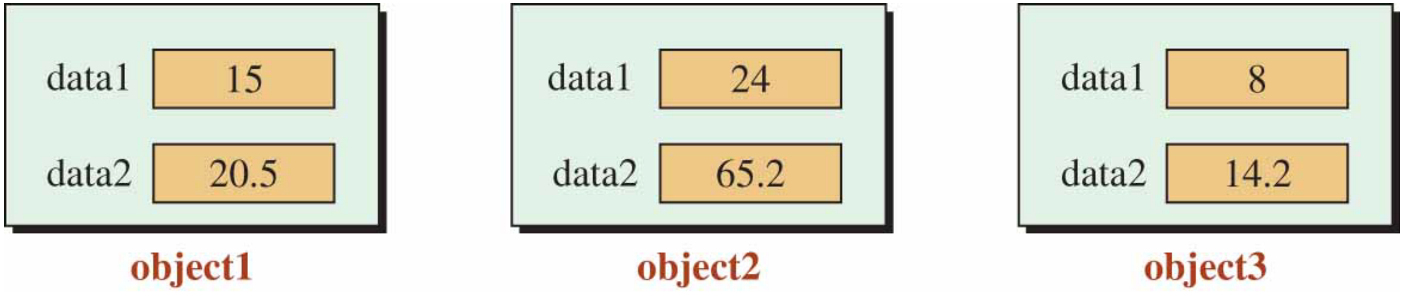
Object-oriented Programming
Arrays

YoungWoon Cha
CSE Department
Spring 2023

Review

Data Members & Member Functions

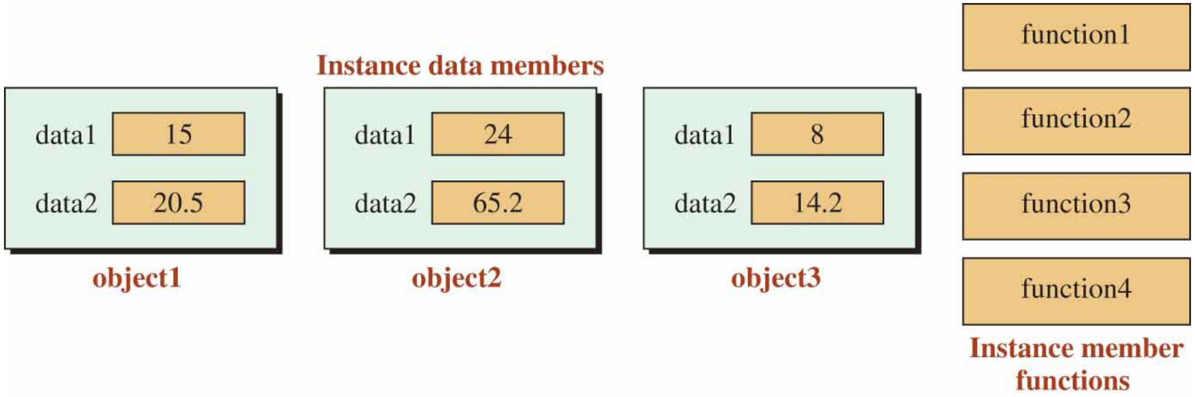
The instance data members of a class are normally private to be accessed only through instance member functions.



	object1	object2	object3
data1	15	24	8
data2	20.5	65.2	14.2

Instance data members encapsulaed in objects

The instance member function of a class needs to be public to be accessed from outside of the class.



	object1	object2	object3
data1	15	24	8
data2	20.5	65.2	14.2

Getter and Setter Member Functions

Accessor Member Function

```
double getRadius() const;       // Host object is read-only
double getPerimeter() const;    // Host object is read-only
double getArea() const;         // Host object is ready-only
```

An accessor instance function must not change the state of the host object; it needs the const modifier.

Mutator Member Function

```
void setRadius(double rds);// No const qualifier for a mutator
```

A mutator instance function changes the state of the host object; it cannot have the *const* modifier.

Static Data Members

A *static data member* is a data member that belongs to all instances; it also belongs to the class itself.

