Topics :

* Learn about array
* Explore how to declare and manipulate data into arrays
* Learn about “array index out of bound”
* Become familiar with the restriction on array processing
* Discover how to pass an array as a parameter to a function
* Learn how to search array
* Learn how to sort array

a data type is called simple if variables of that type can store only one value

at a time. In contrast, in a structured data type, each data item is a collection of other

data items. Simple data types are building blocks of structured data types.

Example-8-1 (see files)

This program works fine. However, if you need to read and process 100 test scores,

you would have to declare 100 variables and write many cin, cout, and if statements.

Thus, for large amounts of data, this type of program is not efficient.

Note the following in the previous program:

1. Five variables must be declared because test scores less than the average test scores need to be printed.

2. All test scores are of the same data type, int.

3. The way in which these variables are declared indicates that the variables to store these numbers all have the same name—except the last character, which is a number.

4. All the if statements are similar, except the name of the variables to store the test scores.

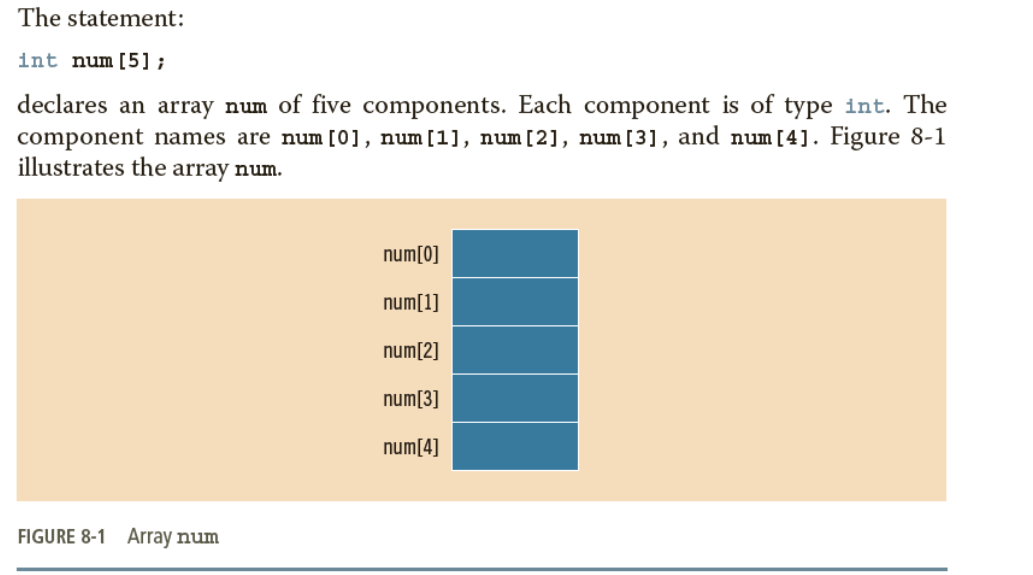
**Arrays**

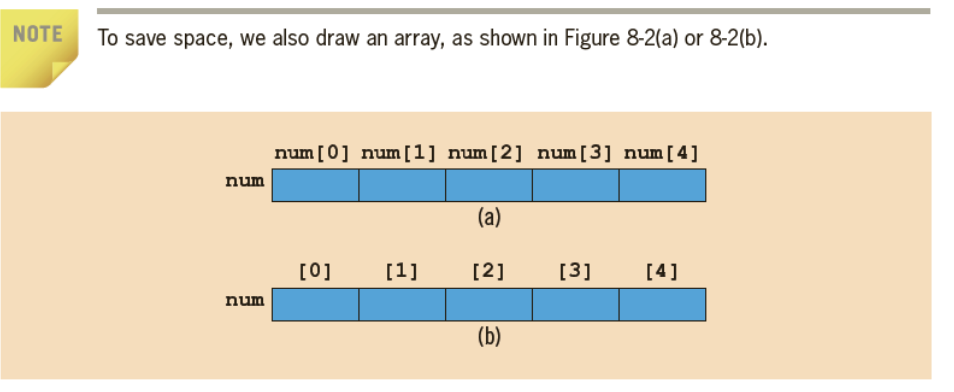
An array is a collection of a fixed number of components (also called elements) all of the same data type and in contiguous (that is, adjacent) memory space. A one-dimensional array is an array in which the components are arranged in a list form. In this section only one-demsional array is discussed, two dimensional array will be discussed alter on.

