CS1002 – Programming Fundamentals

Lecture # 16
Tuesday, October 18, 2022
FALL 2022
FAST – NUCES, Faisalabad Campus

Muhammad Yousaf

Arrays

Introduction

- A simple data type is one, if variables of that type can store only one value at a time
- A structured data type is one in which each data item is a collection of other data items

Example

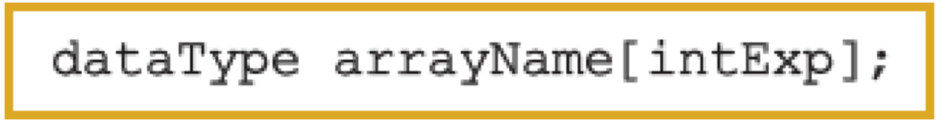
```
//Program that takes five numbers print their average
//and the numbers again
#include<iostream>
using namespace std;
int main(){
        int n1, n2, n3, n4, n5;
        double average;
        cout << "Enter five integers : " ;
        cin >> n1 >> n2 >> n3 >> n4 >> n5 ;

        average = (n1 + n2 + n3 + n4 + n5) / 5.0 ;

        cout << "The average of the given numbers = " << average ;
        cout << "\nand the numbers are n1 = " << n1 << " n2 = " << n2
                << " n3 = " << n3 << " n4 = " << n4
                << " n5 = " << n5 << endl ;
        return 0;
}
```

Example

- Five variables must be declared because the numbers are to be printed later
- All variables are of type int, that is, of the same data type
- 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

Arrays

- **Array:** A collection of a fixed number of components where all of the components have the same data type
- In a one-dimensional array, the components are arranged in a list form
- Syntax for declaring a one-dimensional array:



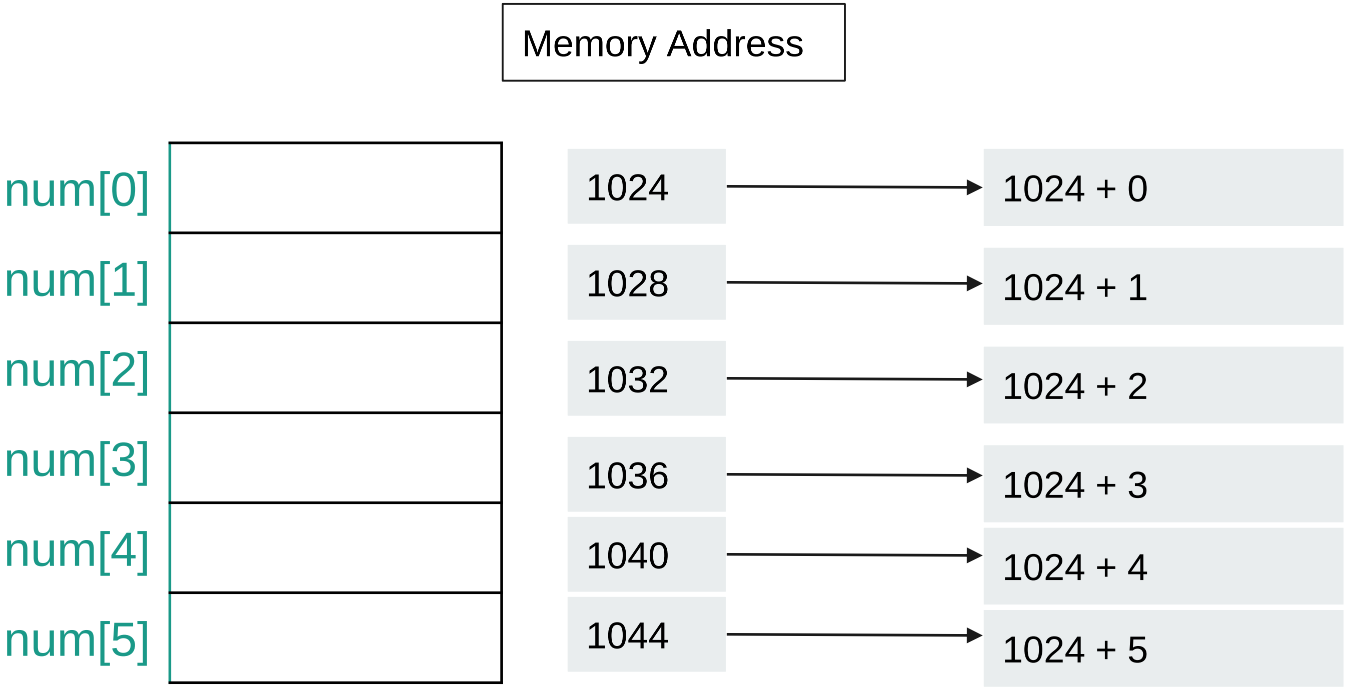
```
dataType arrayName[intExp];
```

  - **intExp** evaluates to a positive integer

Arrays

Example:

int num [6];



Memory Address

num[0]		1024	1024 + 0
num[1]		1028	1024 + 1
num[2]		1032	1024 + 2
num[3]		1036	1024 + 3
num[4]		1040	1024 + 4
num[5]		1044	1024 + 5

	[0]	[1]	[2]	[3]	[4]	[5]
num	22	30	38	45	50	99

Important!