Declaration of a Static Data Member

```
class Rectangle
{
    private:
        …
        static int count;   // Static data member
    public:
        …
}
```

Initialization Of Static Data Members

```
int Rectangle :: count = 0; // initialization of static data member
```

Static Member Function

Declaration of a Static Member Function

```
class Rectangle
{
    private:
    …
        static int count;   // Static data member
    public:
        static int getCount(); // Static member function
    …
}
```

Definition of Static Member Functions

```
int Rectangle :: getCount()
{
    return count;
}
```

Calling Static Member Functions

```
rect.getCount();              // Through an instance
Rectangle :: getCount();      // Through the class
```

A static member function cannot be used to access instance data members because it has no *this* pointer parameter.

Separate Files & Separate Compilation

Figure 7.14 *Three files created in C++ for a class*

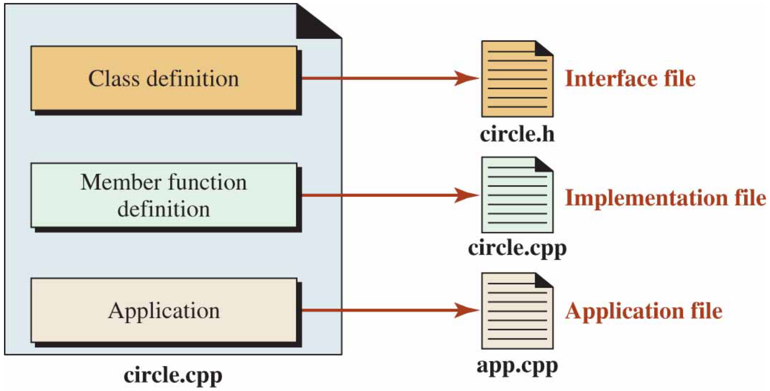
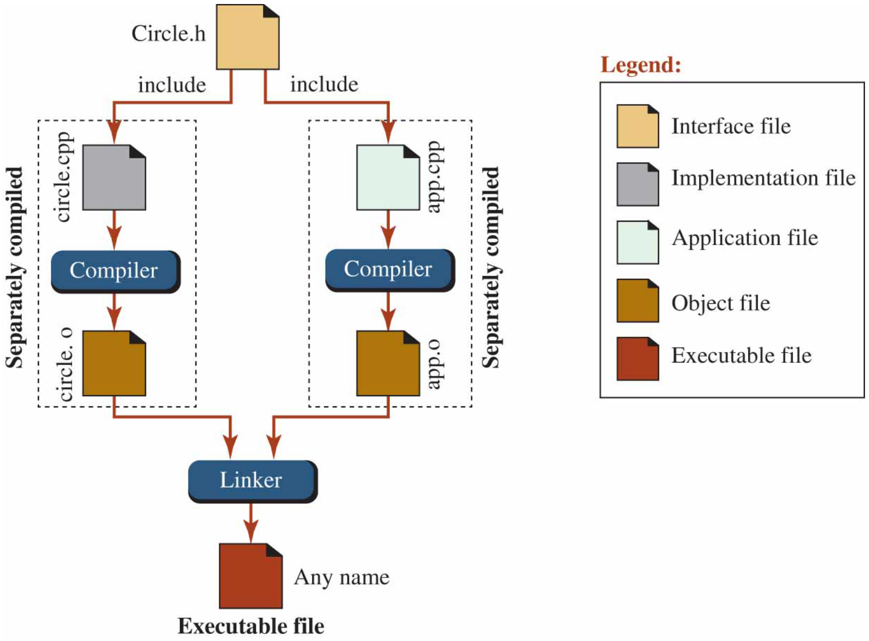
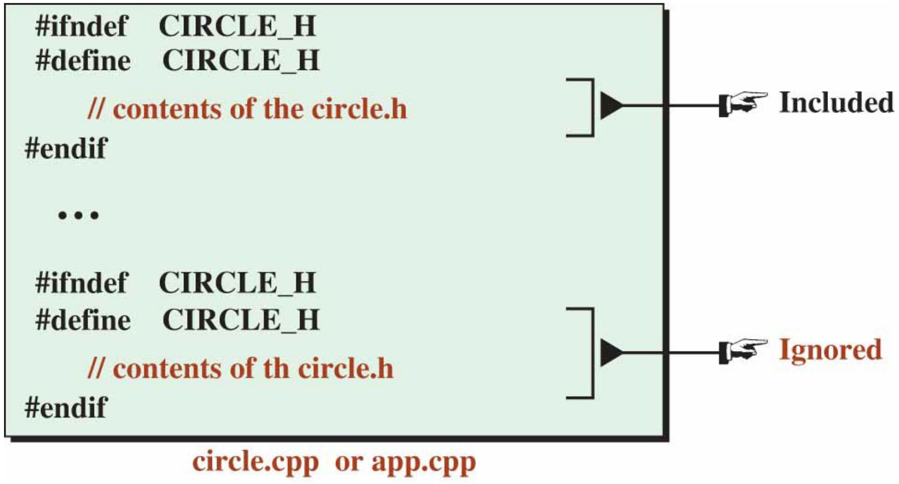


