



## EE1003 Introduction to Computer I



```
111001100111001000000|11001001110010011001100111001000000
00110010101110010■001100101011100100110010101110010■01
10000011000101110001000001100001011100000110000101110000
```

# Chapter 6

## Arrays and Vectors

Andy, Yu-Guang Chen

Assistant Professor, Department of EE

National Central University

[andygchen@ee.ncu.edu.tw](mailto:andygchen@ee.ncu.edu.tw)



2021/10/26

Andy Yu-Guang Chen

1



## Learning Objectives



In this chapter you'll learn:

- To use the array data structure to represent a set of related data items.
- To use arrays to store, sort and search lists and tables of values.
- To declare arrays, initialize arrays and refer to the individual elements of arrays.
- To pass arrays to functions.
- Basic searching and sorting techniques.
- To declare and manipulate multidimensional arrays.
- To use C++ Standard Library class template `vector`.
- To use C++ Standard Library class `string`.



2021/10/26

Andy Yu-Guang Chen

2



# Outline

- 6.1** Introduction
- 6.2** Arrays
- 6.3** Declaring Arrays
- 6.4** Examples Using Arrays
  - 6.4.1 Declaring an Array and Using a Loop to Initialize the Array's Elements
  - 6.4.2 Initializing an Array in a Declaration with an Initializer List
  - 6.4.3 Specifying an Array's Size with a Constant Variable and Setting Array Elements with Calculations
  - 6.4.4 Summing the Elements of an Array
  - 6.4.5 Using Bar Charts to Display Array Data Graphically
  - 6.4.6 Using the Elements of an Array as Counters
  - 6.4.7 Using Arrays to Summarize Survey Results
  - 6.4.8 Static Local Arrays and Automatic Local Arrays
- 6.5** Passing Arrays to Functions
- 6.6** Searching Arrays with Linear Search
- 6.7** Sorting Arrays with Insertion Sort
- 6.8** Multidimensional Arrays
- 6.9** Case Study: Processing Grades in a Two-Dimensional Array
- 6.10** Introduction to C++ Standard Library Class Template `vector`
- 6.11** Standard Library Class `string`
- 6.12** Wrap-Up



2021/10/26

Andy Yu-Guang Chen

3



## 6.1 Introduction

- ◆ This chapter introduces the important topic of **data structures**—collections of related data items.
- ◆ **Arrays** are data structures consisting of related data items of the same type.
- ◆ After discussing how arrays are declared, created and initialized, we present a series of practical examples that demonstrate several common array manipulations.



2021/10/26

Andy Yu-Guang Chen

4



## 6.2 Arrays

- ◆ An array is a consecutive 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 that element.
- ◆ The position number is formally called a **subscript** or **index**.
  - This number specifies the number of elements from the beginning of the array.
  - A subscript must be an integer or integer expression (using any integral type).
- ◆ 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).



2021/10/26

Andy Yu-Guang Chen

5



## 6.2 Arrays

Position number of the element within the array	Name of the array is c	
	c[ 0 ]	-45
	c[ 1 ]	6
	c[ 2 ]	0
	c[ 3 ]	72
Name of an individual array element →		c[ 4 ]
		1543 ← Value
	c[ 5 ]	-89
	c[ 6 ]	0
	c[ 7 ]	62
	c[ 8 ]	-3
	c[ 9 ]	1
	c[ 10 ]	6453
	c[ 11 ]	78

**Fig. 6.1** | Array of 12 elements.



2021/10/26

Andy Yu-Guang Chen

6



## 6.2 Arrays



### Common Programming Error 6.1

Note the difference between the “seventh element of the array” and “array element 7.” Array 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. Unfortunately, this distinction frequently is a 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]`).



2021/10/26

Andy Yu-Guang Chen

7



## 6.3 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:
  - `type arrayName [ arraySize ];`
  - Ex: `int n[5];` → 5 elements
- ◆ The compiler reserves the appropriate amount of memory.
  - All elements are put in a continuous memory space.



2021/10/26

Andy Yu-Guang Chen

8



## 6.4 Examples Using Arrays

- ◆ Following we will take a look for some examples about using arrays.



2021/10/26

Andy Yu-Guang Chen

9



### 6.4.1 Declaring an Array and Using a Loop to Initialize the Array's Elements

- ◆ The program in Fig. 6.3 declares 10-element integer array `n`.

---

```

1 // Fig. 6.3: fig06_03.cpp
2 // Initializing an array.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     int n[ 10 ]; // n is an array of 10 integers
10
11    // initialize elements of array n to 0
12    for ( int i = 0; i < 10; i++ )
13        n[ i ] = 0; // set element at location i to 0
14
15    cout << "Element" << setw( 13 ) << "Value" << endl;
16
17    // output each array element's value
18    for ( int j = 0; j < 10; j++ )
19        cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
20 } // end main

```

---



**Fig. 6.3** | Initializing an array's elements to zeros and printing the array. (Part I of 2.)

2021/10/26

Andy Yu-Guang Chen

10

## 6.4.1 Declaring an Array and Using a Loop to Initialize the Array's Elements

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

**Fig. 6.3** | Initializing an array's elements to zeros and printing the array. (Part 2 of 2.)



2021/10/26

Andy Yu-Guang Chen

11

## 6.4.2 Initializing an Array in a Declaration with an Initializer List

- ◆ The elements of an array also can be initialized by a brace-delimited comma-separated list of **initializers**.
- ◆ The program in Fig. 6.4 uses an **initializer list** to initialize an integer array with 10 values (line 10).
- ◆ If there are fewer initializers than elements in the array, the remaining array elements are initialized to zero.
- ◆ If the array size is omitted from a declaration with an initializer list, the compiler determines the array size automatically
  - By counting the number of elements in the initializer list.
  - Ex: `int n[] = {1,2,3,4,5};` → 5 elements



2021/10/26

Andy Yu-Guang Chen

12



## 6.4.2 Initializing an Array in a Declaration with an Initializer List

---

```

1 // Fig. 6.4: fig06_04.cpp
2 // Initializing an array in a declaration.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     // use initializer list to initialize array n
10    int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
11
12    cout << "Element" << setw( 13 ) << "Value" << endl;
13
14    // output each array element's value
15    for ( int i = 0; i < 10; i++ )
16        cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
17 } // end main

```

---

Element	Value
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37



2021/10/26

Andy Yu-Guang Chen

13



## 6.4.3 Specifying an Array's Size with a Constant Variable and Setting Array Elements with Calculations

- ◆ Figure 6.5 sets the elements of a 10-element array **s** to the even integers 2, 4, 6, ..., 20 (lines 14–15).
- ◆ Line 10 uses the **const qualifier** to declare a so-called **constant variable arraySize** with the value 10.
- ◆ Constant variables must be initialized with a constant expression when they're declared and cannot be modified thereafter.
- ◆ Constant variables are also called **named constants** or **read-only variables**.



