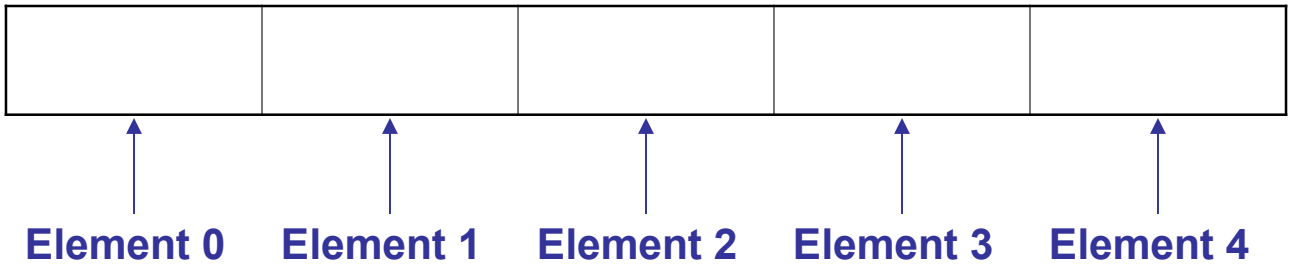


Array Storage in Memory

The definition

```
int tests[ISIZE];  // ISIZE is 5
```

allocates the following memory

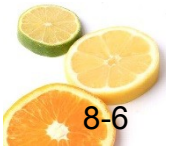


Array Terminology

In the definition `int tests[ISIZE];`

- **`int`** is the data type of the array elements
- **`tests`** is the name of the array
- **`ISIZE`**, in **`[ISIZE]`**, is the size declarator. It shows the number of elements in the array.
- The size of an array is the number of bytes allocated for it

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

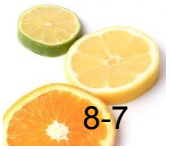


Array Terminology Examples

Examples:

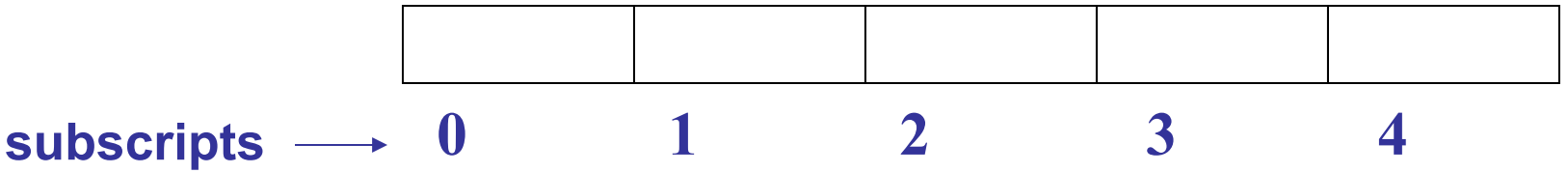
Assumes **`int`** uses 4 bytes and **`double`** uses 8 bytes

```
const int ISIZE = 5, DSIZE = 10;
int tests[ISIZE]; // holds 5 ints, array
                  // occupies 20 bytes
double volumes[DSIZE];// holds 10 doubles,
                      // array occupies
                        // 80 bytes
```



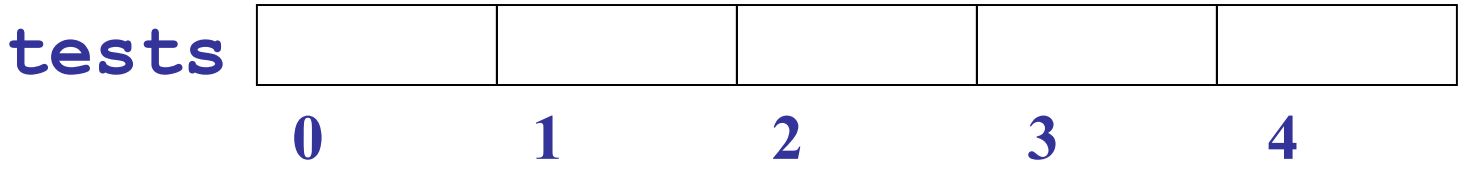
8.2 Accessing Array Elements

- Each array element has a subscript, used to access the element.
- Subscripts start at 0