-Array name **num** stores the memory address of first element called base address i.e. cout<<num; will print 1024.
-Array name **num** is not a simple variable but pointer variable. Pointer variable only stores address of another variable.
-Array name **num** is a constant pointer variable, whose address cannot be changed i.e. num = num+2 or num++ are invalid.

Defining Arrays

- When defining arrays, specify
  - Name
  - Type of array
  - Number of elements

    arrayType arrayName[numberOfElements];

  - Examples:

    int c[10];

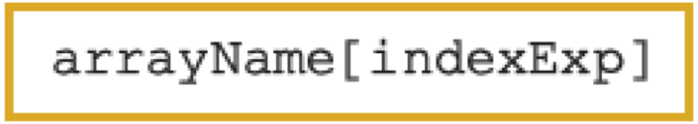
    float myArray[3284];

- Defining multiple arrays of same type
  - Format similar to regular variables
  - Example:

    int b[100], x[27];

Accessing Array Components

- General syntax:



```
arrayName[indexExp]
```

- Where indexExp, called an index, is any expression whose value is a nonnegative integer
- Index value specifies the position of the component in the array
- [] is the **array subscripting operator**
- The array index always starts at 0

Accessing Array Components (cont'd.)

```
int list[8];
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list								

```
list[5] = 75;
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list						75		

Accessing Array Components (cont'd.)

list[3] = 20 ;

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list				20				

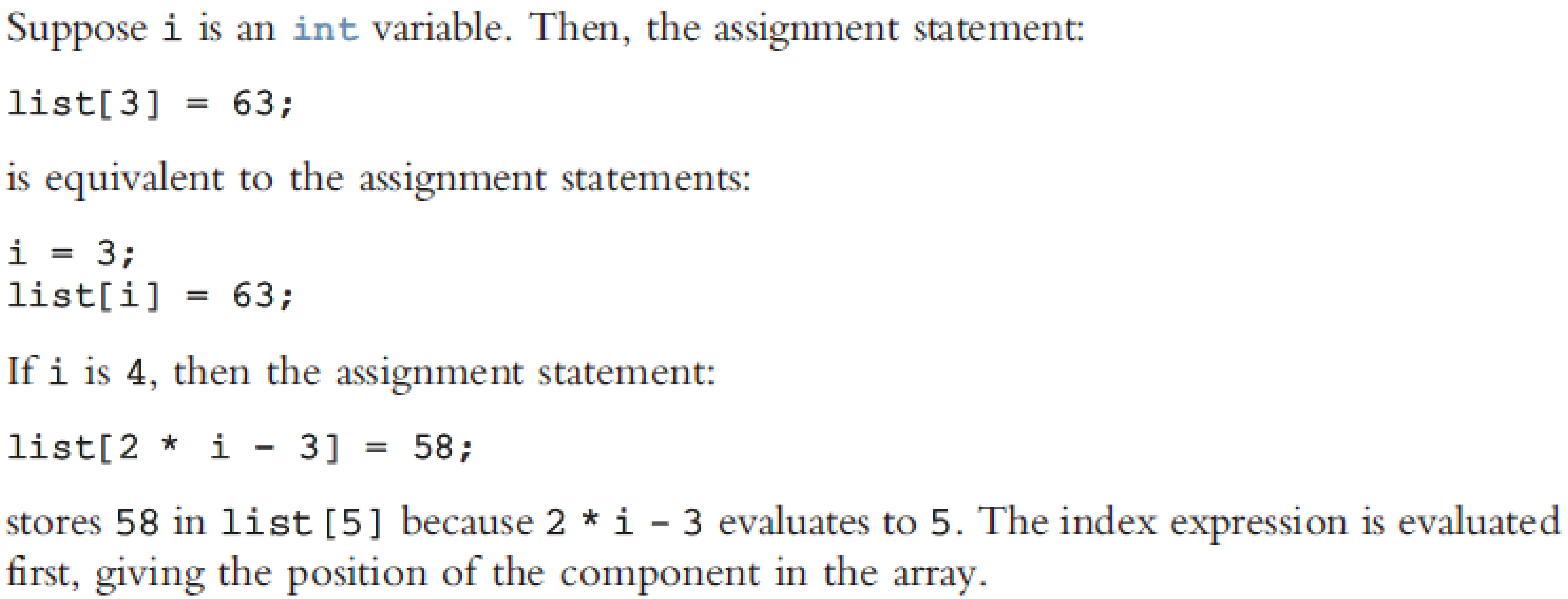
list[6]=100;

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list				20			100	

list[2]= list[3] + list[6];

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
list			120	20			100	

Accessing Array Components (cont'd.)