2021/10/26

Andy Yu-Guang Chen

14



### 6.4.3 Specifying an Array's Size with a Constant Variable and Setting Array Elements with Calculations

```

1 // Fig. 6.5: fig06_05.cpp
2 // Set array s to the even integers from 2 to 20.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     // constant variable can be used to specify array size
10    const int arraySize = 10;
11
12    int s[ arraySize ]; // array s has 10 elements
13
14    for ( int i = 0; i < arraySize; i++ ) // set the values
15        s[ i ] = 2 + 2 * i;
16
17    cout << "Element" << setw( 13 ) << "Value" << endl;
18
19    // output contents of array s in tabular format
20    for ( int j = 0; j < arraySize; j++ )
21        cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
22 } // end main

```

**Fig. 6.5** | Generating values to be placed into elements of an array. (Part 1 of 2.)



2021/10/26

Andy Yu-Guang Chen

15



### 6.4.3 Specifying an Array's Size with a Constant Variable and Setting Array Elements with Calculations

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

**Fig. 6.5** | Generating values to be placed into elements of an array. (Part 2 of 2.)



2021/10/26

Andy Yu-Guang Chen

16



### 6.4.3 Specifying an Array's Size with a Constant Variable and Setting Array Elements with Calculations



```

1 // Fig. 6.7: fig06_07.cpp
2 // A const variable must be initialized.
3
4 int main()
5 {
6     const int x; // Error: x must be initialized
7
8     x = 7; // Error: cannot modify a const variable
9 } // end main

```

*Microsoft Visual C++ compiler error message:*

```
C:\cpphtp6_examples\ch06\fig06_07.cpp(6) : error C2734: 'x' : const object
must be initialized if not extern
C:\cpphtp6_examples\ch06\fig06_07.cpp(8) : error C3892: 'x' : you cannot
assign to a variable that is const
```

*GNU C++ compiler error message:*

```
fig06_07.cpp:6: error: uninitialized const 'x'
fig06_07.cpp:8: error: assignment of read-only variable 'x'
```



**Fig. 6.7** | const variables must be initialized.

2021/10/26

Andy Yu-Guang Chen

17



### 6.4.3 Specifying an Array's Size with a Constant Variable and Setting Array Elements with Calculations



#### Common Programming Error 6.5

*Only constants can be used to declare the size of automatic and static arrays. Not using a constant for this purpose is a compilation error.*



#### Good Programming Practice 6.2

*Defining the size of an array as a constant variable instead of a literal constant makes programs clearer. This technique eliminates so-called **magic numbers**. For example, repeatedly mentioning the size 10 in array-processing code for a 10-element array gives the number 10 an artificial significance and can be confusing when the program includes other 10s that have nothing to do with the array size.*



2021/10/26

Andy Yu-Guang Chen

18



## 6.4.4 Summing the Elements of an Array

- ◆ The program in Fig. 6.8 sums the values contained in the 10-element integer array **a**.

---

```

1 // Fig. 6.8: fig06_08.cpp
2 // Compute the sum of the elements of the array.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     const int arraySize = 10; // constant variable indicating size of array
9     int a[ arraySize ] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
10    int total = 0;
11
12    // sum contents of array a
13    for ( int i = 0; i < arraySize; i++ )
14        total += a[ i ];
15
16    cout << "Total of array elements: " << total << endl;
17 } // end main

```

Total of array elements: 849



**Fig. 6.8** | Computing the sum of the elements of an array.

2021/10/26

Andy Yu-Guang Chen

19



## 6.4.5 Using Bar Charts to Display Array Data Graphically

- ◆ Our next program (Fig. 6.9) stores grade distribution data in an array of 11 elements and displays a bar of (\*) for each element.

---

```

1 // Fig. 6.9: fig06_09.cpp
2 // Bar chart printing program.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     const int arraySize = 11;
10    int n[ arraySize ] = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
11
12    cout << "Grade distribution:" << endl;
13

```



**Fig. 6.9** | Bar chart printing program. (Part 1 of 3.)

2021/10/26

Andy Yu-Guang Chen

20



## 6.4.5 Using Bar Charts to Display Array Data Graphically



```

14 // for each element of array n, output a bar of the chart
15 for ( int i = 0; i < arraySize; i++ )
16 {
17     // output bar labels ("0-9:", ..., "90-99:", "100:")
18     if ( i == 0 )
19         cout << " 0-9: ";
20     else if ( i == 10 )
21         cout << " 100: ";
22     else
23         cout << i * 10 << "-" << ( i * 10 ) + 9 << ": ";
24
25     // print bar of asterisks
26     for ( int stars = 0; stars < n[ i ]; stars++ )
27         cout << '*';
28
29     cout << endl; // start a new line of output
30 } // end outer for
31 } // end main

```

**Fig. 6.9** | Bar chart printing program. (Part 2 of 3.)



2021/10/26

Andy Yu-Guang Chen

21



## 6.4.5 Using Bar Charts to Display Array Data Graphically



```

Grade distribution:
 0-9:
 10-19:
 20-29:
 30-39:
 40-49:
 50-59:
 60-69: *
 70-79: **
 80-89: ****
 90-99: **
 100: *

```

**Fig. 6.9** | Bar chart printing program. (Part 3 of 3.)



2021/10/26

Andy Yu-Guang Chen

22



## 6.4.6 Using the Elements of an Array as Counters



- ◆ Sometimes, programs use counter variables to summarize data, such as the results of a survey.
- ◆ In previous die-rolling program, we used separate counters to track the number of occurrences of each side of a die.
- ◆ An array version of this program is shown in Fig. 6.10.
- ◆ The single statement in line 18 of this program replaces the switch statement in lines 25–47 of Fig. 6.9.



2021/10/26

Andy Yu-Guang Chen

23



## 6.4.6 Using the Elements of an Array as Counters



```

1 // Fig. 6.10: fig06_10.cpp
2 // Roll a six-sided die 6,000,000 times.
3 #include <iostream>
4 #include <iomanip>
5 #include <cstdlib>
6 #include <ctime>
7 using namespace std;
8
9 int main()
10 {
11     const int arraySize = 7; // ignore element zero
12     int frequency[ arraySize ] = {}; // initialize elements to 0
13
14     srand( time( 0 ) ); // seed random number generator
15
16     // roll die 6,000,000 times; use die value as frequency index
17     for ( int roll = 1; roll <= 6000000; roll++ )
18         frequency[ 1 + rand() % 6 ]++;
19
20     cout << "Face" << setw( 13 ) << "Frequency" << endl;
21

```

**Fig. 6.10** | Die-rolling program using an array instead of switch. (Part I of 2.)

2021/10/26

Andy Yu-Guang Chen

24



## 6.4.6 Using the Elements of an Array as Counters

---

```

22     // output each array element's value
23     for ( int face = 1; face < arraySize; face++ )
24         cout << setw( 4 ) << face << setw( 13 ) << frequency[ face ]
25             << endl;
26 } // end main

```

Face	Frequency
1	1000167
2	1000149
3	1000152
4	998748
5	999626
6	1001158



**Fig. 6.10** | Die-rolling program using an array instead of switch. (Part 2 of 2.)



2021/10/26

Andy Yu-Guang Chen

25



## 6.4.7 Using Arrays to Summarize Survey Results

- ◆ Our next example (Fig. 6.11) uses arrays to summarize the results of data collected in a survey.
  - Forty students were asked to rate the quality of the food in the student cafeteria on a scale of 1 to 10.
  - Place the 40 responses in an integer array and summarize the results of the poll.
- ◆ C++ has no array bounds checking to prevent the computer from referring to an element that does not exist.
  - An executing program can “walk off” either end of an array without warning.
- ◆ You should ensure that all array references remain within the bounds of the array.