Figure 7.15 *Process of separate compilation*



Preventing Multiple Inclusion

Figure 7.17 *How conditional directives ignore duplicate inclusion*



```
 #ifndef   CIRCLE_H
 #define   CIRCLE_H
      // contents of the circle.h
#endif

   • • •

 #ifndef   CIRCLE_H
 #define   CIRCLE_H
      // contents of th circle.h
#endif
```

In-class Exercise III

- □ **Modify the "Interface file code" with "#pragma once".**

"*#pragma once*" is a preprocessor directive used in C++ to avoid the multiple inclusion of header files (Windows only).

Using "#pragma once" instead of traditional include guards (*#ifndef, #define, and #endif*) is often preferred as it is more concise and can improve compilation time. However, #pragma once is not part of the C++ standard, and some compilers may not support it (Linux). In such cases, using traditional include guards is a reliable alternative.

- □ **Modify the "Interface file code" to avoid "#include" in .h files.**

When *a header file* contains "*#include*" directives, and that header file is included in multiple source files, the contents of the included files are duplicated in each source file, leading to code bloat and longer compile times. This can also cause naming conflicts, redefinition errors, and other issues, especially if the included files define classes or functions.

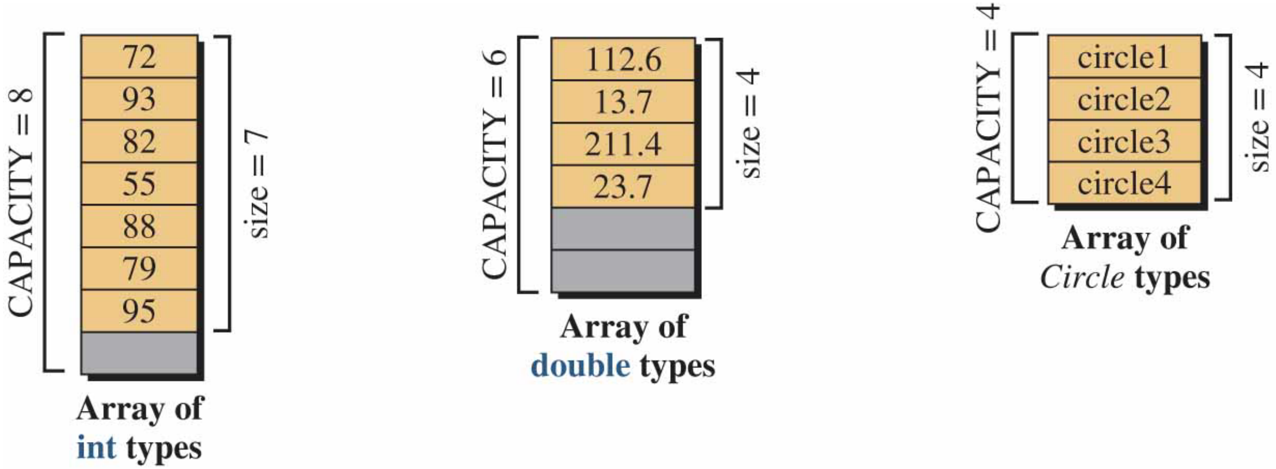
To avoid these issues, it is recommended to include only the necessary declarations (such as class declarations, function prototypes, constants, etc.) in header files, and avoid including any implementation details or other headers that are not required. Instead, the implementation details and necessary headers should be included in the corresponding source (.cpp) files.

Arrays

ONE-DIMENSIONAL ARRAY

A one-dimensional array is a sequence of data items of the same built-in or user-defined type.

When we think of an array, we need to consider three attributes: *type*, *capacity*, and (occupied or being used) *size*.



Note:
We have used uppercase for *CAPACITY* because it is a constant or literal.
We have used lowercase for *size* because it is a variable.
The gray area is part of the array that is not occupied at this moment.

Array Attributes

Type

The *type* of an array is the type of data items (elements) in the array. For example, we can have an int array, a double array, a char array, and a *Circle* array.

