CLASS TEMPLATES

ARRAY **AND** VECTOR

CATCHING EXCEPTIONS

Chapter 7

TOPICS

Introduction to Data Structures

C++ Standard Library Class Template array

- Examples Using Arrays
- Range-Based for Statement
- Case Study: Class GradeBook Using an Array to Store Grades
- Sorting and Searching Arrays

Multidimensional Arrays

- Case Study: Class GradeBook Using a Two-Dimensional Array

C++ Standard Library Class Template vector

Brief Introduction to Exception Handling

DATA STRUCTURES

Data Structure is a Collection of Related Data

Data structure

- Collection of related data items
- Examples: array, linked list, stack, queue, tree

Can be created from scratch, or use existing libraries

Two types discussed in this chapter:

- **array** – fixed-size collection of data items of the same type
- **vector** – dynamically-sized collection of data items of the same type

Both are Standard C++ Library **class templates**

C++ Standard Library
Class Template array

ARRAY

Data structure consisting of related data items of the **same type**

Makes it convenient to process related groups of values

Remains the same length once created

Each item in array is an **element**

ARRAY ELEMENTS

Can be either fundamental types or object types

To refer to a single element, use the element's **index**

Array element is an **lvalue**

LVALUES AND RVALUES

lvalue

- Non-constant variables
- A specific memory location
- Can be thought of as **container**
- Can be used on **left side** of assignment operator, so its value can be changed
- Can be used on **right side** of an assignment, so its value can be used or read

rvalue

- Constant value; a literal value
- Can be thought of as **the thing in a container**
- Cannot be used on **left side** of assignment operator, so its value cannot be changed
- Can be used on **right side** of an assignment, so its value can be used or read

EXAMPLES OF LVALUES AND RVALUES

Statement	Explanation
int x;	• x refers to memory space (a container) that can hold an integer • x is an **lvalue**
x = 10;	• Literal value 10 is placed into the memory location referred to by x • x is an **lvalue** • 10 is an **rvalue**
10 = x;	• Since all literal constants are rvalues, 10 cannot appear on the left side of an assignment operator. • This would not compile.
int y = x;	• y is an **lvalue** • x is also an **lvalue**, and **lvalues** can appear on the right side of an assignment operator. Its value is copied into y.

Examples of lvalues and rvalues – Array Elements

Statement	Explanation
arr[0] = 10;	• arr is an array; arr[0] refers to one element in this array • Literal value 10 is placed into the memory location referred to by arr[0] • arr[0] is an **lvalue** • 10 is an **rvalue**
10 = arr[0];	• Since all literal constants are rvalues, 10 cannot appear on the left side of an assignment operator. • This would not compile.
arr[0] = arr[1];	• arr[0] is an **lvalue** • arr[1] is also an **lvalue**, and **lvalues** can appear on the right side of an assignment operator. Its value is copied into arr[0].

ARRAY SIZE

Arrays know their own size, via the size() member function:

arrayName.size()

Creating an Array in C++

There are two ways to create an array in C++

"C style" can still be used: **int nameOfArray[10]**

In this chapter, we're going to use the C++ array class template style

To create arrays this way, the <array> header must be included:

#include <array>

CREATING AN ARRAY USING THE <ARRAY> CLASS TEMPLATE

General format:

array < *data type*, *size* > *arrayName*;

Keyword **array**

<> notation indicates array is a **class template**

This means any data type can be used (fundamental type or object)

Note comma between the data type and the size

EXAMPLES USING ARRAYS

USING A LOOP TO INITIALIZE AN ARRAY'S ELEMENTS

Fig07_03.cpp

Declares and initializes a 5-element array

<array> header required (line 5)

Note comma between int and 5 on line 10

for statement used to initialize the array (lines 12-14)

- arrays declared **locally** are **non-static**, and therefore **do not have initial values**
- arrays declared **static** are **initialized to zero**

USING A LOOP TO INITIALIZE AN ARRAY'S ELEMENTS – SIZE_T TYPE

Note **size_t** in the for loops

Alias of the fundamental **unsigned integer** data type

It is a type able to represent the **size of any object in bytes**

Recommended for **array sizes** and **subscripts**

Defined in std namespace, and library <cstddef>

Initializing an array in a Declaration with an Initializer List

Array elements can be initialized using an **initializer list**

- Comma-separated list of values within curly braces

General format:

array < *data type*, *size* > *arrayName*{*initVal1, initVal2, …*};

Fig07_04.cpp

INITIALIZING AN ARRAY IN A DECLARATION WITH AN INITIALIZER LIST, CONT'D

Number of initializers must be less than or equal to size of array

If **fewer** initializers than declared elements:

- Remaining elements initialized to zero

So to initialize all elements to 0, could write this, for example:

```
array< int, 5 > n = {};
```

But only once, at the point of declaration.