2021/10/26

Andy Yu-Guang Chen

26



## 6.4.7 Using Arrays to Summarize Survey Results



```

1 // Fig. 6.11: fig06_11.cpp
2 // Poll analysis program.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     // define array sizes
10    const int responseSize = 40; // size of array responses
11    const int frequencySize = 11; // size of array frequency
12
13    // place survey responses in array responses
14    const int responses[ responseSize ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
15        10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
16        5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
17
18    // initialize frequency counters to 0
19    int frequency[ frequencySize ] = {};
20
21    // for each answer, select responses element and use that value
22    // as frequency subscript to determine element to increment
23    for ( int answer = 0; answer < responseSize; answer++ )
24        frequency[ responses[ answer ] ]++;

```



**Fig. 6.11** | Poll analysis program. (Part I of 2.)

2021/10/26

Andy Yu-Guang Chen

27



## 6.4.7 Using Arrays to Summarize Survey Results



```

25    cout << "Rating" << setw( 17 ) << "Frequency" << endl;
26
27
28    // output each array element's value
29    for ( int rating = 1; rating < frequencySize; rating++ )
30        cout << setw( 6 ) << rating << setw( 17 ) << frequency[ rating ]
31        << endl;
32 } // end main

```

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



**Fig. 6.11** | Poll analysis program. (Part 2 of 2.)

2021/10/26

Andy Yu-Guang Chen

28



## Appendix: Variable Length Arrays in C and C++

- ◆ C supports variable sized arrays from C99 standard. For example, the below program compiles and runs fine in C
- ◆ But C++ standard (till C++11) doesn't support variable sized arrays
- ◆ The program may work in **GCC compiler**, because GCC compiler provides an extension to support them.

```
void fun(int n)
{
    int arr[n];
    // .....
}

int main()
{
    fun(6);
}
```



2021/10/26

Andy Yu-Guang Chen

29



## Appendix: C++ Standard

- ◆ C++ is standardized by an ISO working group known as JTC1/SC22/WG21

Year	C++ Standard	Informal name
1998	ISO/IEC 14882:1998 <sup>[23]</sup>	C++98
2003	ISO/IEC 14882:2003 <sup>[24]</sup>	C++03
2011	ISO/IEC 14882:2011 <sup>[25]</sup>	C++11, C++0x
2014	ISO/IEC 14882:2014 <sup>[26]</sup>	C++14, C++1y
2017	ISO/IEC 14882:2017 <sup>[9]</sup>	C++17, C++1z
2020	to be determined	C++20, <sup>[17]</sup> C++2a



2021/10/26

Andy Yu-Guang Chen

30



## 6.5 Passing Arrays to Functions

- ◆ To pass an array argument to a function, specify the name of the array without any brackets.
- ◆ When passing an array to a function, the array size is normally passed as well.
  - Otherwise, we would need to build this knowledge into the called function itself or, worse yet, place the array size in a global variable.
- ◆ C++ passes arrays to functions by reference
  - The called functions can modify the elements in the original arrays.
- ◆ The name of the array stores the address in the computer's memory of the first element of the array.
- ◆ Individual array elements are still passed by value exactly as simple variables are.
  - Use the subscripted name of the array element as the argument.



2021/10/26

Andy Yu-Guang Chen

31



## 6.5 Passing Arrays to Functions



### Performance Tip 6.2

*Passing arrays by reference makes sense for performance reasons. Passing by value would require copying each element. For large, frequently passed arrays, this would be time consuming and would require considerable storage for the copies of the array elements.*



2021/10/26

Andy Yu-Guang Chen

32



## 6.5 Passing Arrays to Functions

```

1 // Fig. 6.13: fig06_13.cpp
2 // Passing arrays and individual array elements to functions.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 void modifyArray( int [], int ); // appears strange; array and size
8 void modifyElement( int ); // receive array element value
9
10 int main()
11 {
12     const int arraySize = 5; // size of array a
13     int a[ arraySize ] = { 0, 1, 2, 3, 4 }; // initialize array a
14
15     cout << "Effects of passing entire array by reference:"
16         << "\n\nThe values of the original array are:\n";
17
18     // output original array elements
19     for ( int i = 0; i < arraySize; i++ )
20         cout << setw( 3 ) << a[ i ];
21
22     cout << endl;
23

```

**Fig. 6.13** | Passing arrays and individual array elements to functions. (Part 1 of 3.)



2021/10/26

Andy Yu-Guang Chen

33



## 6.5 Passing Arrays to Functions

```

24 // pass array a to modifyArray by reference
25 modifyArray( a, arraySize );
26 cout << "The values of the modified array are:\n";
27
28 // output modified array elements
29 for ( int j = 0; j < arraySize; j++ )
30     cout << setw( 3 ) << a[ j ];
31
32 cout << "\n\nEffects of passing array element by value:"
33     << "\n\na[3] before modifyElement: " << a[ 3 ] << endl;
34
35 modifyElement( a[ 3 ] ); // pass array element a[ 3 ] by value
36 cout << "a[3] after modifyElement: " << a[ 3 ] << endl;
37 } // end main
38
39 // in function modifyArray, "b" points to the original array "a" in memory
40 void modifyArray( int b[], int sizeOfArray )
41 {
42     // multiply each array element by 2
43     for ( int k = 0; k < sizeOfArray; k++ )
44         b[ k ] *= 2;
45 } // end function modifyArray
46

```

**Fig. 6.13** | Passing arrays and individual array elements to functions. (Part 2 of 3.)



2021/10/26

Andy Yu-Guang Chen

34



## 6.5 Passing Arrays to Functions

```

47 // in function modifyElement, "e" is a local copy of
48 // array element a[ 3 ] passed from main
49 void modifyElement( int e )
50 {
51     // multiply parameter by 2
52     cout << "Value of element in modifyElement: " << ( e *= 2 ) << endl;
53 } // end function modifyElement

```

Effects of passing entire array by reference:

The values of the original array are:

0 1 2 3 4

The values of the modified array are:

0 2 4 6 8

Effects of passing array element by value:

a[3] before modifyElement: 6

Value of element in modifyElement: 12

a[3] after modifyElement: 6

**Fig. 6.13** | Passing arrays and individual array elements to functions. (Part 3 of 3.)



2021/10/26

Andy Yu-Guang Chen

35



## 6.5 Passing Arrays to Functions



### Software Engineering Observation 6.4

*Applying the `const` type qualifier to an array parameter in a function definition to prevent the original array from being modified in the function body is another example of the principle of least privilege. Functions should not be given the capability to modify an array unless it's absolutely necessary.*



2021/10/26

Andy Yu-Guang Chen

36



## 6.5 Passing Arrays to Functions

```

1 // Fig. 6.14: fig06_14.cpp
2 // Demonstrating the const type qualifier.
3 #include <iostream>
4 using namespace std;
5
6 void tryToModifyArray( const int [] ); // function prototype
7
8 int main()
9 {
10     int a[] = { 10, 20, 30 };
11
12     tryToModifyArray( a );
13     cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
14 } // end main
15
16 // In function tryToModifyArray, "b" cannot be used
17 // to modify the original array "a" in main.
18 void tryToModifyArray( const int b[] )
19 {
20     b[ 0 ] /= 2; // compilation error
21     b[ 1 ] /= 2; // compilation error
22     b[ 2 ] /= 2; // compilation error
23 } // end function tryToModifyArray