The type of all data items in an array must be the same;
the array type is the type of the elements.

Capacity

The *capacity* of an array is the maximum number of elements it can hold. This attribute is either a literal or a constant value that cannot be changed after the array is declared. This is the reason we normally use uppercase letters to define the capacity.

We cannot change the capacity of the array after it has been declared.

(Occupied or being used) Size

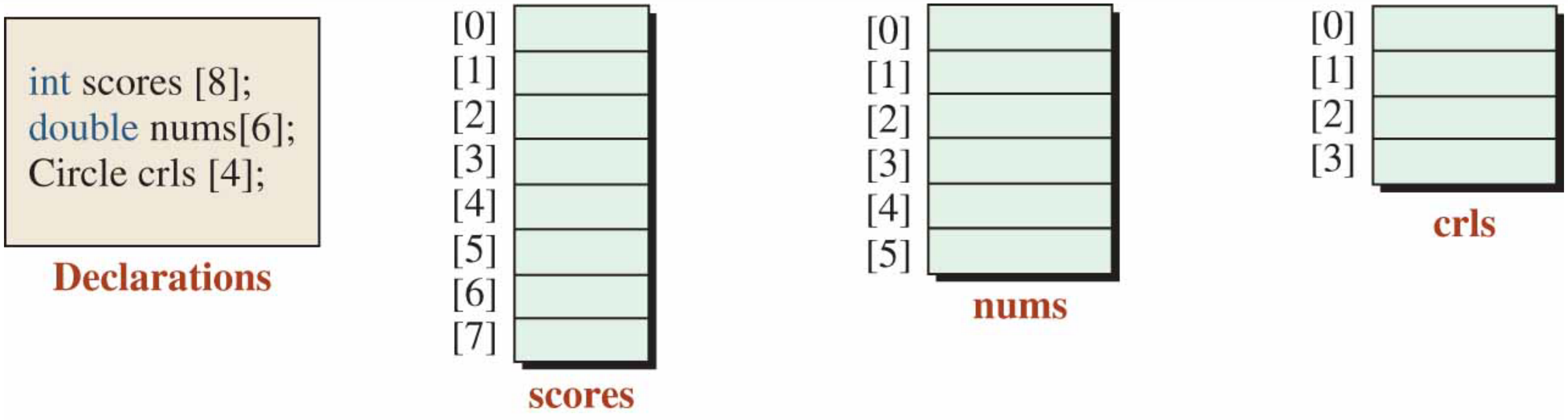
The *size* of an array defines how many elements are valid at each moment, that is how many contain valid data.

We may create an array of capacity 10, but at one moment we may have only three valid elements; at another moment, we may have eight valid elements.

In other words, size is a controlling attribute of the array.

Array size defines the number of valid elements at each moment.

Declaration and Allocation



```
int scores [8];
double nums[6];
Circle crls [4];
```

The array elements are referenced using zero-indexing.

Initialization

Each element of an array is like an individual variable. When we declare an array, the compiler allocates memory locations for each element according to the array type.

1. **If the array is declared in the global area of the program, each element is given a default value according to the array type. The default value for the Boolean type is *false*, for the character type is the *nul* character, for the integer type is 0, for the floating-type is 0.0, and for the object type is the object created by the default constructor.**
2. **If the array is declared inside a function (including *main*), the elements are filled with garbage values (what is left from the previous use of the memory location).**

Initialization

Explicit Initialization

To better control the initial values stored in the array elements, we can explicitly initialize the elements of the array.
The initial values, however, need to be enclosed in braces and separated by commas.

```
const int CAPACITY = 8;
int scores [CAPACITY] = {87, 92, 100, 65, 70, 10, 96, 77};
```

We can also initialize an array of class objects in this way when each element in the initialization is a call to a constructor as shown below:

```
const int CAPACITY = 4;
Circle circles [CAPACITY] = {Circle(4.0), Circle(5.0), Circle(6.0),
Circle(7.0)};
```

Implicit Capacity

When the number of the initialization elements is exactly the capacity of the array, we do not have to define the array capacity as shown below.

```
int scores [ ] = {87, 92, 100, 65, 70, 10, 96, 77};
```