Suppose i is an int variable. Then, the assignment statement:

```
list[3] = 63;
```

is equivalent to the assignment statements:

```
i = 3;
list[i] = 63;
```

If i is 4, then the assignment statement:

```
list[2 * i - 3] = 58;
```

stores 58 in list[5] because 2 * i - 3 evaluates to 5. The index expression is evaluated first, giving the position of the component in the array.

Accessing Array Components (cont'd.)

Array can also be declared as

```
const int SIZE_OF_ARRAY = 20;
int array[SIZE_OF_ARRAY] ;
```

First declare a named constant and then use it to declare an array of this specific size.

When an array is declared its size must be known. You **cannot** do this:

```
int arr_size;
cout << "Enter size of array ";
cin >> arr_size;

int arr[arr_size];
```

Processing One-Dimensional Arrays

- Some basic operations performed on a one-dimensional array are:
  - **Initializing**
  - **Inputting** data
  - **Outputting** data stored in an array
  - **Finding** the largest and/or smallest element
- Each operation requires ability to **step through** the elements of the array
  - Easily accomplished by a **loop**

Processing One-Dimensional Arrays (cont'd.)

- Consider the declaration

```
        int list[100] = {0};  OR   int list[100] = {55}; OR   int list[100] =
{1,3};
        int i;
```

- Using for loops to access array elements:

```
        for (i = 0; i < 100; i++)   //Line 1
            //process list[i]              //Line 2
```

- Example:

```
        for (i = 0; i < 100; i++)   //Line 1
            cin >> list[i];                //Line 2
```

Processing One-Dimensional Arrays (cont'd.)

```
double scores[10];   //Declaring the array
int index;
double largest, sum, average;
```

Initializing an array

```
       for (index = 0 ; index < 10 ; ++index)
              scores[index] = 0.0 ;
```

Reading data into array

```
       for (index = 0 ; index < 10 ; ++index)
              cin >> scores[index] ;
```

Printing the array

```
       for (index = 0 ; index < 10 ; ++index)
              cout << scores[index] << “ ”;
```

Processing One-Dimensional Arrays (cont'd.)

Finding an element in an array

```
int index = -1, value;
cout << “Please enter the value to find : ” ;
cin >> value;
for (int i = 0 ; i < 10 ; ++i){
        if(values[i] == value)
                index = i;
}
if (index != -1)
        cout << “The value was found at index = ” << index ;
```

Processing One-Dimensional Arrays (cont'd.)

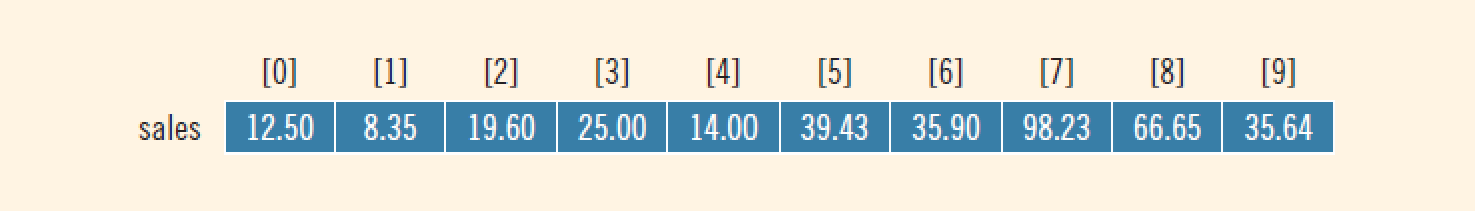
Finding sum and average of an array

```
sum = 0.0;
for (index = 0 ; index < 10 ; ++index)
        sum = sum + scores[index];
average = sum / 10;
```

Largest element in the array

```
maxIndex = 0;
for (index = 1 ; index < 10 ; ++index)
   if (scores[maxIndex] < scores[index])
            maxIndex = index;
largest = scores[maxIndex];
```