```

**Fig. 6.14** | `const` type qualifier applied to an array parameter. (Part I of 2.)



2021/10/26

Andy Yu-Guang Chen

37



## 6.5 Passing Arrays to Functions

*Microsoft Visual C++ compiler error message:*

```

c:\cpphttp6_examples\ch06\fig06_14\fig06_14.cpp(20) : error C3892: 'b' : you
cannot assign to a variable that is const
c:\cpphttp6_examples\ch06\fig06_14\fig06_14.cpp(21) : error C3892: 'b' : you
cannot assign to a variable that is const
c:\cpphttp6_examples\ch06\fig06_14\fig06_14.cpp(22) : error C3892: 'b' : you
cannot assign to a variable that is const

```

*GNU C++ compiler error message:*

```

fig06_14.cpp:20: error: assignment of read-only location
fig06_14.cpp:21: error: assignment of read-only location
fig06_14.cpp:22: error: assignment of read-only location

```

**Fig. 6.14** | `const` type qualifier applied to an array parameter. (Part 2 of 2.)



2021/10/26

Andy Yu-Guang Chen

38



## 6.6 Searching Arrays with Linear Search

- ◆ Often it may be necessary to determine whether an array contains a value that matches a certain **key** value.
  - Called **searching**.
- ◆ The **linear search** compares each element of an array with a **search key** (line 36).
  - Because the array is not in any particular order, it's just as likely that the value will be found in the first element as the last.
  - On average (or it should be expected value), therefore, the program must compare the search key with half the elements of the array.
- ◆ To determine that a value is not in the array, the program must compare the search key to every element of the array.



2021/10/26

Andy Yu-Guang Chen

39



## 6.6 Searching Arrays with Linear Search

```

1 // Fig. 6.15: fig06_15.cpp
2 // Linear search of an array.
3 #include <iostream>
4 using namespace std;
5
6 int linearSearch( const int [], int, int ); // prototype
7
8 int main()
9 {
10    const int arraySize = 100; // size of array a
11    int a[ arraySize ]; // create array a
12    int searchKey; // value to locate in array a
13
14    for ( int i = 0; i < arraySize; i++ )
15        a[ i ] = 2 * i; // create some data
16
17    cout << "Enter integer search key: ";
18    cin >> searchKey;
19
20    // attempt to locate searchKey in array a
21    int element = linearSearch( a, searchKey, arraySize );
22

```

**Fig. 6.15** | Linear search of an array. (Part I of 3.)



2021/10/26

Andy Yu-Guang Chen

40



## 6.6 Searching Arrays with Linear Search

```

23     // display results
24     if ( element != -1 )
25         cout << "Found value in element " << element << endl;
26     else
27         cout << "Value not found" << endl;
28 } // end main
29
30 // compare key to every element of array until location is
31 // found or until end of array is reached; return subscript of
32 // element if key is found or -1 if key not found
33 int linearSearch( const int array[], int key, int sizeOfArray )
34 {
35     for ( int j = 0; j < sizeOfArray; j++ )
36         if ( array[ j ] == key ) // if found,
37             return j; // return location of key
38
39     return -1; // key not found
40 } // end function linearSearch

```

Enter integer search key: 36  
Found value in element 18

Enter integer search key: 37  
Value not found



2021/10/26

Andy Yu-Guang Chen

41



## 6.7 Sorting Arrays with Insertion Sort

- ◆ **Sorting** is to place the data into some particular order such as ascending or descending.
  - An important problem with many applications in computer science.
- ◆ **Insertion sort**—a simple, but inefficient, sorting algorithm.
- ◆ The first iteration of this algorithm looks at the second element and, if it's less than the first element, insert the second element in front of the first element.
- ◆ The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements.
  - All three elements are in order.
- ◆ At the  $i^{th}$  iteration of this algorithm, the first  $i$  elements in the original array will be sorted.



2021/10/26

Andy Yu-Guang Chen

42



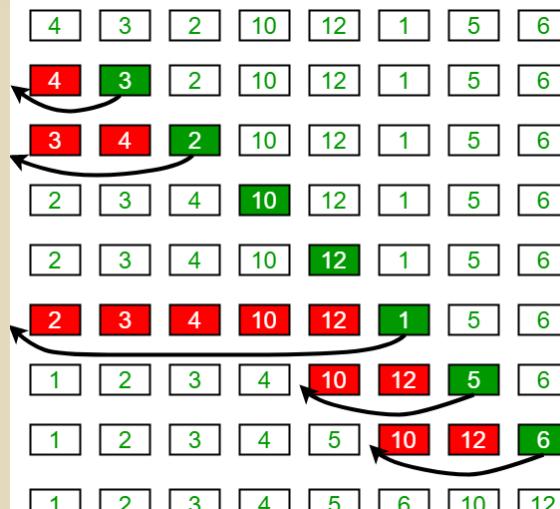
## 6.7 Sorting Arrays with Insertion Sort



2021/10/26

Andy Yu-Guang Chen

43



## 6.7 Sorting Arrays with Insertion Sort

```

1 // Fig. 6.16: fig06_16.cpp
2 // This program sorts an array's values into ascending order.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 int main()
8 {
9     const int arraySize = 10; // size of array a
10    int data[ arraySize ] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
11    int insert; // temporary variable to hold element to insert
12
13    cout << "Unsorted array:\n";
14
15    // output original array
16    for ( int i = 0; i < arraySize; i++ )
17        cout << setw( 4 ) << data[ i ];
18

```

**Fig. 6.16** | Sorting an array with insertion sort. (Part I of 3.)



2021/10/26

Andy Yu-Guang Chen

44



## 6.7 Sorting Arrays with Insertion Sort

```

19 // insertion sort
20 // loop over the elements of the array
21 for ( int next = 1; next < arraySize; next++ )
22 {
23     insert = data[ next ]; // store the value in the current element
24
25     int moveItem = next; // initialize location to place element
26
27     // search for the location in which to put the current element
28     while ( ( moveItem > 0 ) && ( data[ moveItem - 1 ] > insert ) )
29     {
30         // shift element one slot to the right
31         data[ moveItem ] = data[ moveItem - 1 ];
32         moveItem--;
33     } // end while
34
35     data[ moveItem ] = insert; // place inserted element into the array
36 } // end for
37
38 cout << "\nSorted array:\n";
39

```

**Fig. 6.16** | Sorting an array with insertion sort. (Part 2 of 3.)



2021/10/26

Andy Yu-Guang Chen

45



## 6.7 Sorting Arrays with Insertion Sort

```

40 // output sorted array
41 for ( int i = 0; i < arraySize; i++ )
42     cout << setw( 4 ) << data[ i ];
43
44 cout << endl;
45 } // end main

```

```

Unsorted array:
34 56 4 10 77 51 93 30 5 52
Sorted array:
4 5 10 30 34 51 52 56 77 93

```

**Fig. 6.16** | Sorting an array with insertion sort. (Part 3 of 3.)



2021/10/26

Andy Yu-Guang Chen

46



## 6.8 Multidimensional Arrays

- ◆ Arrays with two dimensions often represent **tables of values** arranged in **rows** and **columns**.
- ◆ To identify a particular element, we must specify two subscripts.
  - By convention, the first identifies the element's row and the second identifies the element's column.
- ◆ Arrays with two or more dimensions are known as **multidimensional arrays**.