Initialization

Partial Default Filling

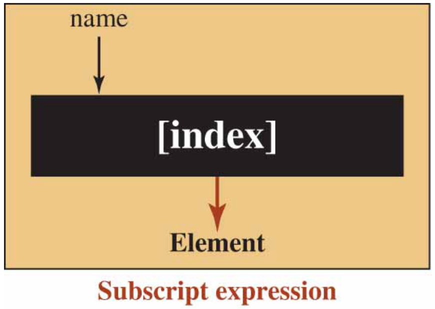
The number of initialization values cannot be larger than the capacity of the array (compilation error), but it can be less than the capacity of the array. In this case, the rest of elements are filled with default values no matter if the array is declared in the global area or inside a function.

```
const int CAPACITY = 10;
int scores [CAPACITY] = {87, 92, 100};
```

The following shows how we can explicitly initialize all elements of in an array of 100 elements to 0.0.

```
const int CAPACITY = 100;
double anArray [CAPACITY] = {0.0};
```

Accessing Array Elements

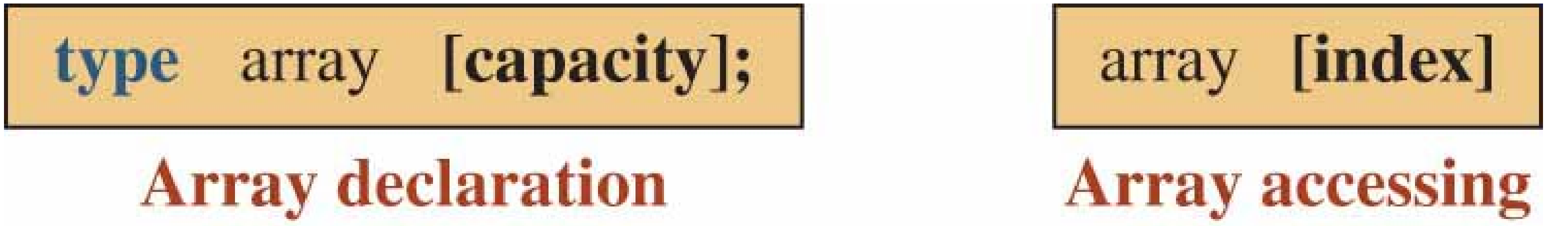


Subscript expression

Notes:
The operator takes the name of the array as the operand.
The operator returns an element of the array, which can be used to access or change the value in the element.

Two Uses of Brackets

Figure 8.6 *Two uses of brackets*



Out of Range Error

One of the hidden errors that cannot be caught during compilation or run time is accessing an element of an array that is not bounded by the capacity of the array.
The index used in the subscript expression, arrayName [index], needs to be in the range 0 and CAPACITY - 1.

Out-of-range error is a serious issue that needs to be carefully avoided.