Specifying an Array's Size with a Constant Variable

Fig07_05.cpp

To declare a constant variable, use the keyword **const** (line 5)

Remember that these must be initialized at declaration and then cannot be changed later

Also called **named constants** or **read-only variables**

If named with a meaningful name, can be useful for self-documenting code

SUMMING THE ELEMENTS OF AN ARRAY

Fig07_06.cpp

USING BAR CHARTS TO DISPLAY ARRAY DATA GRAPHICALLY

Fig07_07.cpp

Using the Elements of an array as Counters

Fig07_08.cpp

Frequency of dice rolls using an array

- Like Fig06_07.cpp

One line (line 21) replaces entire switch in earlier version

Use the dice value as the index to the frequency array

Note that we're using a 7-element array, and ignoring the first element, at index 0

USING ARRAYS TO SUMMARIZE SURVEY RESULTS

Uses arrays to summarize results of data collected in a survey:

Twenty students rate food in student cafeteria on a scale of 1 to 5. Place the 20 responses in an integer array and determine the frequency of each rating.

Fig07_09.cpp

No **automatic bounds checking** when accessing an array element using [] – we'll see an example of built-in bounds checking later.

Buffer overflow – trying to access memory past the end of the array

STATIC LOCAL ARRAY

Remember that a **static local variable** in a function exists for program's duration (and therefore keeps its value throughout the life of the program), but is **visible** only in function's body

We can create a local array using **static**, so that it is not created, initialized, and destroyed every time the function is called and then ends

- Improves performance

Static local arrays are automatically initialized to 0, if not explicitly initialized by programmer

STATIC LOCAL ARRAY EXAMPLE

Fig07_10.cpp

staticArrayInit function (line 23-40) with a static local array (line 27)

- Values in array are **KEPT**, and are the same when the function is called again as they were when the function ended

automaticArrayInit function (line 43-60) with automatic local array (line 46)

- Values in array are **NOT** kept, and the array is rebuilt, and reinitialized when the function is called again

RANGE-BASED FOR STATEMENT

RANGE-BASED FOR STATEMENT DEFINITION

Common to process all elements in an array, from start to finish

Range-based for statement does this **without needing a counter**

Avoids possibility of accessing past end of array – no need for bounds checking

The items in the array are processed one after the other, from start to finish, but without using a subscript

Use if you don't need to access/use a subscript

RANGE-BASED FOR STATEMENT GENERAL FORMAT

General format:

```
for ( rangeVariableDeclaration : expression ) {
    statements in loop;
}
```

rangeVariableDeclaration has a type and an identifier

- Like the parameter in a function header

expression is the name of the array

RANGE-BASED FOR STATEMENT EXAMPLE SYNTAX

Example:

```
for ( int item : myArray )

    cout << item << endl;
```

Array is **myArray**; each element of the array saved to **item**

for each iteration of the loop, assign the next element of the **myArray** array to the variable **item**, then execute the loop's body, which prints the value of each item

RANGE-BASED FOR STATEMENT EXAMPLE IN CODE

Fig07_11.cpp

Note loop in lines 12-14

Within this for loop, the variable **item** is **local**

If we were to change item's value within this loop, it would not change the contents of the array

The range variable is local to the loop, and the value of each element of the array is **copied** into it

USING RANGE-BASED FOR TO CHANGE AN ARRAY

Line 17 uses **reference variable**

Remember a reference variable is an **alias** for the variable it's "pointing" to

Can use this approach to change the actual values in the array

TO ACCESS AN ELEMENT'S SUBSCRIPT (INDEX)

Can't use range-based **for** if it is important to know the element's subscript (index)

Use a standard counter-controlled for instead

Remember – use range-based for when you don't need to deal with the element's subscript

Case Study: Class GradeBook Using an Array to Store Grades

CASE STUDY: CLASS GRADEBOOK

Fig07_14.cpp; GradeBook.h

Previous versions of this GradeBook example process a set of grades entered by user, but do not store those individual grade values

Repeat calculations, like calculating an average, or finding the highest or lowest grades, or determining the frequency of grade values require user to reenter same grades

We solve this problem by storing grades in an **array**.

GradeBook Class (GradeBook.h)

The array is a data member of the GradeBook class

- Its size is determined by a value passed to class constructor
- So the GradeBook object in this example can process a variable number of grades.