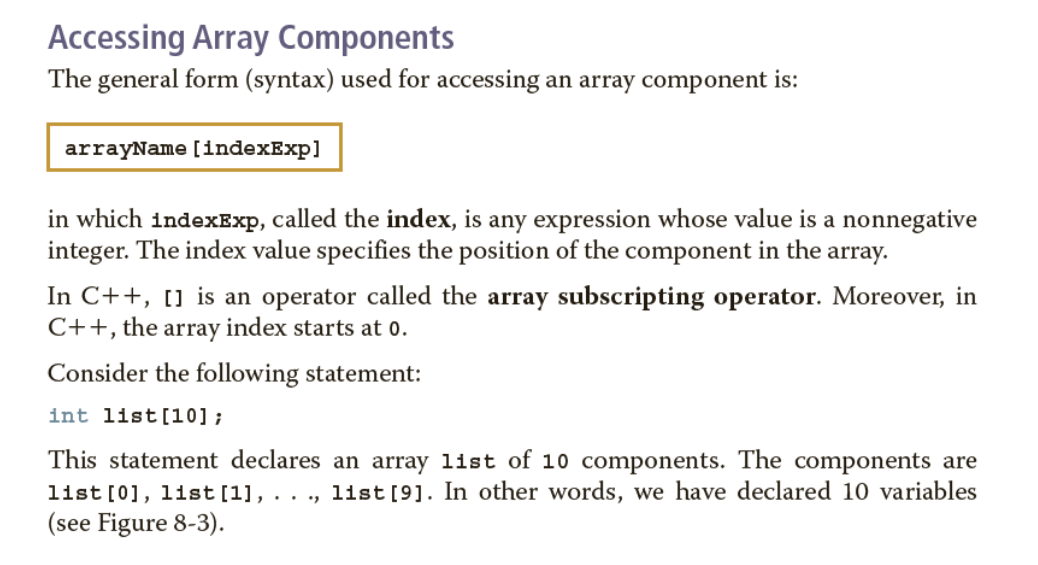
General form of declaration of arary:

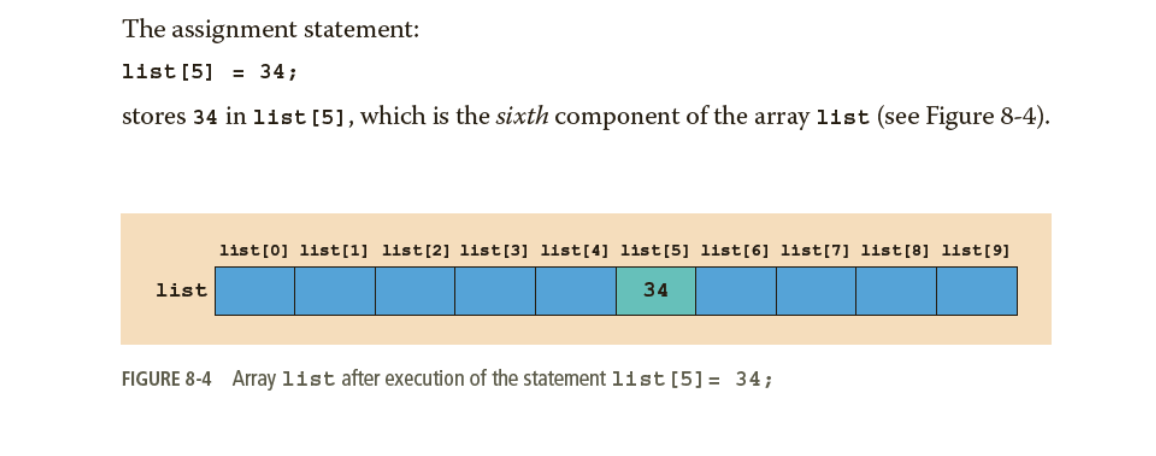
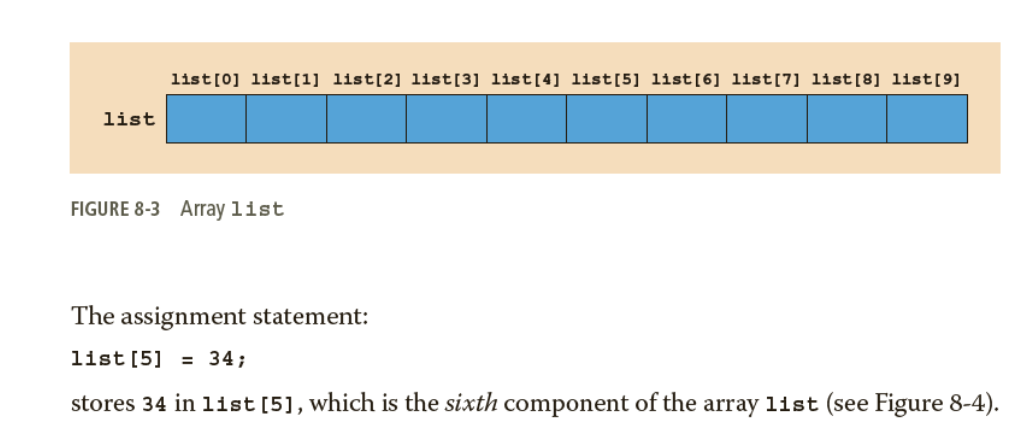
datatype arrayName [intExp]