Processing One-Dimensional Arrays



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
sales	12.50	8.35	19.60	25.00	14.00	39.43	35.90	98.23	66.65	35.64

index	maxIndex	sales [maxIndex]	sales [index]	sales[maxIndex]< sales[index]
1	0	12.50	8.35	12.50 < 8.35 is false

2	0	12.50	19.60	12.50 < 19.60 is true;
				maxIndex = 2

3	2	19.60	25.00	19.60 < 25.00 is true;
				maxIndex = 3

4	3	25.00	14.00	25.00 < 14.00 is false

5	3	25.00	39.43	25.00 < 39.43 is true;
				maxIndex = 5

6	5	39.43	35.90	39.43 < 35.90 is false

7	5	39.43	98.23	39.43 < 98.23 is true;
				maxIndex = 7

8	7	98.23	66.65	98.23 < 66.65 is false
9	7	98.23	35.64	98.23 < 35.64 is false

After the for loop executes, maxIndex = 7, giving the index of the largest element in the array sales. Thus, largestSale = sales[maxIndex] = 98.23.

Does C++ allow -ve negative indexes for arrays?

Answer: No

Python allow -ve indexes for arrays! Yes

Arrays by an Example …

```
// C++ Program
// Initializing an array.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setw;

int main()
{
   int n[ 10 ];  // n is an array of 10 integers

   // initialize elements of array n to 0
   for ( int i = 0; i < 10; i++ )
      n[ i ] = 0;   // set element at location i to 0

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output contents of array n in tabular format
   for ( int j = 0; j < 10; j++ )
      cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;

   return 0;  // indicates successful termination

} // end main
```

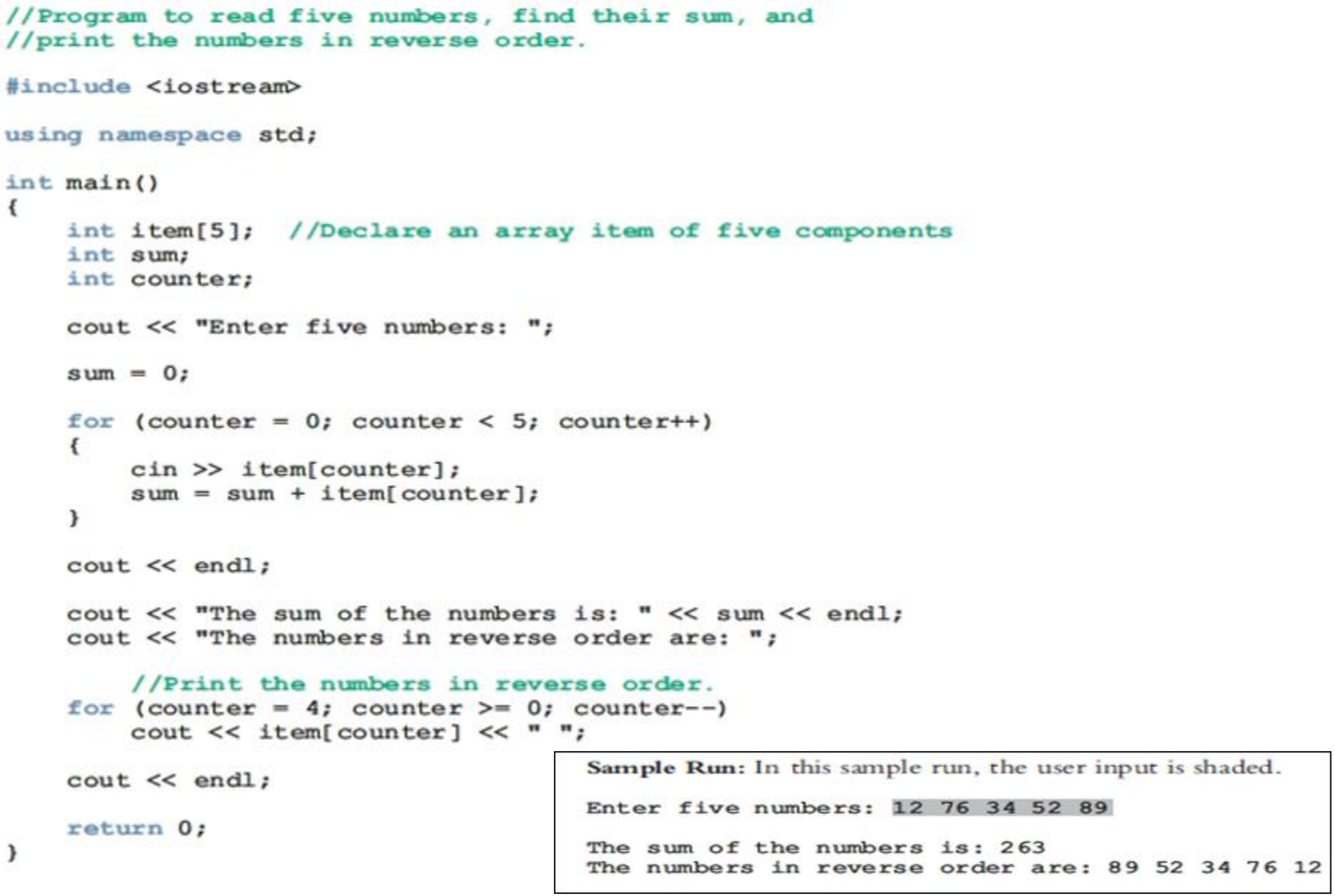
Declare a 10-element array of integers.