Variable **students** is used as the size (line 12)

- Declared **public**, **static** and **const**
- Public – it can be accessed outside of class
- Static – all objects of the GradeBook class share it – all GradeBook objects store grades for the same number of students
- Const – value cannot be changed after it's initialized

A Note About Static Data Members

A variable like students, that is declared as a **static data member**, is also called a **class variable**

All objects of the class **share this variable and its value**

Can be accessed outside of class **even when no objects of the class exist**

More about static data members in chapter 9

BACK TO THE GRADEBOOK CLASS (GRADEBOOK.H)

GradeBook constructor has two parameters (lines 15-18):

1. A **reference** to an array of integers
   - **More efficient** than copying an array into an argument and passing it, where its values are then copied a second time
   - Values are **copied from original array** into data member array of class
2. A **reference** to string
   - More efficient for same reason
   - Both parameters are **const** because the class has no need to change them

Note that the setCourseName function has reference parameters too (line 21)

FUNCTIONS WITHIN GRADEBOOK CLASS (GRADEBOOK.H)

processGrades function (lines 38-50)

- Calls other functions to process grades

outputGrades function (lines 135-138)

- Note **for** loop, and **student + 1** value

getAverage function (lines 83-93)

- Note **cast** in line 92

outputBarChart function (lines 104-106)

- Uses another array for frequency distribution

DRIVER

Fig07_14.cpp

Note use of scope resolution operator :: to access the static constant **students** (lines 9-10)

SORTING AND SEARCHING ARRAYS

Functions for Sorting and Searching Arrays

Built-in C++ Standard Library functions for sorting and searching

Sorting

- Putting data in some kind of order
- Common process when working with data
- Standard library function sort()

Searching

- Checking if an array holds a certain value, called a **key value**
- Standard library function binary_search()

DEMONSTRATING ARRAY FUNCTIONS SORT AND BINARY_SEARCH

Fig07_15.cpp

sort() function

- Takes two arguments – the start and end of the range to be sorted
- **begin()** and **end()** are **iterators** that mean the beginning and end of the array (more in chapter 15)

binary_search() function

- items must be sorted in ascending order first
- arguments are the start and end of range to be searched, and key to search for
- returns bool indicating if key is found (recall that bool values are true or false)

MULTIDIMENSIONAL ARRAYS

Multidimensional Array Definition

Represents data in rows and columns (table)

Each element must be identified with two subscripts

- By convention, row then column

A multidimensional array is really a single-dimensional array of arrays…

Fig07_17.cpp

DECLARING A MULTIDIMENSIONAL ARRAY – STEP 1

Remember the general format to declare an array:

array < *data type*, *size* > *arrayName*;

When declaring a multidimensional array, we replace the data type with another array declaration.

This is called a **nested array declaration syntax**

It creates an array of arrays

Declaring a Multidimensional Array – Step 2

array < *data type*, *size* > *arrayName*;

This part

Is replaced with this:

array < *data type*, *size* >

Which results in this:

array < array < *data type*, *size* >, *size* > *arrayName*;

DECLARING A MULTIDIMENSIONAL ARRAY – STEP 3

Returning to the example (Fig07_17.cpp), line 12 creates a multidimensional array called **array1**, and initializes it with the values in the curly braces:

array<array<int, columns>, rows> array1{1, 2, 3, 4, 5, 6};

Name of the multidimensional array:	array1
Data type of each ROW in this array:	an array of 3 integers
Data type of each ELEMENT in this array:	integer
Number of rows in this array:	2 – The value of the variable rows.
Number of columns in this array:	3 – The value of the variable columns.

DECLARING A MULTIDIMENSIONAL ARRAY

array<array<int, columns>, rows> array1{1, 2, 3, 4, 5, 6};

The initializer values are stored in the array like this:

1	2	3
4	5	6

Declaring a Multidimensional Array – Step 4

Line 13 creates another array called array2:

array<array<int, columns>, rows> array2{1, 2, 3, 4, 5};

Since this has fewer initializer values than elements, the remaining element is 0-filled:

1	2	3
4	5	0

Multidimensional Arrays – Nested Range-Based For Loops

Note the **nested range-based for loops** (lines 25-32)

Outer loop reads rows: **for (auto const& row : a)**

- **auto** keyword – compiler will **infer** (determine) a variable's data type based on the variable's initializer value
- So the variable **row** is set to an element from parameter **a**
- **a** is an array of arrays, so one element of it is an int array of size 3
- So **row** then becomes a reference to an int array of size 3

Inner loop reads each element in the row:
for (auto const& element : row)

- The variable **element** becomes a reference to each int in the row, one at a time

Multidimensional Arrays – Passing by Reference

printArray prototype and function definition (lines 9 & 23)

- The array is passed as a parameter **by reference**

Could have used nested counter-controlled loops for various processes

- See examples p. 310

CASE STUDY: CLASS GRADEBOOK USING A TWO-DIMENSIONAL ARRAY

CASE STUDY: CLASS GRADEBOOK – TWO-DIMENSIONAL ARRAY EXAMPLE PART 1

Stores 10 students' grades on 3 exams

Fig07_20.cpp; GradeBook.h

Note uses of **const** for functions and parameters

- What is the difference?

