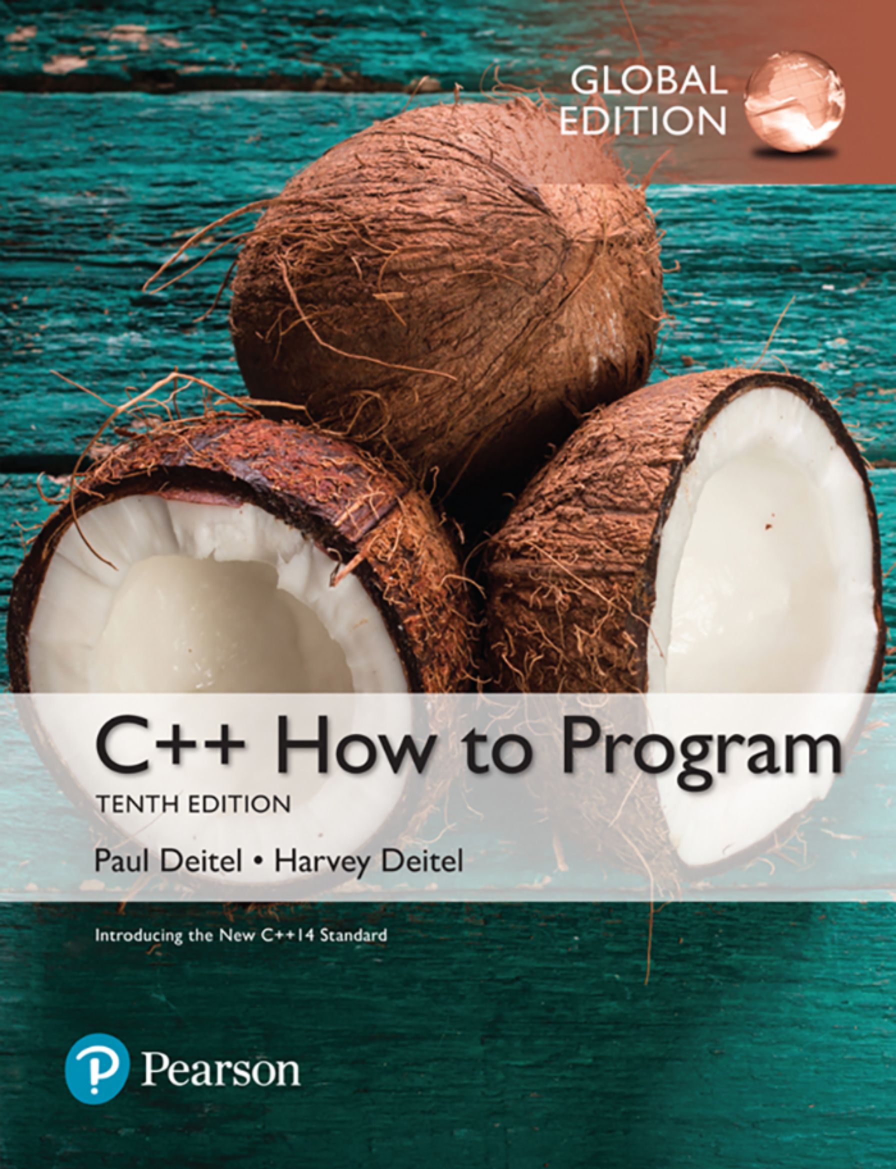

GLOBAL EDITION

C++ How to Program

TENTH EDITION

Paul Deitel • Harvey Deitel

Introducing the New C++14 Standard

Pearson

C++ HOW TO PROGRAM

6.61 ***(Computer-Assisted Instruction: Varying the Types of Problems)*** Modify the program of Exercise 6.60 to allow the user to pick a type of arithmetic problem to study. An option of `1` means addition problems only, `2` means subtraction problems only, `3` means multiplication problems only, `4` means division problems only and `5` means a random mixture of all these types.

Class Templates **array** and **vector**; Catching Exceptions

7

Objectives

In this chapter you'll:

- Use C++ Standard Library class template **array**—a fixed-size collection of related data items.
- Declare **array**s, initialize **array**s and refer to the elements of **array**s.
- Use **array**s to store, sort and search lists and tables of values.
- Use the range-based **for** statement.
- Pass **array**s to functions.
- Use C++ Standard Library function **sort** to arrange **array** elements in ascending order.
- Use C++ Standard Library function **binary_search** to locate an element in a sorted **array**.
- Declare and manipulate multidimensional **array**s.
- Use one- and two-dimensional **array**s to build a real-world **GradeBook** class.
- Use C++ Standard Library class template **vector**—a variable-size collection of related data items.

Outline

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*. To use them, you must include the <array> and <vector> headers, respectively.