	Column 0	Column 1	Column 2	Column 3
Row 0	a[ 0 ][ 0 ]	a[ 0 ][ 1 ]	a[ 0 ][ 2 ]	a[ 0 ][ 3 ]
Row 1	a[ 1 ][ 0 ]	a[ 1 ][ 1 ]	a[ 1 ][ 2 ]	a[ 1 ][ 3 ]
Row 2	a[ 2 ][ 0 ]	a[ 2 ][ 1 ]	a[ 2 ][ 2 ]	a[ 2 ][ 3 ]

↑  
↑  
↑

Column subscript  
Row subscript  
Array name



2021/10/26

Andy Yu-Guang Chen

47



## 6.8 Multidimensional Arrays



### Common Programming Error 6.9

Referencing a two-dimensional array element  $a[x][y]$  incorrectly as  $a[x, y]$  is an error. Actually,  $a[x, y]$  is treated as  $a[y]$ , because C++ evaluates the expression  $x, y$  (containing a comma operator) simply as  $y$  (the last of the comma-separated expressions).



2021/10/26

Andy Yu-Guang Chen

48



## 6.8 Multidimensional Arrays

- ◆ A multidimensional array can be initialized in its declaration much like a one-dimensional array.
- ◆ The values are grouped by row in braces.
- ◆ If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0.
- ◆ Figure 6.21 demonstrates initializing two-dimensional arrays in declarations.



2021/10/26

Andy Yu-Guang Chen

49



## 6.8 Multidimensional Arrays

```

1 // Fig. 6.18: fig06_18.cpp
2 // Initializing multidimensional arrays.
3 #include <iostream>
4 using namespace std;
5
6 void printArray( const int [][] [ 3 ] ); // prototype
7 const int rows = 2;
8 const int columns = 3;
9
10 int main()
11 {
12     int array1[ rows ][ columns ] = { { 1, 2, 3 }, { 4, 5, 6 } };
13     int array2[ rows ][ columns ] = { { 1, 2, 3, 4, 5 } };
14     int array3[ rows ][ columns ] = { { { 1, 2 }, { 4 } } };
15
16     cout << "Values in array1 by row are:" << endl;
17     printArray( array1 );
18
19     cout << "\nValues in array2 by row are:" << endl;
20     printArray( array2 );
21
22     cout << "\nValues in array3 by row are:" << endl;
23     printArray( array3 );
24 } // end main

```

**Fig. 6.18** | Initializing multidimensional arrays. (Part I of 3.)

2021/10/26

Andy Yu-Guang Chen

50



## 6.8 Multidimensional Arrays

```

25
26 // output array with two rows and three columns
27 void printArray( const int a[][ columns ] )
28 {
29     // loop through array's rows
30     for ( int i = 0; i < rows; i++ )
31     {
32         // loop through columns of current row
33         for ( int j = 0; j < columns; j++ )
34             cout << a[ i ][ j ] << ' ';
35
36         cout << endl; // start new line of output
37     } // end outer for
38 } // end function printArray

```

Values in array1 by row are:

1 2 3  
4 5 6

Values in array2 by row are:

1 2 3  
4 5 0

Values in array3 by row are:

1 2 0  
4 0 0



2021/10/26

Andy Yu-Guang Chen

51



## 6.9 Case Study: Processing Grades in a Two-Dimensional Array

- ◆ Figure 6.19 uses a two-dimensional array **grades** to store the grades of a number of students on multiple exams.
- ◆ Each row of the array represents a single student's grades for the entire course, and each column represents all the grades the students earned for one particular exam.



2021/10/26

Andy Yu-Guang Chen

52



## 6.9 Case Study: Processing Grades in a Two-Dimensional Array

```

1 // Fig. 6.19: fig06_19.cpp
2 // Analyzing a two-dimensional array of grades.
3 #include <iostream>
4 #include <iomanip> // parameterized stream manipulators
5 using namespace std;
6
7 const int students = 10; // number of students
8 const int tests = 3; // number of tests
9
10 // function prototypes
11 int minimum( const int [][] tests ), int, int );
12 int maximum( const int [][] tests ), int, int );
13 double average( const int [], int );
14 void outputGrades( const int [][] tests ), int, int );
15 void outputBarChart( const int [][] tests ), int, int );
16
17 int main()
18 {

```

**Fig. 6.19** | Analyzing a two-dimensional array of grades. (Part 1 of 9.)



2021/10/26

Andy Yu-Guang Chen

53



## 6.9 Case Study: Processing Grades in a Two-Dimensional Array

```

19 // two-dimensional array of student grades
20 int studentGrades[ students ][ tests ] =
21 { { 87, 96, 70 },
22 { 68, 87, 90 },
23 { 94, 100, 90 },
24 { 100, 81, 82 },
25 { 83, 65, 85 },
26 { 78, 87, 65 },
27 { 85, 75, 83 },
28 { 91, 94, 100 },
29 { 76, 72, 84 },
30 { 87, 93, 73 } };
31
32 // output the studentGrades array showing each student's average
33 outputGrades( studentGrades, students, tests );
34
35 // call functions minimum and maximum
36 cout << "\nLowest of all the grades is "
37 << minimum( studentGrades, students, tests )
38 << "\nHighest of all the grades is "
39 << maximum( studentGrades, students, tests ) << endl;
40
41 // display a bar chart of the grades
42 outputBarChart( studentGrades, students, tests );
43 } // end main

```

**Fig. 6.19** | Analyzing a two-dimensional array of grades. (Part 2 of 9.)



2021/10/26

Andy Yu-Guang Chen

54





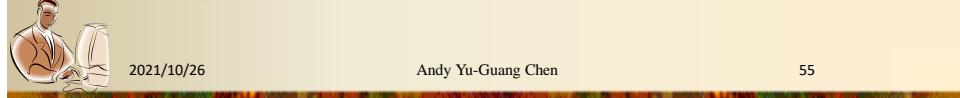
## 6.9 Case Study: Processing Grades in a Two-Dimensional Array

```

44 // find the minimum of all the grades in the double array
45 int minimum( const int grades[][ tests ], int pupils, int exams )
46 {
47     int lowGrade = 100; // assume lowest grade is 100
48
49     // loop through rows of grades array
50     for ( int student = 0; student < pupils; student++ )
51     {
52         // loop through columns of current row
53         for ( int test = 0; test < exams; test++ )
54         {
55             // if current grade less than lowGrade, assign it to lowGrade
56             if ( grades[ student ][ test ] < lowGrade )
57                 lowGrade = grades[ student ][ test ]; // new lowest grade
58         } // end inner for
59     } // end outer for
60
61     return lowGrade; // return lowest grade
62 } // end function minimum
63
64

```

**Fig. 6.19** | Analyzing a two-dimensional array of grades. (Part 3 of 9.)



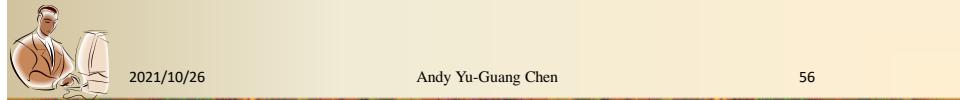

## 6.9 Case Study: Processing Grades in a Two-Dimensional Array

```

65 // find the maximum of all the grades in the double array
66 int maximum( const int grades[][ tests ], int pupils, int exams )
67 {
68     int highGrade = 0; // assume highest grade is 0
69
70     // loop through rows of grades array
71     for ( int student = 0; student < pupils; student++ )
72     {
73         // loop through columns of current row
74         for ( int test = 0; test < exams; test++ )
75         {
76             // if current grade greater than highGrade, assign to highGrade
77             if ( grades[ student ][ test ] > highGrade )
78                 highGrade = grades[ student ][ test ]; // new highest grade
79         } // end inner for
80     } // end outer for
81
82     return highGrade; // return highest grade
83 } // end function maximum
84