Note **range-based for loops**

- Why use counter-controlled for in outputBarChart and outputGrades functions?

Remember that a multidimensional array is an array of arrays

CASE STUDY: CLASS GRADEBOOK – TWO-DIMENSIONAL ARRAY EXAMPLE PART 2

What is the difference in using **const** for functions and parameters?

const applied to a **function** means the function cannot change the object where the function lives

const applied to a **parameter (or any variable)** means the variable's value cannot be changed

CASE STUDY: CLASS GRADEBOOK – TWO-DIMENSIONAL ARRAY EXAMPLE PART 3

Why is a counter-controlled for loop used in outputBarChart and outputGrades functions?

We need to access the array subscripts in order to determine how many asterisks to print, and to identify the student and test number.

If we need to know the subscript of an array element, because the subscript conveys some important information, then a counter-controlled for loop is used, not the range-based for loop.

C++ Standard Library
Class Template vector

INTRODUCTION TO C++ STANDARD LIBRARY CLASS TEMPLATE VECTOR

Similar to class template **array** but supports **dynamic resizing**

This means we can add and remove elements from it

Fig07_21.cpp

<vector> class library (line 5)

- In std namespace

CREATING VECTOR OBJECTS GENERAL FORMAT

Can create vectors of any data type, just like arrays

General format:

vector<*data type*> *name of vector* (*size*);

So line 13 in the example:

vector<int> integers1(7);

Creates a vector of integers of size 7

All elements in a vector initialized by default

CREATING VECTOR OBJECTS EXAMPLE

Note use of parentheses instead of braces here:

```
vector<int> integers1(7);
```

This statement passes an int size value to **vector object's constructor**

If braces are used instead, and the value in the braces is the same data type as the vector's data type, it's treated as an **initializer list**, and the vector would be size 1, and its element would have the value 7:

```
vector<int> integers1{7};
```

USING VECTORS

vector class has a member function size(), just like the array class

- line 17

inputVector function(lines 28-29)

- uses a range-based for with a reference, so values can be assigned to vector

Comparing vector Objects for Inequality

Can compare two vectors for equality/inequality using != and == operators (line 40)

Compares **contents** of the vectors, not the objects themselves

Initializing a Vector with Contents of Another

Can initialize new vector with contents of another using copy (line 46)

```
vector<int> integers3{integers1};
```

Invokes **copy constructor**

- We'll see this in more detail in chapter 10.

ASSIGNING A VECTOR

Can assign a vector to another vector (line 54)

integers1 = integers2;

The contents of the integers2 vector are copied to the integers1 vector

Using the [] Operator to Access and Modify vector Elements

We can use a regular counter-controlled loop to access vector elements, just like we can with arrays

Using the [] operator to obtain a vector element allows it to be used as either an rvalue or lvalue

Remember rvalue and lvalue:

- rvalue cannot be changed
- lvalue can be changed

Lines 69 and 73

No bounds checking though when using []

Brief Introduction to Exception Handling

Exception Handling Definitions

Exception

- A problem that occurs during program execution
- Often a run-time error that would cause the program to shut down

Exception-handling

- Handling the error so that program can continue to execute normally, or can terminate in a controlled way

Exception Handling Created and Handled

Exceptions are **created** by the **throws** statement (chapter 17)

Exceptions are **handled** by the **try/catch control statement**

Try block contains any code that might throw an exception

- So, code that may have a problem at runtime, like attempting to access an array or vector out of bounds

Catch block contains code to handle the exception

BACK TO OUR VECTOR EXAMPLE

Fig07_21.cpp

Vector function **at()** checks for an element at a particular subscript and throws **out_of_range** exception if subscript is invalid

- out_of_range exception declared in header <stdexcept>

Catch block executes if out_of_range exception generated in corresponding try block (line 82)

Exception object's **what()** function identifies problem (line 83)

- So a helpful error message can be displayed

CHANGING THE SIZE OF A VECTOR

Vector can be added to or removed from

- This is what makes a vector different from an array

Its size can be changed dynamically

- While the program is executing

Vector's **push_back()** function adds a new element to end of vector (line 87)

Line 89 shows new size of this vector