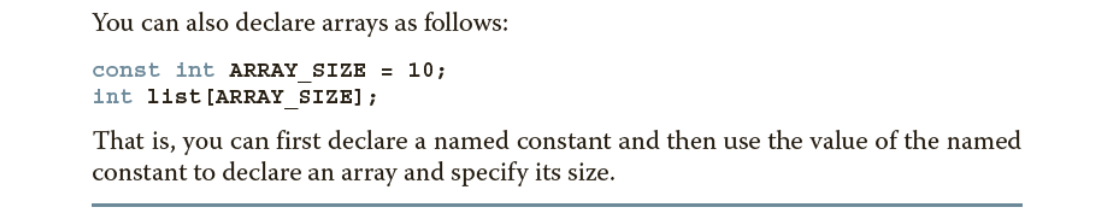
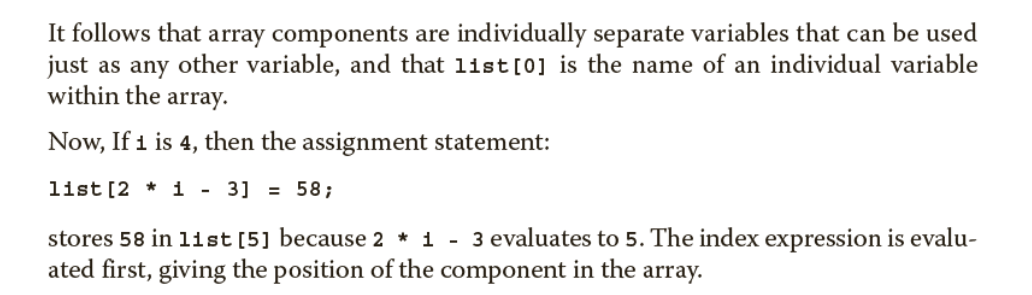
intExp specifies number components in the array and can be constant expression that evaluates to a positive numbers.