Initialize array to `0` using a for loop. Note that the array has elements **`n[0]`** to **`n[9]`**.

Output

```
Element      Value
   0           0
   1           0
   2           0
   3           0
   4           0
   5           0
   6           0
   7           0
   8           0
   9           0
```

Example



```
//Program to read five numbers, find their sum, and
//print the numbers in reverse order.

#include <iostream>

using namespace std;

int main()
{
    int item[5];  //Declare an array item of five components
    int sum;
    int counter;

    cout << "Enter five numbers: ";

    sum = 0;

    for (counter = 0; counter < 5; counter++)
    {
        cin >> item[counter];
        sum = sum + item[counter];
    }

    cout << endl;

    cout << "The sum of the numbers is: " << sum << endl;
    cout << "The numbers in reverse order are: ";

        //Print the numbers in reverse order.
    for (counter = 4; counter >= 0; counter--)
        cout << item[counter] << " ";

    cout << endl;

    return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Enter five numbers: 12 76 34 52 89

The sum of the numbers is: 263
The numbers in reverse order are: 89 52 34 76 12
```

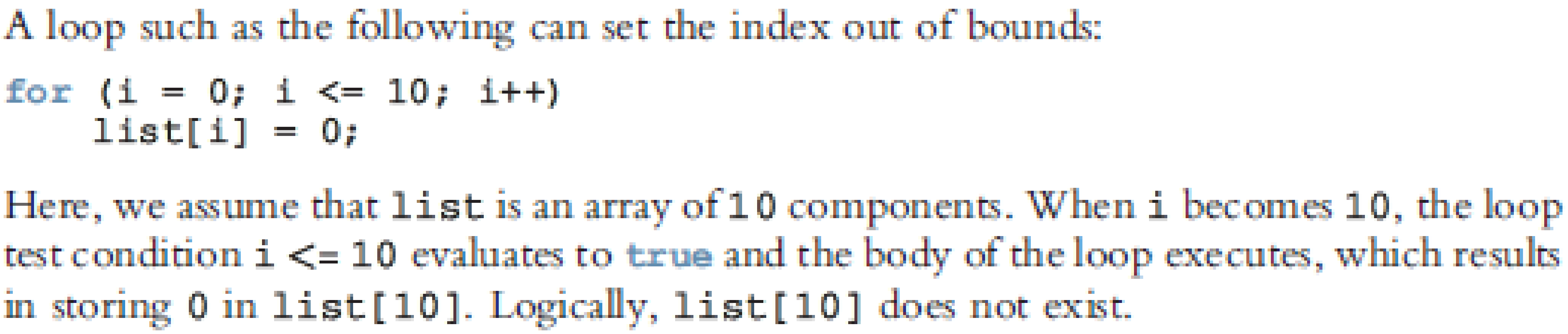
Array Index Out of Bounds

- If we have the statements:

  ```
  const ARRAY_SIZE = 10;
  double num[ARRAY_SIZE];
  int i;
  ```

- The component `num[i]` is valid if `i = 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9`
- The index of an array is in bounds if the `index >=0` and the `index <= ARRAY_SIZE-1`
- Otherwise, we say the index is out of bounds
- **In C++, there is no guard against indices that are out of bounds**

Array Index Out of Bounds



A loop such as the following can set the index out of bounds:

```
for (i = 0; i <= 10; i++)
    list[i] = 0;
```

Here, we assume that `list` is an array of `10` components. When `i` becomes `10`, the loop test condition `i <= 10` evaluates to `true` and the body of the loop executes, which results in storing `0` in `list[10]`. Logically, `list[10]` does not exist.