Accessing Array Elements

Array elements (accessed by array name and subscript) can be used as regular variables

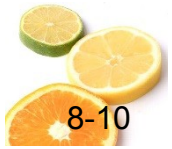



```
tests[0] = 79;
cout << tests[0];
cin  >> tests[1];
tests[4] = tests[0] + tests[1];
cout << tests; // illegal due to
               // missing subscript
```

8.3 Inputting and Displaying Array Contents

cout and **cin** can be used to display values from and store values into an array

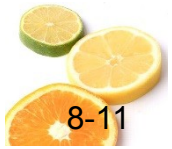
```
const int ISIZE = 5;

int tests[ISIZE]; // Define 5-elt. array
cout << "Enter first test score ";
cin  >>  tests[0];
```



Array Subscripts

- Array subscript can be an integer constant, integer variable, or integer expression
- Examples:

	Subscript is
`cin >> tests[3];`	int constant
`cout << tests[i];`	int variable
`cout << tests[i+j];`	int expression

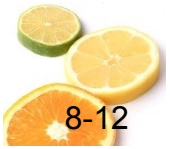


Accessing All Array Elements

To access each element of an array

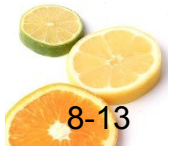
- Use a loop
- Let the loop control variable be the array subscript
- A different array element will be referenced each time through the loop

```
for (i = 0; i < 5; i++)
    cout << tests[i] << endl;
```



Getting Array Data from a File

```
const int ISIZE = 5;
int sales[ISIZE];
ifstream dataFile;
dataFile.open("sales.dat");
if (!dataFile)
   cout << "Error opening data file\n";
else  // Input daily sales
{  for (int day = 0; day < ISIZE; day++)
      dataFile >> sales[day];
   dataFile.close();
}
```



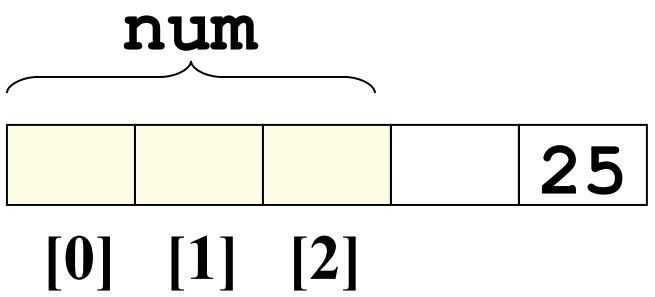
Sending Array Data to a File

- Open the file using an **`ofstream`** object
- Use a loop to write each array element to the file
- Close the file

No Bounds Checking

- There are no checks in C++ that an array subscript is in range
- An invalid array subscript can cause the program to overwrite other memory
- Example:

```
const int ISIZE = 3;
int i = 4;
int num[ISIZE];
num[i] = 25;
```



Off-By-One Errors

- These most often occur when a program accesses data one position beyond the end of an array, or misses the first or last element of an array.

- If an array has size **`n`**, then the subscripts range from `0` to **`n-1`**

8.4 Array Initialization

- An array can be initialized during program execution with assignment statements

```
tests[0] = 79;
tests[1] = 82; // etc.
```

- It can be initialized at array definition with an initialization list

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

Start at element 0 or 1?

- You may choose to declare arrays to be one larger than needed. This allows you to use the element with subscript 1 as the 'first' element, etc., and may minimize off-by-one errors.
- The element with subscript 0 is not used.
- This is most often done when working with ordered data, *e.g.*, months of the year or days of the week

Partial Array Initialization

- If array is initialized at definition with fewer values than the size declarator of the array, the remaining elements will be set to 0 or the empty string

```
int tests[ISIZE] = {79, 82};
```

79	82	0	0	0

- Initial values are used in order; you cannot skip over elements to initialize a noncontiguous range
- You cannot have more values in the initialization list than the declared size of the array

Implicit Array Sizing

- C++ can determine the array size by the size of the initialization list

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

12	17	15	11

- You must use either array size declarator or an initialization list when the array is defined