```

**Fig. 6.19** | Analyzing a two-dimensional array of grades. (Part 4 of 9.)





## 6.9 Case Study: Processing Grades in a Two-Dimensional Array



```

85 // determine average grade for particular set of grades
86 double average( const int setOfGrades[], const int gradeCount )
87 {
88     int total = 0; // initialize total
89
90     // sum grades in array
91     for ( int grade = 0; grade < gradeCount; grade++ )
92         total += setOfGrades[ grade ];
93
94     // return average of grades
95     return static_cast< double >( total ) / gradeCount;
96 } // end function average
97
98 // output bar chart displaying grade distribution
99 void outputBarChart( const int grades[][ tests ], int pupils, int exams )
100 {
101     cout << "\nOverall grade distribution:" << endl;
102
103     // stores frequency of grades in each range of 10 grades
104     const int frequencySize = 11;
105     int frequency[ frequencySize ] = {}; // initialize elements to 0
106

```

**Fig. 6.19** | Analyzing a two-dimensional array of grades. (Part 5 of 9.)



2021/10/26

Andy Yu-Guang Chen

57



## 6.9 Case Study: Processing Grades in a Two-Dimensional Array



```

107 // for each grade, increment the appropriate frequency
108 for ( int student = 0; student < pupils; student++ )
109 {
110     for ( int test = 0; test < exams; test++ )
111         ++frequency[ grades[ student ][ test ] / 10 ];
112
113 // for each grade frequency, print bar in chart
114 for ( int count = 0; count < frequencySize; count++ )
115 {
116     // output bar label ("0-9:", ..., "90-99:", "100:")
117     if ( count == 0 )
118         cout << " 0-9: ";
119     else if ( count == 10 )
120         cout << " 100: ";
121     else
122         cout << count * 10 << "-" << ( count * 10 ) + 9 << ": ";
123
124     // print bar of asterisks
125     for ( int stars = 0; stars < frequency[ count ]; stars++ )
126         cout << '*';
127
128     cout << endl; // start a new line of output
129 } // end outer for
130 } // end function outputBarChart

```

**Fig. 6.19** | Analyzing a two-dimensional array of grades. (Part 6 of 9.)



2021/10/26

Andy Yu-Guang Chen

58



## 6.9 Case Study: Processing Grades in a Two-Dimensional Array

```

131 // output the contents of the grades array
132 void outputGrades( const int grades[][ tests ], int pupils, int exams )
133 {
134     cout << "\nThe grades are:\n\n";
135     cout << "          "; // align column heads
136
137     // create a column heading for each of the tests
138     for ( int test = 0; test < tests; test++ )
139         cout << "Test " << test + 1 << " ";
140
141     cout << "Average" << endl; // student average column heading
142
143     // create rows/columns of text representing array grades
144     for ( int student = 0; student < pupils; student++ )
145     {
146         cout << "Student " << setw( 2 ) << student + 1;
147
148         // output student's grades
149         for ( int test = 0; test < exams; test++ )
150             cout << setw( 8 ) << grades[ student ][ test ];
151
152

```

**Fig. 6.19** | Analyzing a two-dimensional array of grades. (Part 7 of 9.)



2021/10/26

Andy Yu-Guang Chen

59



## 6.9 Case Study: Processing Grades in a Two-Dimensional Array

```

153     // call function average to calculate student's average;
154     // pass row of grades and the value of tests as the arguments
155     double averageGrade = average( grades[ student ], tests );
156     cout << setw( 9 ) << setprecision( 2 ) << fixed
157         << averageGrade << endl;
158 } // end outer for
159 } // end function outputGrades

```

The grades are:

	Test 1	Test 2	Test 3	Average
Student 1	87	96	70	84.33
Student 2	68	87	90	81.67
Student 3	94	100	90	94.67
Student 4	100	81	82	87.67
Student 5	83	65	85	77.67
Student 6	78	87	65	76.67
Student 7	85	75	83	81.00
Student 8	91	94	100	95.00
Student 9	76	72	84	77.33
Student 10	87	93	73	84.33

**Fig. 6.19** | Analyzing a two-dimensional array of grades. (Part 8 of 9.)



2021/10/26

Andy Yu-Guang Chen

60



## 6.9 Case Study: Processing Grades in a Two-Dimensional Array



```
Lowest of all the grades is 65
Highest of all the grades is 100
```

Overall grade distribution:

0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: *****
80-89: *****
90-99: *****
100: ***

Fig. 6.19 | Analyzing a two-dimensional array of grades. (Part 9 of 9.)



2021/10/26

Andy Yu-Guang Chen

61



## 6.10 Introduction to C++ Standard Library Class Template `vector`



- ◆ C-style pointer-based arrays have great potential for errors and are not flexible

- A program can easily “walk off” either end of an array, because C++ does not check whether subscripts fall outside the range of an array.
- Two arrays cannot be meaningfully compared with equality operators or relational operators.
- The size of the array must be passed as an additional argument when an array is passed to a general-purpose function.
- One array cannot be assigned to another with the assignment operator(s).

- ◆ C++ Standard class template `vector` represents a more robust type of array featuring many additional capabilities.

- Defined in header `<vector>` and belongs to namespace `std`.



2021/10/26

Andy Yu-Guang Chen

62



## 6.10 Introduction to C++ Standard Library



### Class Template vector

- ◆ The program of Fig. 6.25 demonstrates capabilities provided by class template **vector** that are not available for C-style pointer-based arrays.
- ◆ Standard class template **vector** is defined in header `<vector>` and belongs to namespace **std**.



2021/10/26

Andy Yu-Guang Chen

63



## 6.10 Introduction to C++ Standard Library



### Class Template vector

```

1 // Fig. 6.20: fig06_20.cpp
2 // Demonstrating C++ Standard Library class template vector.
3 #include <iostream>
4 #include <iomanip>
5 #include <vector>
6 using namespace std;
7
8 void outputVector( const vector< int > & );
// display the vector
9 void inputVector( vector< int > & );
// input values into the vector
10
11 int main()
12 {
13     vector< int > integers1( 7 ); // 7-element vector< int >
14     vector< int > integers2( 10 ); // 10-element vector< int >
15
16     // print integers1 size and contents
17     cout << "Size of vector integers1 is " << integers1.size()
18     << "\nvector after initialization:" << endl;
19     outputVector( integers1 );
20
21     // print integers2 size and contents
22     cout << "\nSize of vector integers2 is " << integers2.size()
23     << "\nvector after initialization:" << endl;
24     outputVector( integers2 );