In-class Exercise I

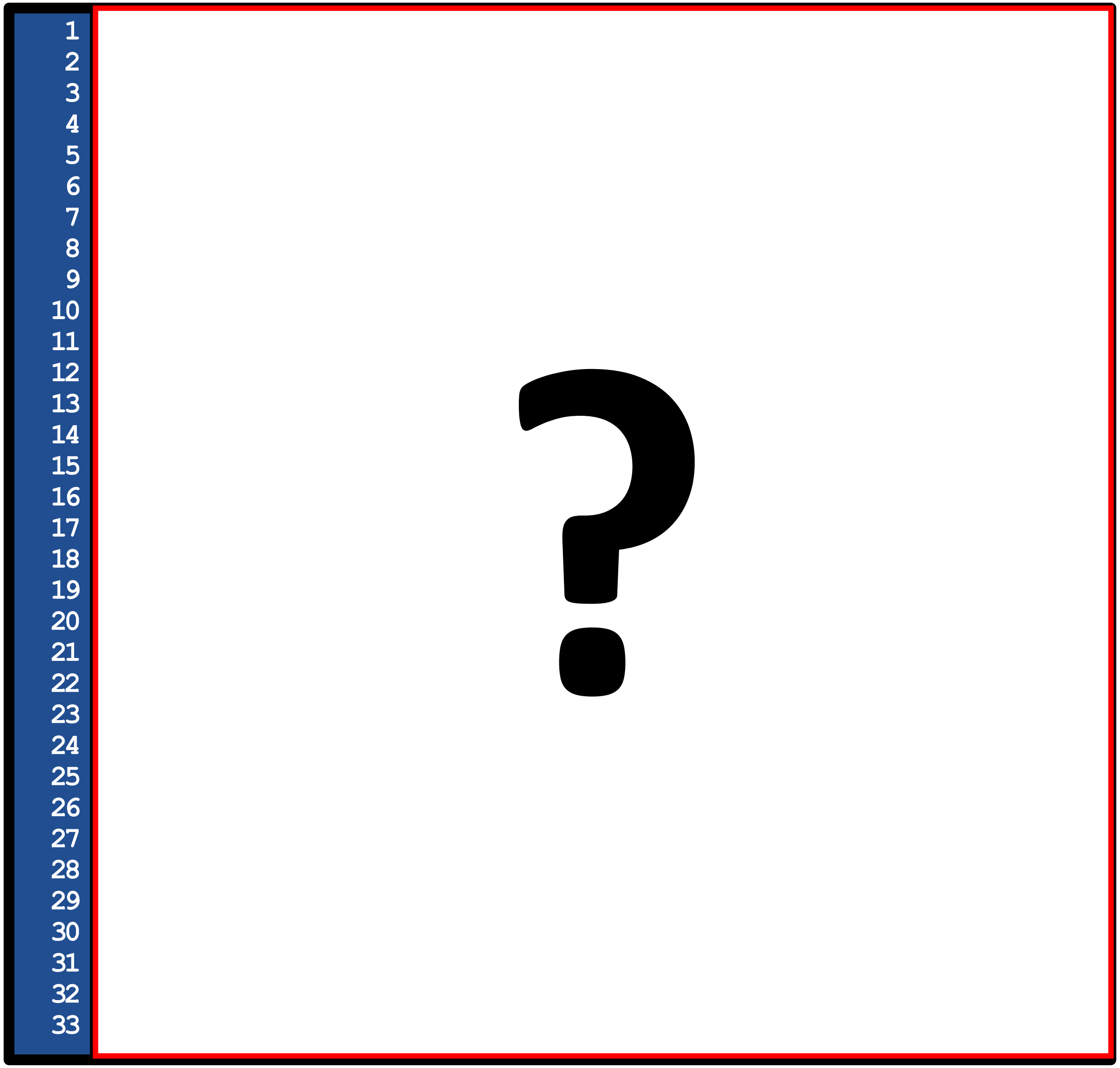
❑ Printing a List in Reverse Order

- Implement your code to get the following output:
- Assume the capacity of the array is 10.

```
Run:
Enter the size (1 to 10): 10
Enter 10 integer(s): 2 3 4 5 6 7 8 9 10 11
Integer(s) in reversed order: 11 10 9 8 7 6 5 4 3 2
```

```
Run:
Enter the size (1 to 10): 0
Enter the size (1 to 10): 11
Enter the size (1 to 10): 7
Enter 7 integer(s): 4 11 78 2 5 3 8 9 // The last integer is ignored.
Integer(s) in reversed order: 8 3 5 2 78 11 4
```

In-class Exercise I



Printing a List in Reverse Order

We have two comments here.

1. We do not have the out-of-range problem in this array because we forced the value of the size variable to be between 1 and CAPACITY.

2. The keyboard is treated as a file in which the numbers are keyed one after another with at least one space between them. As long as the read loop is not terminated, the numbers are read one by one. Even if the user enters a return key before inputting the number of integers defined by the size variable, the program waits for the user to enter the rest of the integers.

Printing a List in Reverse Order Using Files Part 1

Program 8.2 *Reversing the order of a list of numbers using files*

```
/***************************************************************
 * Use of an array to read a list of integers from a file, to  *
 * reverse the order of elements, and to write the reversed    *
 * elements to another file                                    *
 ***************************************************************/
#include <iostream>
#include <fstream>
using namespace std;

int main ( )
{
    // Declarations
    const int CAPACITY = 50;
    int numbers [CAPACITY];
    int size = 0;
    ifstream inputFile;
    ofstream outputFile;
    // Openning the input file
    inputFile.open ("input.txt");
    if (!inputFile)
```

Printing a List in Reverse Order Using Files Part 2

Program 8.2 *Reversing the order of a list of numbers using files*

```
    {
        cout << "Error. Input file cannot be opened." << endl;
        cout << "The program is terminated";
        return 0;
    }
    // Reading the list of numbers from the input file into array
    while (inputFile >> numbers [size] && size <= 50)
    {
        size++;
    }
    // Closing the input file
    inputFile.close();
    // Openning the output file
    outputFile.open ("output.txt");
    if (!outputFile)
    {
        cout << "Error. Output file cannot be opened." << endl;
        cout << "The program is terminated";
        return 0;
    }
```

Printing a List in Reverse Order Using Files Part 3

Program 8.2 *Reversing the order of a list of numbers using files*

```
    // Writing the elements of the reversed array into the output file
    for (int i = size - 1 ; i >= 0 ; i- -)
    {
        outputFile << numbers[i] << " " ;
    }
    // Closing the output file
    outputFile.close();
    return 0;
}
```

When we open the input file and the output file in a text editor, we get the following contents in which the lists are inverse of each other.