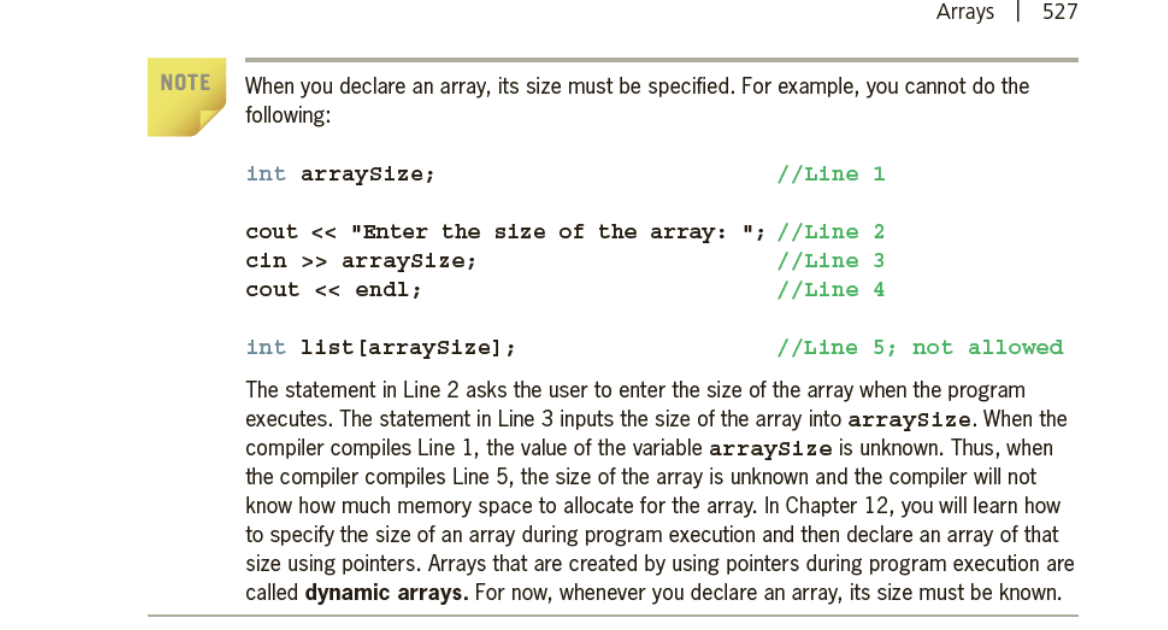


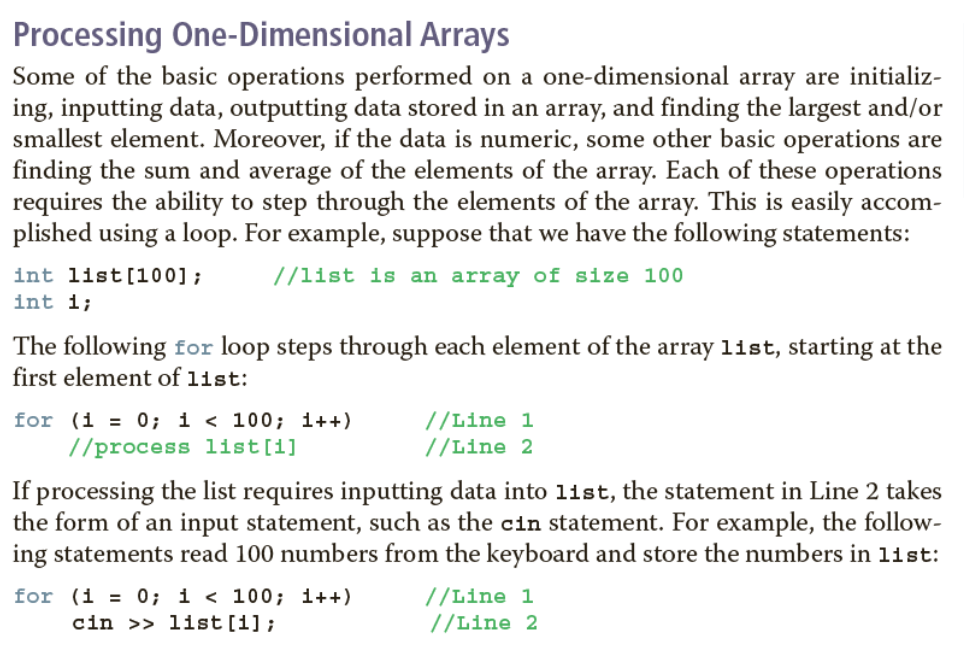












**double sales[10];**

**double largestSale, sum, average;**

The first statement declares an array **sales** of **10** components, with each component being of type **double**. The meaning of the other statements is clear.

a. **Initializing an array:** The following loop initializes every component of the array **sales** to **0.0**.