How array element is calculated

- Each element is placed at consecutive location
- We can calculate address of any element of array by performing simple arithmetic

  address_of_index_x = base_address + size_of_datatype * x

- Example

  int array[10] ;

  Assume this array starts at address 100 in memory. To calculate address of index 5.

  address_of_index_5 = 100 + 4*5 = 120

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
100 103	104 107	108 111	112 115	116 119	120 123	124 127	128 131	132 135	136 139

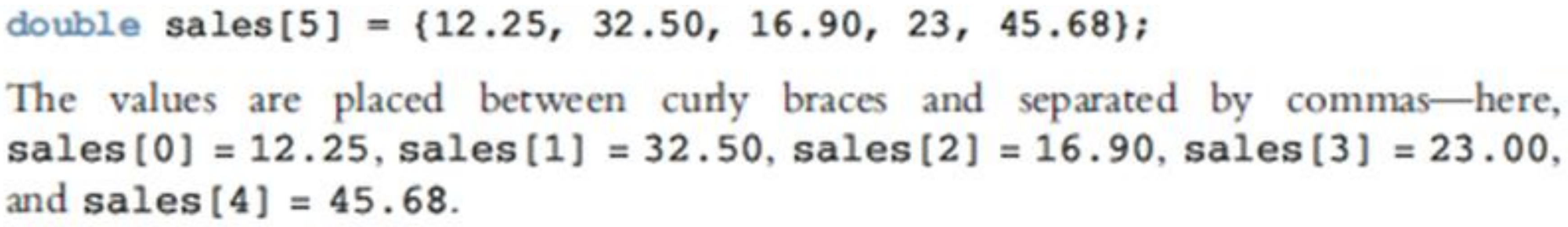
Note: Compiler just knows the address of the first element of the array known as the base address of the array. All other indexes are calculated relative to base address.

How array element is calculated

- Can we have –ve indexes in the array
- Like **array[-2]**

Array Initialization During Declaration

- Arrays can be initialized during declaration
- In this case, it is not necessary to specify the size of the array
- Size determined by the number of initial values in the braces
- Example:



```
double sales[5] = {12.25, 32.50, 16.90, 23, 45.68};
```

The values are placed between curly braces and separated by commas—here, `sales[0] = 12.25`, `sales[1] = 32.50`, `sales[2] = 16.90`, `sales[3] = 23.00`, and `sales[4] = 45.68`.

- double sales[] = {12.25, 32.50, 16.90, 23, 45.68};

Partial Initialization of Arrays During Declaration

- The statement:

```
int list[10] = {0};
```

declares list to be an array of 10 components and initializes all of them to zero

- The statement:

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

declares list to be an array of 10 components, initializes list[0] to 8, list[1] to 5, list[2] to 12 and all other components are initialized to 0

Partial Initialization of Arrays During Declaration (cont'd.)

- The statement:

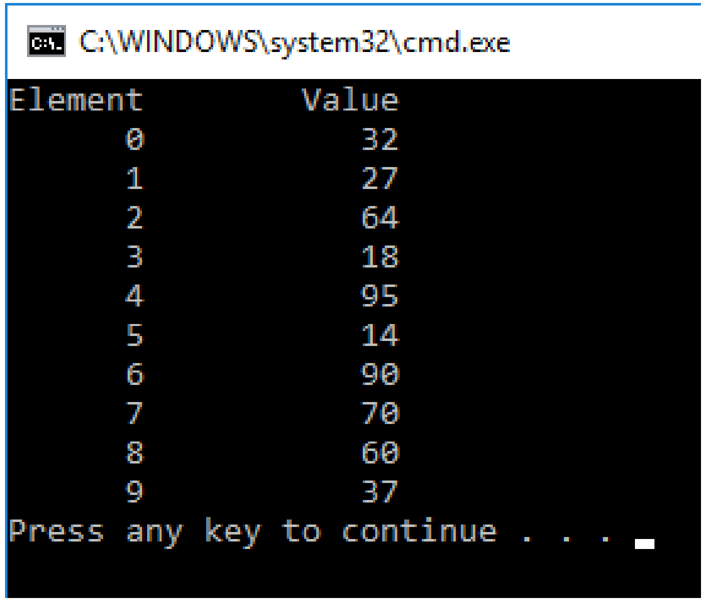
```
int list[] = {5, 6, 3};
```

declares list to be an array of 3 components and initializes list[0] to 5, list[1] to 6, and list[2] to 3

Partial Initialization of Arrays During Declaration (cont'd.)