Alternate Ways to Initialize Variables

- You can initialize a variable at definition time using a functional notation

```
int length(12);  // same result as
                 // int length = 12;
```

- In C++ 11 and higher, you can also do this:

```
int length{12};
```

- The second approach checks the argument to ensure that it matches the data type of the variable, and will generate a compiler error if not

8.5 The Range-Based `for` Loop

- NOTE: This is a C++ 11 feature
- This is a loop that can simplify array processing.
- It uses a variable that will hold a different array element for each iteration
- Format:

```
for (data_type var : array)
  statement;
```

- **`data_type`** : the type of the variable
- **`var`**: the variable
- **`statement;`** : the loop body

Range-Based `for` Loop - Details

- **`data_type`** must be the type of the array elements, or a type that the array elements can be automatically converted to
- **`var`** will hold the value of successive array elements as the loop iterates. Each array element is processed in the loop
- **`statement;`** can be a single statement or a block of statements enclosed in { }

Range-Based `for` Loop - Example

```
// sum the elements of an array
int [] grades = {68,84,75};
int sum = 0;
for (int score : grades)
   sum += score;
```

Range-Based `for` Loop - Example

```
// modify the contents of an array
const int ISIZE = 3;
int [ISIZE] grades;
for (int &score : grades)
{
   cout << "Enter a score: ";
   cin >> score;
}
```

Comparison: Range-Based `for` Loop vs. Regular `for` Loop

- The range-based `for` loop provides a simple notation to use to process all of the elements of an array.
- However, it does not give you access to the subscripts of the array elements.
- If you need to know the element locations as well as the element values, then use a regular `for` loop.

8.6 Processing Array Contents

- Array elements can be
  - treated as ordinary variables of the same type as the array
  - used in arithmetic operations, in relational expressions, etc.
- Example:

```
if (principalAmt[3] >= 10000)
  interest = principalAmt[3] * intRate1;
else
  interest = principalAmt[3] * intRate2;
```

Using Increment and Decrement Operators with Array Elements

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

```
tests[i]++;  // adds 1 to tests[i]
tests[i++];  // increments i, but has
             // no effect on tests
```

Copying One Array to Another

- You cannot copy with an assignment statement:

```
tests2 = tests;  //won't work
```

- You must instead use a loop to copy element-by-element:

```
for (int indx=0; indx < ISIZE; indx++)
   tests2[indx] = tests[indx];
```

Are Two Arrays Equal?

- Like copying, you cannot compare two arrays in a single expression:

```
if (tests2 == tests)
```

- You can use a while loop with a **bool** variable:

```
bool areEqual=true;
int indx=0;
while (areEqual && indx < ISIZE)
{
   if(tests[indx] != tests2[indx]
      areEqual = false;
   indx++;
}
```

Find the Sum, Average of Array Elements

- Use a simple loop to add together array elements

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

- Or use C++ 11 range-based **`for`** loop:

```
for (int num : tests)
    sum += num;
```

- Once summed, average can be computed

```
average = sum/ISIZE;
```

Find the Largest Array Element

- Use a loop to examine each element and find the largest element (*i.e.*, one with the largest value)

```
int largest = tests[0];
for (int tnum = 1; tnum < ISIZE; tnum++)
{  if (tests[tnum] > largest)
      largest = tests[tnum];
}
cout << "Highest score is " << largest;
```

- A similar algorithm exists to find the smallest element

Using Arrays vs. Using Simple Variables

- An array is probably not needed if the input data is only processed once:
  - Find the sum or average of a set of numbers
  - Find the largest or smallest of a set of values
- If the input data must be processed more than once, an array is probably a good idea:
  - Calculate the average, then determine and display which values are above the average, at the average, and below the average

Partially-Filled Arrays

- The exact amount of data (and, therefore, the array size) may not be known when a program is written.
- The programmer makes a best estimate for the maximum amount of data, then sizes arrays accordingly. A sentinel value can be used to indicate end-of-data.
- The programmer must also keep track of how many array elements are actually used

C-Strings and `string` Objects

They can be processed using the array name

- Entire string at once, or
- One element at a time by using a subscript

```
string city;
cout << "Enter city name: ";
cin  >> city;
```