```

**Fig. 6.20** | C++ Standard Library class template **vector**. (Part 1 of 7.)



2021/10/26

Andy Yu-Guang Chen

64

## 6.10 Introduction to C++ Standard Library



### Class Template vector

```

25 // input and print integers1 and integers2
26 cout << "\nEnter 17 integers:" << endl;
27 inputVector( integers1 );
28 inputVector( integers2 );
29
30 cout << "\nAfter input, the vectors contain:\n"
31     << "integers1:" << endl;
32 outputVector( integers1 );
33 cout << "integers2:" << endl;
34 outputVector( integers2 );
35
36 // use inequality (!=) operator with vector objects
37 cout << "\nEvaluating: integers1 != integers2" << endl;
38
39 if ( integers1 != integers2 )
40     cout << "integers1 and integers2 are not equal" << endl;
41
42 // create vector integers3 using integers1 as an
43 // initializer; print size and contents
44 vector< int > integers3( integers1 ); // copy constructor
45
46

```

**Fig. 6.20** | C++ Standard Library class template `vector`. (Part 2 of 7.)



2021/10/26

Andy Yu-Guang Chen

65

## 6.10 Introduction to C++ Standard Library



### Class Template vector

```

47 cout << "\nSize of vector integers3 is " << integers3.size()
48     << "\nvector after initialization:" << endl;
49 outputVector( integers3 );
50
51 // use overloaded assignment (=) operator
52 cout << "\nAssigning integers2 to integers1:" << endl;
53 integers1 = integers2; // assign integers2 to integers1
54
55 cout << "integers1:" << endl;
56 outputVector( integers1 );
57 cout << "integers2:" << endl;
58 outputVector( integers2 );
59
60 // use equality (==) operator with vector objects
61 cout << "\nEvaluating: integers1 == integers2" << endl;
62
63 if ( integers1 == integers2 )
64     cout << "integers1 and integers2 are equal" << endl;
65
66 // use square brackets to create rvalue
67 cout << "\nintegers1[5] is " << integers1[ 5 ];
68

```

**Fig. 6.20** | C++ Standard Library class template `vector`. (Part 3 of 7.)



2021/10/26

Andy Yu-Guang Chen

66

## 6.10 Introduction to C++ Standard Library



```

69 // use square brackets to create lvalue
70 cout << "\n\nAssigning 1000 to integers1[5]" << endl;
71 integers1[ 5 ] = 1000;
72 cout << "integers1:" << endl;
73 outputVector( integers1 );
74
75 // attempt to use out-of-range subscript
76 cout << "\nAttempt to assign 1000 to integers1.at( 15 )" << endl;
77 integers1.at( 15 ) = 1000; // ERROR: out of range
78 } // end main
79
80 // output vector contents
81 void outputVector( const vector< int > &array )
82 {
83     size_t i; // declare control variable
84
85     for ( i = 0; i < array.size(); i++ )
86     {
87         cout << setw( 12 ) << array[ i ];
88
89         if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
90             cout << endl;
91     } // end for
92 }
```

**Fig. 6.20** | C++ Standard Library class template vector. (Part 4 of 7.)



2021/10/26

Andy Yu-Guang Chen

67

## 6.10 Introduction to C++ Standard Library



### Class Template vector

```

93     if ( i % 4 != 0 )
94         cout << endl;
95 } // end function outputVector
96
97 // input vector contents
98 void inputVector( vector< int > &array )
99 {
100     for ( size_t i = 0; i < array.size(); i++ )
101         cin >> array[ i ];
102 } // end function inputVector
```

```
Size of vector integers1 is 7
vector after initialization:
    0      0      0      0
```

```
Size of vector integers2 is 10
vector after initialization:
    0      0      0      0
    0      0      0      0
    0      0      0      0
```

```
Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

**Fig. 6.20** | C++ Standard Library class template vector. (Part 5 of 7.)



2021/10/26

Andy Yu-Guang Chen

68

## 6.10 Introduction to C++ Standard Library



### Class Template vector

```

After input, the vectors contain:
integers1:
    1      2      3      4
    5      6      7
integers2:
    8      9      10     11
    12     13     14     15
    16     17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of vector integers3 is 7
vector after initialization:
    1      2      3      4
    5      6      7

Assigning integers2 to integers1:
integers1:
    8      9      10     11
    12     13     14     15
    16     17
integers2:
    8      9      10     11
    12     13     14     15
    16     17
  
```

**Fig. 6.20** | C++ Standard Library class template `vector`. (Part 6 of 7.)



2021/10/26

Andy Yu-Guang Chen

69

## 6.10 Introduction to C++ Standard Library



### Class Template vector

```

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
    8      9      10     11
    12     1000    14     15
    16     17

Attempt to assign 1000 to integers1.at( 15 )
Platform specific error message will be displayed
  
```

**Fig. 6.20** | C++ Standard Library class template `vector`. (Part 7 of 7.)



2021/10/26

Andy Yu-Guang Chen

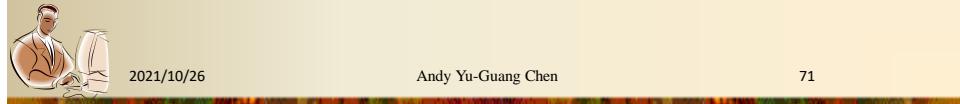
70

## 6.10 Introduction to C++ Standard Library



### Class Template vector

- ◆ By default, all the elements of a **vector** object are set to 0.
- ◆ **vectors** can be defined to store any data type.
- ◆ **vector** member function **size** obtain the number of elements in the **vector**.
- ◆ You can use square brackets, **[ ]**, to access the elements in a **vector**.
- ◆ **vector** objects can be compared with one another using the equality operators.
- ◆ You can create a new **vector** object that is initialized with a copy of an existing **vector**.

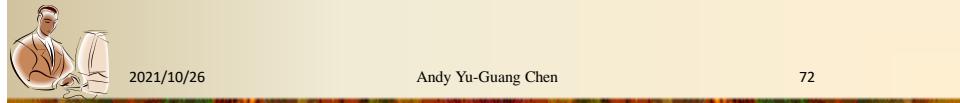


## 6.10 Introduction to C++ Standard Library



### Class Template vector

- ◆ You can use the assignment (=) operator with **vector** objects.
- ◆ As with C-style pointer-based arrays, C++ does not perform any bounds checking when **vector** elements are accessed with square brackets.
- ◆ Standard class template **vector** provides bounds checking in its member function **at**, which “throws an exception” (see Chapter 16, Exception Handling) if its argument is an invalid subscript.



## 6.11 Introduction to C++ Standard Library



### Class string

- ◆ The string in C/C++ is a series of characters that are treated as a single unit
  - Written in double quotes, ex: "Hello"
- ◆ In C, strings are arrays of characters that end with **an extra character '\0'** (NULL, ASCII code=0)
  - Ex: 'H', 'e', 'l', 'l', 'o', '\0' (6 elements)
  - Not convenient to deal with
- ◆ C++ standard library class **string** provides more powerful capabilities
- ◆ In figure 6.21, lines 9–12 create 4 String objects
  - s1 is initialized with the literal "happy"
  - s2 is initialized with the literal " birthday"
  - s3 and s4 create empty strings



2021/10/26

Andy Yu-Guang Chen

73

## 6.11 Introduction to C++ Standard Library



### Class string