After discussing how arrays are declared, created and initialized, we present examples that demonstrate several common array manipulations. We show how to *search* arrays to find particular elements and *sort* arrays to put their data in *order*.

We build two versions of an instructor GradeBook case study that use arrays to maintain sets of student grades *in memory* and analyze student grades. We introduce the *exception-handling* mechanism and use it to allow a program to continue executing when it attempts to access an array or vector element that does not exist.

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, we specify the name of the array and the **position number** of the particular element in the array.

Figure 7.1 shows an integer array called c that contains 12 **elements**. You refer to any one of these elements by giving the array name followed by the particular element's *position number* in square brackets ([]). 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 has **subscript 0** (**zero**) and is sometimes called the **zeroth element**. Thus, the elements of array c are c[0] (pronounced “c sub zero”), c[1], c[2] and so on. The highest subscript in array c is 11, which is 1 less than the number of elements in the array (12). array names follow the same conventions as other variable names.

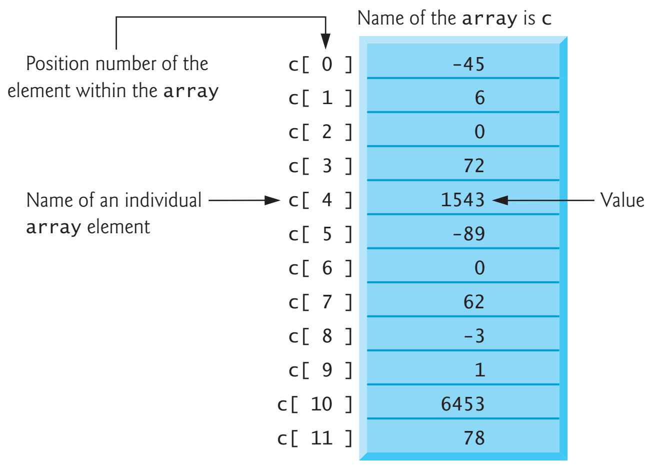

Fig. 7.1 | array of 12 elements.

A subscript must be an integer or integer expression (using any integral type). If a program uses an expression as a subscript, then the program evaluates the expression to determine the subscript. For example, if we assume that variable a is equal to 5 and that variable b is equal to 6, then the statement

```
c[a + b] += 2;
```

adds 2 to array element c[11]. A subscripted array name is an *lvalue*—it can be used on the left side of an assignment, just as non-array variable names can.

Let’s examine array c in Fig. 7.1 more closely. The **name** of the entire array is c. Each array *knows its own size*, which can be determined by calling its **size** member function as in c.size(). Its 12 elements are referred to as c[0] to c[11]. The **value** of c[0] is -45, the value of c[7] is 62 and the value of c[11] is 78. To print the sum of the values contained in the first three elements of array c, we’d write

```
cout << c[0] + c[1] + c[2] << endl;
```

To divide the value of c[6] by 2 and assign the result to the variable x, we’d write

```
x = c[6] / 2;
```

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 the subscript 6, while “array element 7” has the subscript 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]).*

The brackets that enclose a subscript are actually an *operator* that has the same precedence as parentheses used to call a function. Figure 7.2 shows the precedence and associativity of the operators introduced so far. The operators are shown top to bottom in decreasing order of precedence with their associativity and type.

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
`=` `+=` `-=` `*=` `/=` `%=`	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

arrays 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*. (Recall that a declaration which reserves memory is more specifically known as a *definition*.) The *arraySize* must be an unsigned integer. To tell the compiler to reserve 12 elements for integer array c, use the declaration

```
array<int, 12> c; // c is an array of 12 int values
```

arrays can be declared to contain values of most data types. For example, an array of type string can be used to store character strings.

7.4 Examples Using arrays

The following examples demonstrate how to declare, initialize and manipulate arrays.

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 9). Line 5 includes the `<array>` header, which contains the definition of class template `array`. Lines 12–14 use a `for` statement to initialize the `array` elements to zeros. Like other non-`static` local variables, `array`s are *not* implicitly initialized to zero (`static` `array`s are). The first output statement (line 16) displays the column headings for the columns printed in the subsequent `for` statement (lines 19–21), which prints the `array` in tabular format. Remember that `setw` specifies the field width in which only the *next* value is to be output.

```
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     array<int, 5> n; // n is an array of 5 int values
10
11    // initialize elements of array n to 0
12    for (size_t i{0}; i < n.size(); ++i) {
13       n[i] = 0; // set element at location i to 0
14    }
15
16    cout << "Element" << setw(10) << "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(10) << n[j] << endl;
21    }
22 }
```

```
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`.

In this program, the control variables `i` (line 12) and `j` (line 19) that specify `array` subscripts are declared to be of type **`size_t`**. According to the C++ standard `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 add `#include <cstddef>` to your program.