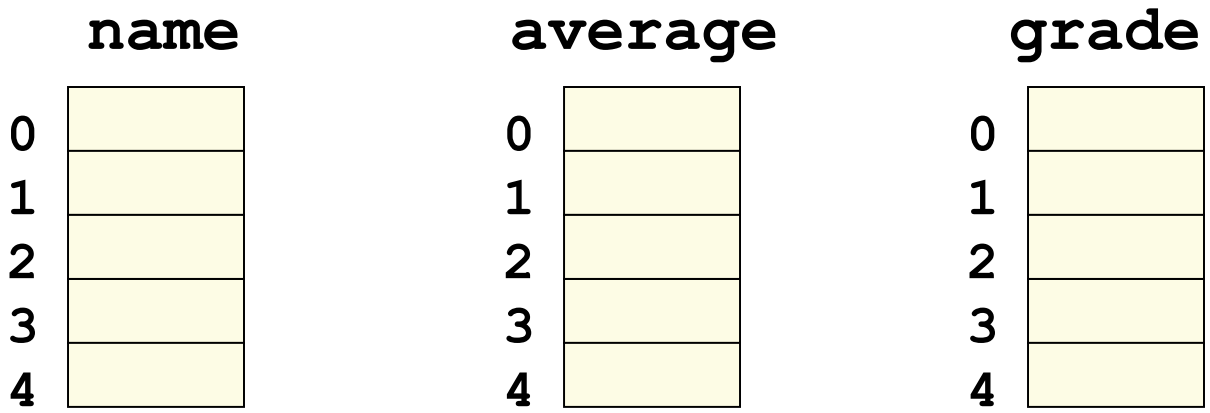
'S'	'a'	'l'	'e'	'm'
city[0]	city[1]	city[2]	city[3]	city[4]

8.7 Using Parallel Arrays

- Parallel arrays: two or more arrays that contain related data
- The subscript is used to relate arrays
  - elements at same subscript are related
- The arrays do not have to hold data of the same type

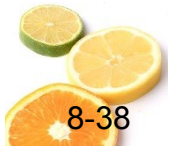
Parallel Array Example

```
const int ISIZE = 5;
string name[ISIZE];   // student name
float average[ISIZE]; // course average
char grade[ISIZE];    // course grade
```



Parallel Array Processing

```
const int ISIZE = 5;
string name[ISIZE];   // student name
float average[ISIZE]; // course average
char grade[ISIZE];    // course grade
...
for (int i = 0; i < ISIZE; i++)
     cout << " Student: " << name[i]
          << " Average: " << average[i]
          << " Grade: "   << grade[i]
          << endl;
```

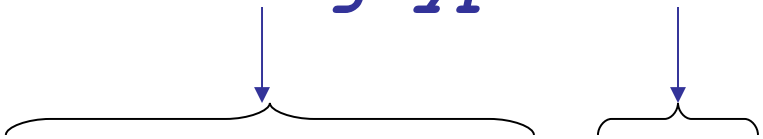


8.8 The `typedef` Statement

- Creates an alias for a simple or structured data type
- Format:

```
typedef existingType newName;
```

- Example:



```
typedef unsigned int Uint;
Uint tests[ISIZE]; // array of
                   // unsigned ints
```

Uses of `typedef`

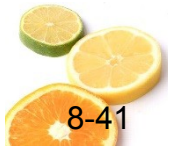
- It can be used to make code more readable
- Can be used to create an alias for an array of a particular type

```
// Define yearArray as a data type
// that is an array of 12 ints
typedef int yearArray[MONTHS];

// Create two of these arrays
yearArray highTemps, lowTemps;
```

8.9 Arrays as Function Arguments

- Passing a single array element to a function is no different than passing a regular variable of that data type
- The function does not need to know that the value it receives is coming from an array

```
displayValue(score[i]);      // call
void displayValue(int item) // header
{  cout << item << endl;
}
```



Passing an Entire Array

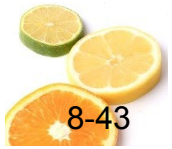
- To define a function that has an array parameter, use empty `[]` to indicate the array in the prototype and header
- To pass an array to a function, just use the array name

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

// Function header
void showScores(int tests[])