Input file	Output file
12 56 72 89 11 71 61 92 34 13	13 34 92 61 71 11 89 72 56 12

Printing a List in Reverse Order Using Files Part 4

There are several points that we need to explain in this program:

- ❑ We have included the <fstream> header file to be able to use operations on files.
- ❑ Since we do not know the count of the numbers in the input file, we need to be cautious and select a large number (50 in this case) for the capacity.
- ❑ If the input file is not successfully opened, we terminate the program with a message (lines 19 to 25). Similarly if the out file is not opened successfully, we terminate the program with a message (lines 34 to 40).
- ❑ The size of the array is automatically set when we reach the end of the input file (line 29).
- ❑ When we run the program, we see nothing unless there is a problem with opening the input or output file.
- ❑ The program never reads more than 50 integers.

In-class Exercise II

Printing a List in Reverse Order Using Files

- Modify the previous code so that it works with arbitrary input and output file names.
- Hint use this:

```
int main(int argc, char **argv)
{
    return 0;
}
```

Array of Three Circles Part 1

Program 8.4 *Creating an array of three circles*

```
/**************************************************************
 *A program that uses the compiled version of the Circle class*
 *to create an array of three circles.                        *
 **************************************************************/
#include <iostream>
#include "circle.h"
using namespace std;

int main ( )
{
    // Declaration of array
    Circle circles [3];
    // Instantiation of objects
    circles [0] = Circle (3.0);
    circles [1] = Circle (4.0);
    circles [2] = Circle (5.0);
    // Printing information
    for (int i = 0; i < 3 ; i++)
    {
       cout << "Information about circle [" << i << "]" << endl;
```

Array of Three Circles Part 2

Program 8.4 *Creating an array of three circles*

```
        cout << "Radius: " << circles[i].getRadius() << " ";
        cout << "Area: " << circles[i].getArea() << " ";
        cout << "perimeter: " << circles[i].getPerimeter() << " ";
        cout << endl;
    }
    return 0;
}
```

```
c++ -c circle.cpp
c++ -c app.cpp
c++ -o application circle.o app.o
application
```

Array of Three Circles Part 3

Program 8.4 *Creating an array of three circles*

```
Run:
Information about circle [0]
Radius: 3 Area: 28.26 perimeter: 18.84


Information about circle [1]
Radius: 4 Area: 50.24 perimeter: 25.12


Information about circle [2]
Radius: 5 Area: 78.5 perimeter: 31.4
```

❑ **Implement the Circle class to get the same result!**

Accessing Operations Part 1

Program 8.5 *Finding the sum, average, smallest, and largest in a sequence*

```
/**************************************************************
 * Use of an array to read a list of integers from a file and *
 * prints the sum, the average, the smallest, and largest, of *
 * the numbers in the file.                                   *
 **************************************************************/
#include <iostream>
#include <fstream>
using namespace std;

int main ( )
{
    // File declaration
    ifstream inputFile;
    // Array and variable declarations
    const int CAPACITY = 50;
    int numbers [CAPACITY];
    int size = 0;
    // Initialization
    int sum = 0;
    double average;
```

Accessing Operations Part 2

Program 8.5 *Finding the sum, average, smallest, and largest in a sequence*

```
    int smallest = 1000000;
    int largest = -1000000;
    // Openning input file with openning validation
    inputFile.open ("numFile.dat");
    if (!inputFile)
    {
        cout << "Error. Input file cannot be opened." << endl;
        cout << "The program is terminated";
        return 0;
    }
    // Reading (copying) numbers from the file
    while (inputFile >> numbers [size])
    {
        size++;
    }
    // Closing input file
    inputFile.close();
    // Finding sum, average, smallest and the largest
    for (int i = 0; i < size; i++)
    {
```