- 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

Arrays by an Example II …



```
Element        Value
      0           32
      1           27
      2           64
      3           18
      4           95
      5           14
      6           90
      7           70
      8           60
      9           37
Press any key to continue . . .
```

```
 // C++ Program
 // Initializing an array with a declaration.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <iomanip>

using std::setw;

int main()
{
   // use initializer list to initialize array n
   int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output contents of array n in tabular format
   for ( int i = 0; i < 10; i++ )
      cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;

   return 0;  // indicates successful termination

} // end main
```

Note the use of the initializer list.

Some Restrictions on Array Processing

- Consider the following statements:

```
int myList[5] = {0, 4, 8, 12, 16};//Line 1
int yourList[5];           //Line 2
```

- C++ does not allow aggregate operations on an array:

```
youtList = myList ; //illegal
```

- Solution:

```
for(int i = 0 ; i < 5 ; ++i)
       yourList[i] = myList[i] ;
```

Some Restrictions on Array Processing (cont'd.)

- The following is illegal too:

```
cin >> yourList ; //illegal
```

- Solution:

```
for(int i=0 ; i<5 ; ++i)
        cin >> yourList[i] ;
```

- The following statements are legal, but do not give the desired results:

```
cout << yourList ;
if(myList <= yourList)
        .
        .
        .
```

Examples Using Arrays

- **Array size**
  - Can be specified with constant variable (const)

    ```
    const int SIZE = 20;
    ```

  - Constants cannot be changed
  - Constants must be initialized when declared
  - Also called named constants or read-only variables

Arrays by an Example - III …

```
// C++ Program
// Initialize array s to the even integers from 2 to 20.
#include <iostream>

using namespace std;
#include <iomanip>


int main()
{
   // constant variable can be used to specify array size
   const int ARRAYSIZE = 10;

   int s[ARRAYSIZE ];  // array s has 10 elements

   for ( int i = 0; i < ARRAYSIZE; i++ )  // set the values
      s[ i ] = 2 + 2 * i;

   cout << "Element" << setw( 13 ) << "Value" << endl;

   // output contents of array s in tabular format
   for ( int j = 0; j < ARRAYSIZE; j++ )
      cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;

   return 0;  // indicates successful termination

} // end main
```

Note use of **`const`** keyword. Only **`const`** variables can specify array sizes.

The program becomes more scalable when we set the array size using a **`const`** variable. We can change **`arraySize`**, and all the loops will still work (otherwise, we’d have to update every loop in the program).

Program Output …

```
Element      Value
   0          2
   1          4
   2          6
   3          8
   4         10
   5         12
   6         14
   7         16
   8         18
   9         20
```

Arrays by an Example -IV…

```cpp
// C++ Program
// Compute the sum of the elements of the array.
#include <iostream>

using std::cout;
using std::endl;

int main()
{
  const int ARRAYSIZE = 10;

  int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

  int total = 0;

  // sum contents of array a
  for ( int i = 0; i < ARRAYSIZE; i++ )
     total += a[ i ];

  cout << "Total of array element values is " << total << endl;

  return 0;  // indicates successful termination

} // end main
```

Output:
Total of array element values is 55

Arrays

```
// C++ Program
// Roll a six-sided die 6000 times.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setw;

#include <cstdlib>
#include <ctime>

int main()
{
   const int ARRAY_SIZE= 7;
   int frequency[ARRAY_SIZE ] = { 0 };

   srand( time( 0 ) );  // seed random-number generator

   // roll dice 6000 times
```

An array is used instead of 6 regular variables, and the proper element can be updated