// Function call
showScores(tests);
```

Passing an Entire Array

- Use the array name, without any brackets, as the argument
- You can also pass the array size as a separate parameter so the function knows how many elements to process

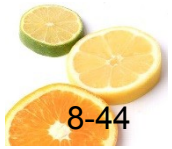
```
showScores(tests, 5);          // call
void showScores(int[], int); // prototype
void showScores(int A[],
                int size) // header
```



Using `typedef` with a Passed Array

You can use **`typedef`** to simplify function prototype and header

```
// Make intArray an integer array
// of unspecified size
typedef int intArray[];

// Function prototype
void showScores(intArray, int);

// Function header
void showScores(intArray tests,
                int size)
```



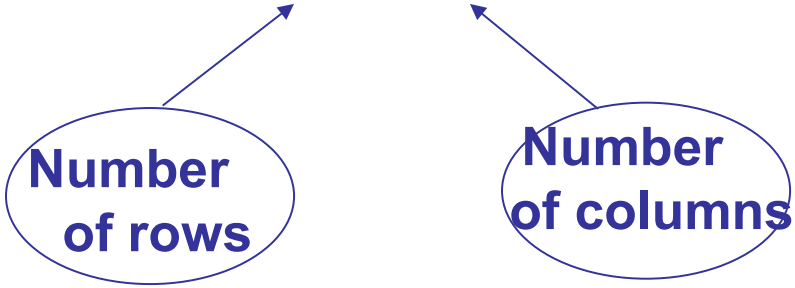
Modifying Arrays in Functions

- Array parameters in functions are similar to reference variables
- Changes made to array passed to a function are made to the actual array in the calling function
- The programmer must be careful that an array is not inadvertently changed by a function
- The `const` keyword can be used in prototype and header to prevent changes

8.10 Two-Dimensional Arrays

- You can define one array for multiple sets of data
- It is like a table in a spreadsheet
- Use two size declarators in definition

```
int exams[4][3];
```



Two-Dimensional Array Representation

```
int exams[4][3];
```

	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][1] = 86;
```

Two-Dimensional Array Access

When you use access an element of a two-dimensional array

```
exams[2][1] = 86;
```

- The first subscript, `[2]`, indicates the row in the array
- The second subscript, `[1]`, indicates the column in the array

Initialization at Definition

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

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

84	78
92	97

- Can omit inner `{ }`

Passing a Two-Dimensional Array to a Function

- Use the array name and the number of columns as arguments in the function call

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

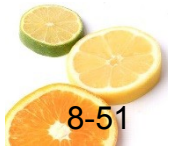
- Use empty `[]` for row and a size declarator for the number of columns in the prototype and header

```
// Prototype, where NUM_COLS is 2
void getExams(int[][NUM_COLS], int);

// Header
void getExams
          (int exams[][NUM_COLS], int rows)
```

Using `typedef` with a Two-Dimensional Array

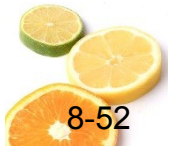
Can use `typedef` for simpler notation

```
typedef int intExams[][2];
   ...
// Function prototype
void getExams(intExams, int);

// Function header
void getExams(intExams exams, int rows)
```



Two-Dimensional Array Traversal

- Use nested loops, one for the row and one for the column, to visit each array element.
- Accumulators can be used to calculate the sum the elements row-by-row, column-by-column, or over the entire array.



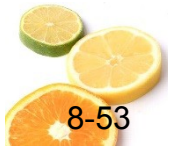
8.11 Arrays with Three or More Dimensions

- You can define arrays that have any number of dimensions

```
short rectSolid(2,3,5);
double timeGrid(3,4,3,4);
```

- When this type of array is used as a parameter, specify the size of all but the 1st dimension

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



8.12 Vectors

- A vector holds a set of elements, like a one-dimensional array
- It has a flexible number of elements. It can grow and shrink
  - There is no need to specify the size when defined
  - Space is automatically added as needed
- Defined in the Standard Template Library (STL)
  - Covered in a later chapter
- Must include **`vector`** header file to use vectors

```
#include <vector>
```

Vectors

- It can hold values of any data type, specified when the vector is defined

```
vector<int> scores;
vector<double> volumes;
```

