Chapter 7, Class Templates `array` and `vector`; Catching Exceptions

C++ How to Program, 9/e

OBJECTIVES

In this chapter you'll:

- Use C++ Standard Library class template `array`—a fixed-size collection of related data items.
- Use `array`s to store, sort and search lists and tables of values.
- Declare `array`s, initialize `array`s and refer to the elements of `array`s.
- Use the range-based `for` statement.
- Pass `array`s to functions.
- Declare and manipulate multidimensional `array`s.
- Use C++ Standard Library class template `vector`—a variable-size collection of related data items.

7.1 Introduction

- This chapter introduces the topic of data structures—*collections* of related data items.
- We discuss `arrays` which are *fixed-size* collections consisting of data items of the *same* type, and `vectors` which are collections (also of data items of the *same* type) that can grow and shrink *dynamically* at execution time.
- Both `array` and `vector` are C++ standard library class templates.
- After discussing how arrays are declared, created and initialized, we present examples that demonstrate several common array manipulations.

7.2 arrays

- An array is a *contiguous* group of memory locations that all have the same type.
- To refer to a particular location or element in the array, specify the name of the array and the position number of the particular element.
- Figure 7.1 shows an integer array called `c`.
- 12 elements.
- The position number is more formally called a subscript or index `(this number specifies the number of elements from the beginning of the array).`
- The first element in every array has subscript 0 (zero) and is sometimes called the zeroth element.
- The highest subscript in array `c` is 11, which is 1 less than the number of elements in the array (12).
- A subscript must be an integer or integer expression (using any integral type).

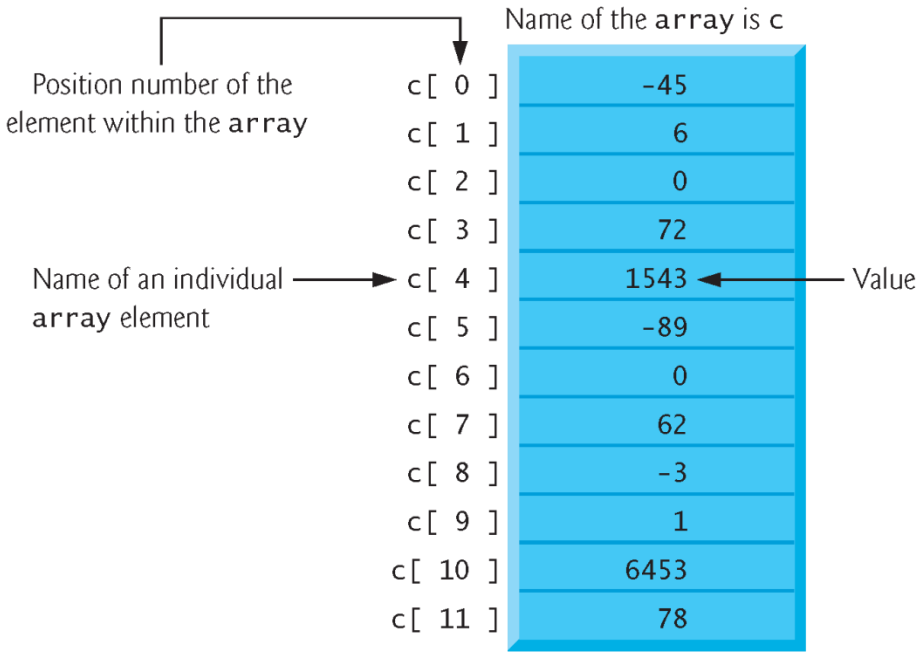

Fig. 7.1 | array of 12 elements.

Common Programming Error 7.1

Note the difference between the "seventh element of the `array`" and "`array` element 7." Subscripts begin at 0, so the "seventh element of the `array`" has a subscript of 6, while "`array` element 7" has a subscript of 7 and is actually the eighth element of the `array`. This distinction is a frequent source of off-by-one errors. To avoid such errors, we refer to specific `array` elements explicitly by their `array` name and subscript number (e.g., `c[6]` or `c[7]`).

Operators	Associativity	Type
`::` `()`	left to right *[See caution in Fig. 2.10 regarding grouping parentheses.]*	primary
`()` `[]` `++` `--` `static_cast<type>(operand)`	left to right	postfix
`++` `--` `+` `-` `!`	right to left	unary (prefix)
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<<` `>>`	left to right	insertion/extraction
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional

Fig. 7.2 | Precedence and associativity of the operators introduced to this point.

Operators	Associativity	Type
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 7.2 | Precedence and associativity of the operators introduced to this point.

7.3 Declaring `arrays`

- `array`s occupy space in memory.
- To specify the type of the elements and the number of elements required by an array use a declaration of the form:
  - `array`< *type, arraySize* > `arrayName;`
- The notation <*type, arraySize*> indicates that `array` is a class template.
- The compiler reserves the appropriate amount of memory based on the *type* of the elements and the *arraySize*.
- `array`s can be declared to contain values of most data types.

7.4 Examples Using `arrays`

- The following examples demonstrate how to declare arrays, how to initialize arrays and how to perform common array manipulations.

7.4.1 Declaring an `array` and Using a Loop to Initialize the `array`'s Elements

- The program in Fig. 7.3 declares five-element integer array `n` (line 10).
- `size_t` represents an unsigned integral type.
- This type is recommended for any variable that represents an `array`'s size or an `array`'s subscripts. Type `size_t` is defined in the `std` namespace and is in header `<cstddef>`, which is included by various other headers.
- If you attempt to compile a program that uses type `size_t` and receive errors indicating that it's not defined, simply include `<cstddef>` in your program.

```
1  // Fig. 7.3: fig07_03.cpp
2  // Initializing an array's elements to zeros and printing the array.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  using namespace std;
7
8  int main()
9  {
10    array< int, 5 > n; // n is an array of 5 int values
11
12    // initialize elements of array n to 0
13    for ( size_t i = 0; i < n.size(); ++i )
14       n[ i ] = 0; // set element at location i to 0
15
16    cout << "Element" << setw( 13 ) << "Value" << endl;
17
18    // output each array element's value
19    for ( size_t j = 0; j < n.size(); ++j )
20       cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
21 } // end main
```

Fig. 7.3 | Initializing an `array`'s elements to zeros and printing the `array`. (Part 1 of 2.)

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
```

Fig. 7.3 | Initializing an `array`'s elements to zeros and printing the `array`. (Part 2 of 2.)