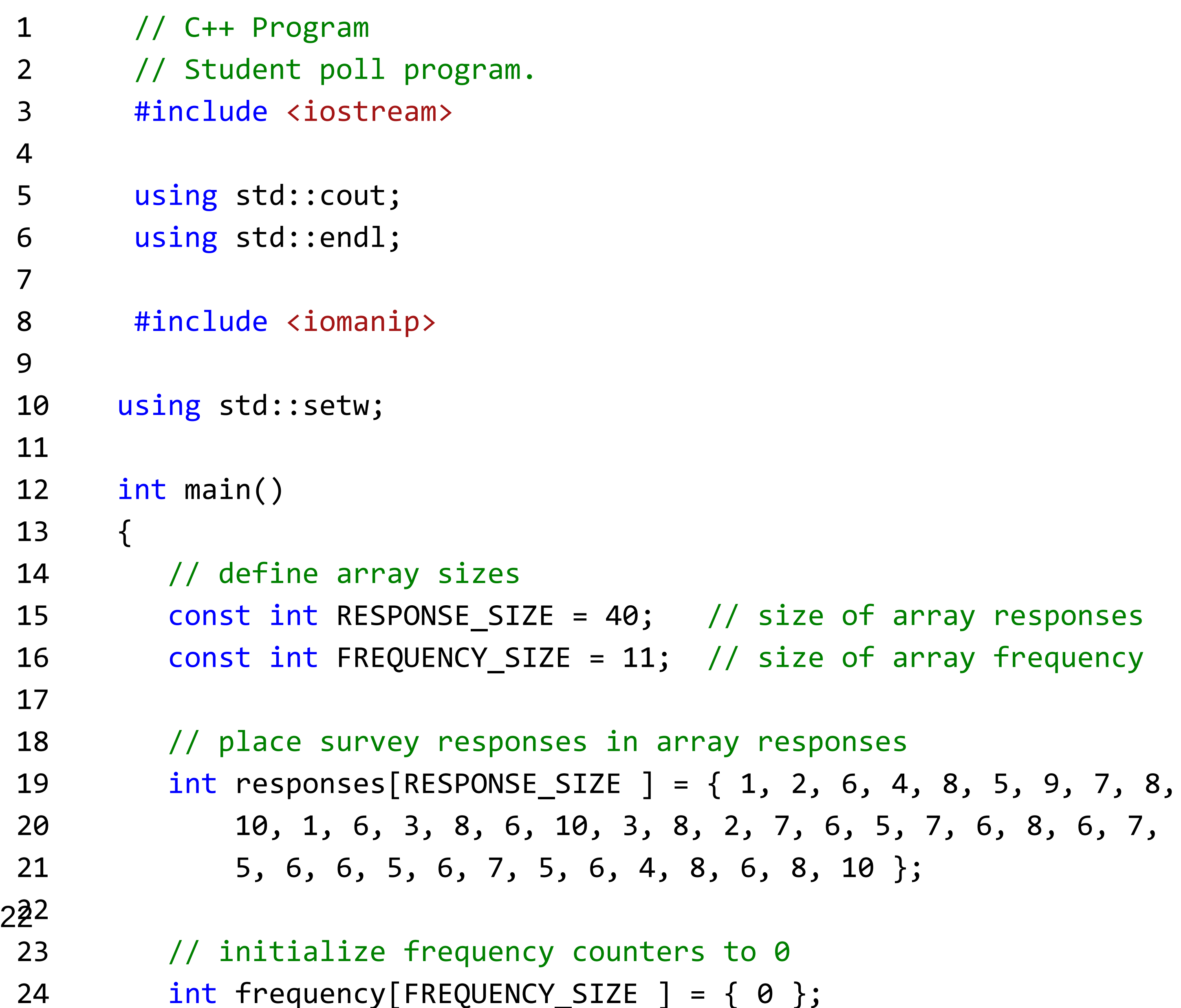
This creates a number between 1 and 6, which determines the index of **`frequency[]`** that should be incremented.

```cpp

   cout << "Face" << setw( 13 ) << "Frequency" << endl;

   // output frequency elements 1-6 in tabular format
   for ( int face = 1; face < ARRAY_SIZE; face++ )
      cout << setw( 4 ) << face
           << setw( 13 ) << frequency[ face ] << endl;

   return 0;  // indicates successful termination

} // end main
```

Program Output …

```
Face    Frequency
   1         1003
   2         1004
   3          999
   4          980
   5         1013
   6         1001
```

Arrays by an Example – VII



```
 // C++ Program
 // Student poll program.
 #include <iostream>

 using std::cout;
 using std::endl;

 #include <iomanip>

using std::setw;

int main()
{
   // define array sizes
   const int RESPONSE_SIZE = 40;   // size of array responses
   const int FREQUENCY_SIZE = 11;  // size of array frequency

   // place survey responses in array responses
   int responses[RESPONSE_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
       10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
       5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };

   // initialize frequency counters to 0
   int frequency[FREQUENCY_SIZE ] = { 0 };
   // for each answer, select value of an element of array
   // responses and use that value as subscript in array
   // frequency to determine element to increment
   for ( int answer = 0; answer < RESPONSE_SIZE; answer++ )
      ++frequency[ responses[answer] ];

   // display results
   cout << "Rating" << setw( 17 ) << "Frequency" << endl;

   // output frequencies in tabular format
   for ( int rating = 1; rating < FREQUENCY_SIZE ; rating++ )
      cout << setw( 6 ) << rating
           << setw( 17 ) << frequency[ rating ] << endl;

   return 0;  // indicates successful terminat

} // end main
```

`responses[answer]` is the rating (from 1 to 10). This determines the index in **`frequency[]`** to increment.

Program Output …

```
Rating       Frequency
    1            2
    2            2
    3            2
    4            2
    5            5
    6           11
    7            5
    8            7
    9            1
   10             3
```

Questions