**for (int index = 0; index < 10; index++)**

**sales[index] = 0.0;**

b. **Reading data into an array:** The following loop inputs the data into

the array **sales**. For simplicity, we assume that the data is entered

from the keyboard.

**for (int index = 0; index < 10; index++)**

**cin >> sales[index];**

c. **Printing an array:** The following loop outputs the array **sales**. For

simplicity, we assume that the output goes to the screen.

**for (int index = 0; index < 10; index++)**

**cout << sales[index] << " ";**

d. **Finding the sum and average of an array:** Because the array **sales**,

as its name implies, represents certain sales data, it is natural to find

the total sale and average sale amounts. The following C11 code

finds the sum of the elements of the array **sales** and the average sale

amount:

**sum = 0; for (int index = 0; index < 10; index++) sum = sum + sales[index]; average = sum / 10**

**Array Index Out of Bounds:**

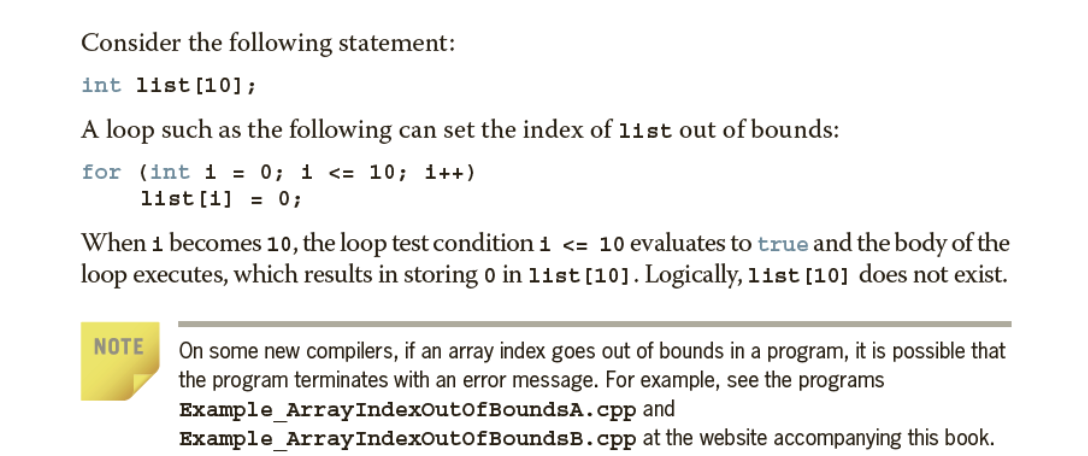
Consider the following declaration:

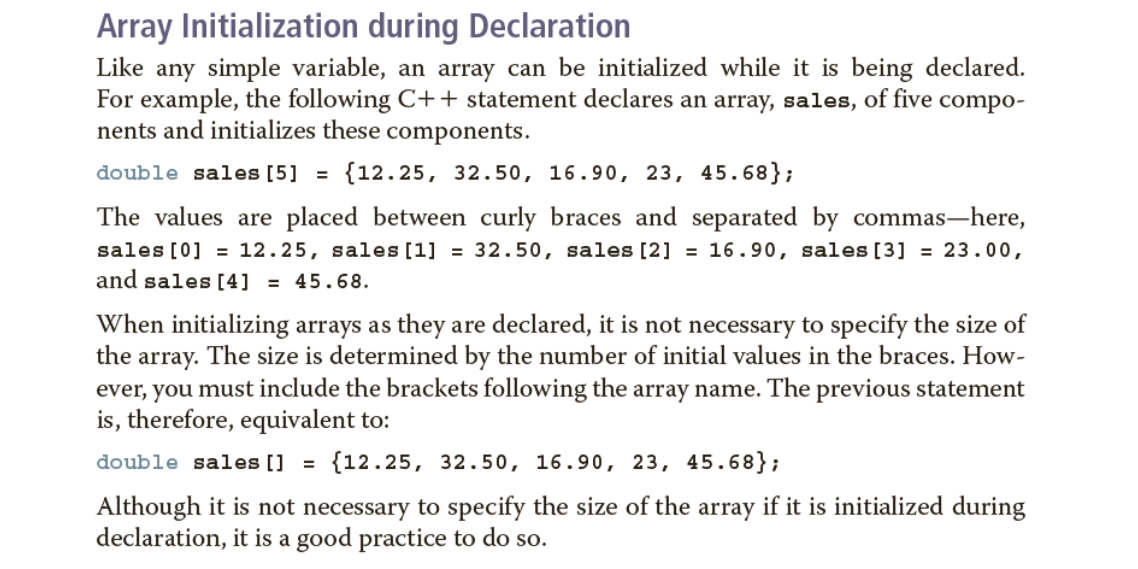
**double num[10]; int i;**

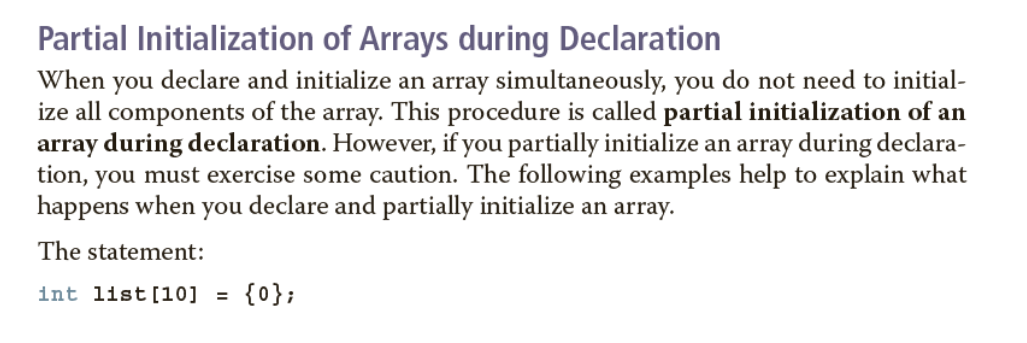
The component **num[i]** is *valid*, that is, **i** is a valid index if **i = 0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, or **9**. The index—say, **index**—of an array is **in bounds** if **index** is between **0** and

**ARRAY\_ SIZE - 1**, that is, **0 <= index <= ARRAY\_SIZE - 1**. If **index** is negative or **index** is greater than **ARRAY\_SIZE - 1**, then we say that the index is **out of bounds**.

Unfortunately, C11 does not check whether the index value is within range—that is, between **0** and **ARRAY\_SIZE - 1**. If the index goes out of bounds and the program tries to access the component specified by the index, then whatever memory location is indicated by the index that location is accessed. This situation can result in altering or accessing the data of a memory location that you never intended to modify or access, or in trying to access protected memory that causes the program to instantly halt. Consequently, several strange things can happen if the index goes out of bounds during execution. It is solely the programmer’s responsibility to make sure that the index is within bounds.







declares **list** to be an array of **10** components and initializes all of the components to **0**. The statement:

**int list[10] = {8, 5, 12};**

declares **list** to be an array of **10** components and initializes **list[0]** to **8**, **list[1]** to **5**, **list[2]** to **12**, and all other components to **0**. Thus, if all of the values are not specified in the initialization statement, the array components for which the values are not specified are initialized to **0**. Note that, here, the size of the array in the declaration statement does matter. For example, the statement:

**int list[] = {5, 6, 3};**

declares **list** to be an array of three components and initializes **list[0]** to **5**, **list[1]** to **6**, and **list[2]** to **3**. In contrast, the statement:

**int list[25] = {4, 7};**

declares **list** to be an array of **25** components. The first two components are initialized to **4** and **7**, respectively, and all other components are initialized to **0**.

Suppose that you have the following statement: **int x[5] = {};**. Then some compilers may initialize each element of the array **x** to **0**.

When you partially initialize an array, then all of the elements that follow the first uninitialized element must be uninitialized. Therefore, the following statement will result in a syntax error:

**int list[10] = {2, 5, 6, , 8}; //illegal**

In this initialization, because the fourth element is uninitialized, all elements that follow the fourth element must be left uninitialized.