```

1 // Fig. 6.21: fig06_21.cpp
2 // Standard Library class string.
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string s1( "happy" ); // string object s1 initialized to "happy"
10    string s2( " birthday" ); // s2 initialized to " birthday"
11    string s3; // s3 is empty
12    string s4; // s4 is empty
13
14    // read a line of text into a string object
15    cout << "Enter a line of text: ";
16    getline( cin, s4 ); // read line of text into s4
17
18    // display each string's contents
19    cout << "s1 is '" << s1 << "'"; s2 is '" << s2
20    << "'"; s3 is '" << s3 << "'"; s4 is '" << s4 << "'\n\n";
21
22    // determine the length of each string
23    cout << "s1 length " << s1.length() << "; s2 length " << s2.length()
24    << "; s3 length " << s3.length() << "; s4 length " << s4.length();

```

Fig. 6.21 | Standard Library class **string**. (Part 1 of 5.)

2021/10/26

Andy Yu-Guang Chen

74

## 6.11 Introduction to C++ Standard Library



### Class string

```

25 // test equality and relational operators
26 cout << "\n\nThe results of comparing s2 and s1:"
27     << "\ns2 == s1 yields " << ( s2 == s1 ? "true" : "false" )
28     << "\ns2 != s1 yields " << ( s2 != s1 ? "true" : "false" )
29     << "\ns2 > s1 yields " << ( s2 > s1 ? "true" : "false" )
30     << "\ns2 < s1 yields " << ( s2 < s1 ? "true" : "false" )
31     << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
32     << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );
33
34
35 // test string member function empty
36 cout << "\n\nTesting s3.empty():" << endl;
37
38 if ( s3.empty() )
39 {
40     cout << "s3 is empty; assigning s1 to s3;" << endl;
41     s3 = s1; // assign s1 to s3
42     cout << "s3 is '" << s3 << "'";
43 } // end if
44
45 // test string concatenation operator
46 cout << "\n\nAfter s1 += s2, s1 is ";
47 s1 += s2; // concatenate s2 to the end of s1
48 cout << s1;

```

**Fig. 6.21** | Standard Library class `string`. (Part 2 of 5.)



2021/10/26

Andy Yu-Guang Chen

75

## 6.11 Introduction to C++ Standard Library



### Class string

```

49 // test string concatenation operator with string literal
50 cout << "\n\ns1 += \" to you\" yields" << endl;
51 s1 += " to you";
52 cout << "s1 is " << s1 << "\n\n";
53
54
55 // test string member function substr
56 cout << "The substring of s1 starting at location 0 for\n"
57     << "14 characters, s1.substr(0, 14), is:\n"
58     << s1.substr( 0, 14 ) << "\n\n"; // displays "happy birthday "
59
60
61 // test substr "to-end-of-string" option
62 cout << "The substring of s1 starting at\n"
63     << "location 15, s1.substr(15), is:\n"
64     << s1.substr( 15 ) << endl; // displays "to you"
65
66 // making a copy of a string
67 string s5( s1 ); // creates s5 as a copy of s1
68 cout << "\ns5 is " << s5;
69
70 // test the subscript operator to create lvalues
71 s1[ 0 ] = 'H'; // replaces h with H
72 s1[ 6 ] = 'B'; // replaces b with B
73 cout << "\n\ns1 after s1[0] = 'H' and s1[6] = 'B' is: " << s1;

```

**Fig. 6.21** | Standard Library class `string`. (Part 3 of 5.)



2021/10/26

Andy Yu-Guang Chen

76

## 6.11 Introduction to C++ Standard Library



### Class string

```

73 // test the subscript operator to create rvalues
74 cout << "\n\ns1[0] is " << s1[ 0 ] << "; s1[2] is " << s1[ 2 ]
75     << "; s1[s1.length()-1] is " << s1[ s1.length() - 1 ];
76
77 // test subscript out of range with string member function "at"
78 cout << "\n\nAttempt to assign 'd' to s1.at( 30 ) yields:" << endl;
79     s1.at( 30 ) = 'd'; // ERROR: subscript out of range
80
81 } // end main

```

```

Enter a line of text: Using class string
s1 is "happy"; s2 is " birthday"; s3 is ""; s4 is "Using class string"

s1 length 5; s2 length 9; s3 length 0; s4 length 18

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 > s1 yields false
s2 < s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

```



**Fig. 6.21** | Standard Library class `string`. (Part 4 of 5.)

2021/10/26

Andy Yu-Guang Chen

77

## 6.11 Introduction to C++ Standard Library



### Class string

```

Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

After s1 += s2, s1 is happy birthday

s1 += " to you" yields
s1 is happy birthday to you

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at
location 15, s1.substr(15), is:
to you

s5 is happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you
s1[0] is H; s1[2] is p; s1[s1.length()-1] is u

Attempt to assign 'd' to s1.at( 30 ) yields:
Platform specific error message will be displayed

```



**Fig. 6.21** | Standard Library class `string`. (Part 5 of 5.)

2021/10/26

Andy Yu-Guang Chen

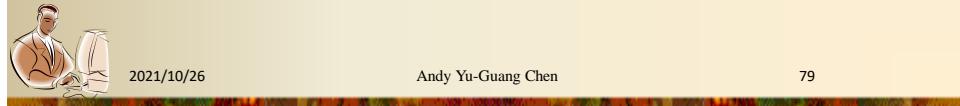
78

## 6.11 Introduction to C++ Standard Library



### Class string

- ◆ Line 16 reads a line of text by using the library function `getline`.
- ◆ From the header file `<string>`
- ◆ When `cin` is used with the stream extraction operator, it reads characters until the first white-space character is reached.
- ◆ Lines 19–20 output string objects, using `cout` and operator `<<`.
- ◆ The `<<` operator is overloaded to handle string objects.

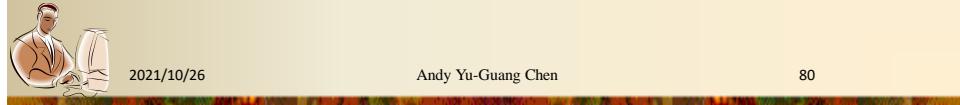


## 6.11 Introduction to C++ Standard Library



### Class string

- ◆ Equality and relational operators are overloaded to compare the string objects (lines 27–33) .
  - Compare the numerical values of the characters (ASCII code) in each string.
- ◆ Line 41 demonstrates class `x`'s overloaded assignment operator.
- ◆ Line 47 demonstrates class `string`'s overloaded `+=` operator for string concatenation.
  - Line 52 demonstrates that a string literal can also be appended to a string object by using operator `+=`.



## 6.11 Introduction to C++ Standard Library



### Class string

- ◆ The member function `length` obtains the number of characters in each string object. (lines 23–24)
- ◆ The member function `empty` returns true if the string is empty; otherwise, it returns false.
- ◆ The member function `substr` (lines 58 and 63) returns a portion of a string as a string object.
  - Two arguments: obtains the number of characters specified by the second argument starting at the position specified by the first argument.
  - One argument: obtains a substring starting from the specified index.



2021/10/26

Andy Yu-Guang Chen

81

## 6.11 Introduction to C++ Standard Library



### Class string

- ◆ Class string's overloaded [] operator enables new characters to replace existing characters in a string.
  - No bounds checking is performed.
- ◆ Class string's overloaded [] operator obtains the character at a specified index.
- ◆ The member function `at` provides bounds checking, which throws an exception if its argument is an invalid subscript.



2021/10/26

Andy Yu-Guang Chen

82



## Summary



- ◆ Introduction to Array
- ◆ How to use Array
- ◆ Array and Function
- ◆ Searching and sorting
- ◆ Multidimensional Arrays
- ◆ Vector and String



2021/10/26

Andy Yu-Guang Chen

83