- You can specify initial size if desired

```
vector<int> scores(24);
```

- Use `[]` to access individual elements

Defining Vectors

- Define a vector of integers (starts with 0 elements)

```
vector<int> scores;
```

- Define **int** vector with initial size 30 elements

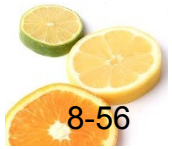
```
vector<int> scores(30);
```

- Define 20-element **int** vector and initialize all elements to 0

```
vector<int> scores(20, 0);
```

- Define **int** vector initialized to size and contents of vector **finals**

```
vector<int> scores(finals);
```



C++ 11 Features for Vectors

- C++ 11 supports vector definitions that use an initialization list

```
vector<int> scores {88, 67, 79, 84};
```

- Note: no = operator between the vector name and the initialization list
- A range-based for loop can be used to access the elements of a vector

```
for (int test : scores)
    cout << test << " ";
```

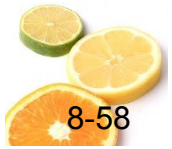
Growing a Vector's Size

- Use the **push_back** member function to add an element to a full vector or to a vector that had no defined size

```
// Add a new element holding a 75
scores.push_back(75);
```

- Use the **size** member function to determine the number of elements currently in a vector

```
howbig = scores.size();
```



Removing Vector Elements

- Use the **pop_back** member function to remove the last element from a vector

```
scores.pop_back();
```

- Note: **pop_back** removes the last element but does not return it
- To remove all of the elements from a vector, use the **clear** member function

```
scores.clear();
```

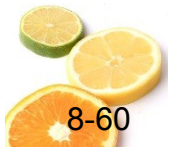
- To determine if a vector is empty, use **empty** member function

```
while (!scores.empty()) ...
```

8.13 Arrays of Objects

- Objects can be used as array elements

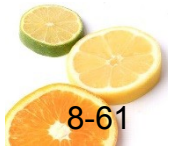
```
class Square
{ private:
     int side;
  public:
     Square(int s = 1)
     { side = s; }
     int getSide()
     { return side; }
};
Square shapes[10];  // Create array of 10
                    // Square objects
```



Arrays of Objects

- Use the array subscript to access a specific object in the array
- Then use dot operator to access the member methods of that object

```
for (i = 0; i < 10; i++)
  cout << shapes[i].getSide() << endl;
```



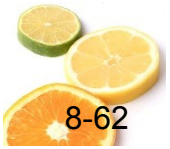
Initializing Arrays of Objects

- You can use the default constructor to perform the same initialization for all objects
- You can use an initialization list to supply specific initial values for each object

```
Square shapes[5] = {1,2,3,4,5};
```

- The default constructor is used for the remaining objects if initialization list is too short

```
Square boxes[5] = {1,2,3};
```

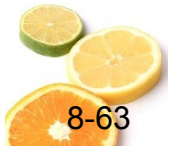


Initializing Arrays of Objects

- If an object is initialized with a constructor that takes > 1 argument, the initialization list must include a call to the constructor for that object

```
Rectangle spaces[3]={ Rectangle(2,5),
                      Rectangle(1,3),
                      Rectangle(7,7)
                    };
```

- The same constructor does not have to be used for every object that is being initialized

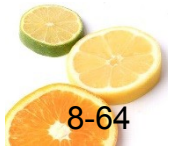


Arrays of Structures

- Structures can be used as array elements

```
struct Student
{
      int studentID;
      string name;
   short year;
      double gpa;
};
const int CSIZE = 30;
Student class[CSIZE]; // Holds 30
                      // Student structures
```

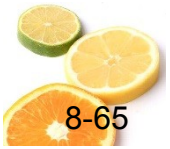
- An array of structures can be used as an alternative to parallel arrays



Arrays of Structures

- Use the array subscript to access a specific structure in the array
- Then use the dot operator to access members of that structure

```
cin  >> class[25].studentID;

cout << class[i].name << " has GPA "
     << class[i].gpa << endl;
```



Chapter 8: Arrays

Starting Out with C++
Early Objects
Ninth Edition

by Tony Gaddis, Judy Walters,
and Godfrey Muganda