Accessing Operations Part 3

Program 8.5 *Finding the sum, average, smallest, and largest in a sequence*

```cpp
        sum += numbers[i];
        if (numbers[i] < smallest)
        {
            smallest = numbers[i];
        }
        if (numbers[i] > largest)
        {
            largest = numbers[i];
        }
    }
    average = static_cast <double> (sum) / size;
    // Printing results
    cout << "There are " << size << " numbers in the list " << endl;
    cout << "The sum of them is: " << sum << endl;
    cout << "The average of them is: " << average << endl;
    cout << "The smallest number is: " << smallest << endl;
    cout << "The largest number is: " << largest << endl;
    return 0;
}
```

Accessing Operations Part 4

Program 8.5 *Finding the sum, average, smallest, and largest in a sequence*

```
Run:
There are 10 numbers in the list.
The sum of them is: 484
The average of them is: 48.4
The smallest number is: 14
The largest number is: 95
```

❑ **Create an input file to run this code!**

Modifying Operations Part 1

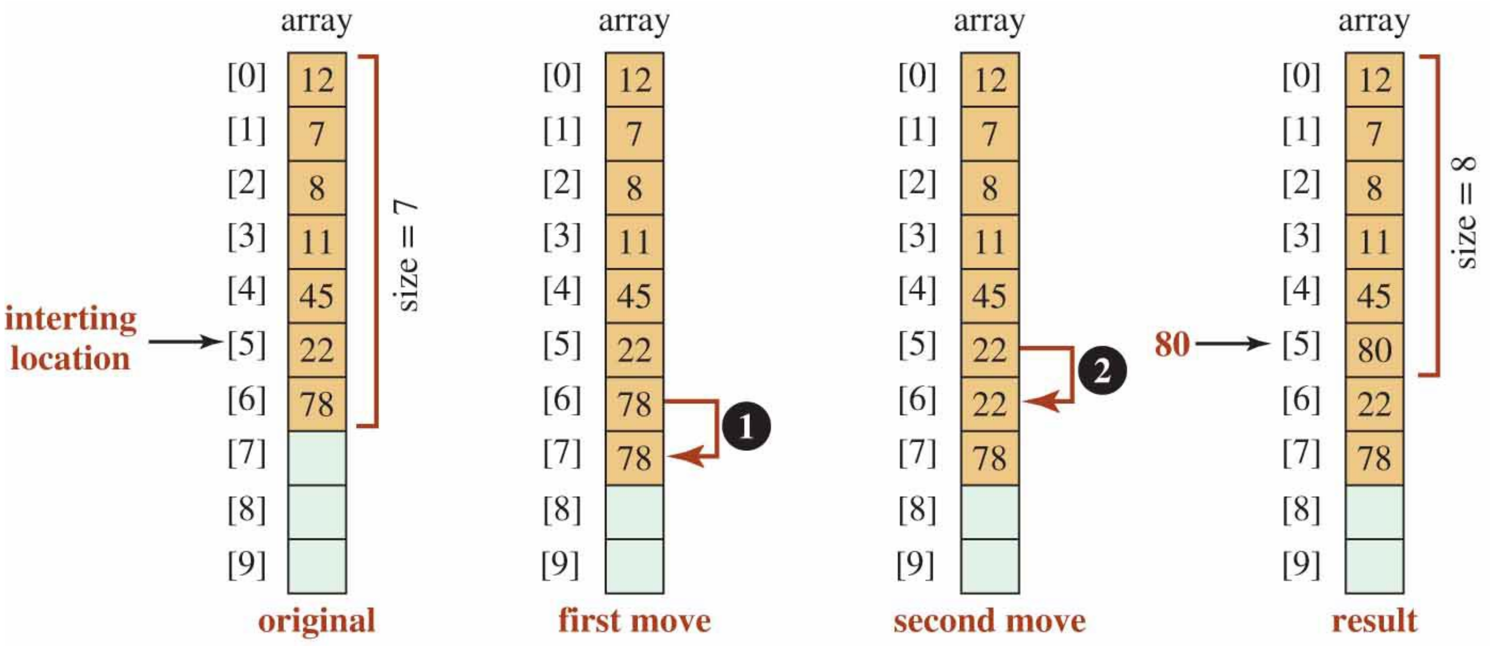
Inserting an Element

We can insert an element of value 80 at index 5.

All elements need to be move one position toward the end to make space for the insertion.

However, this time we need to start movements from the last element.

Figure 8.10 *Inserting an Element*



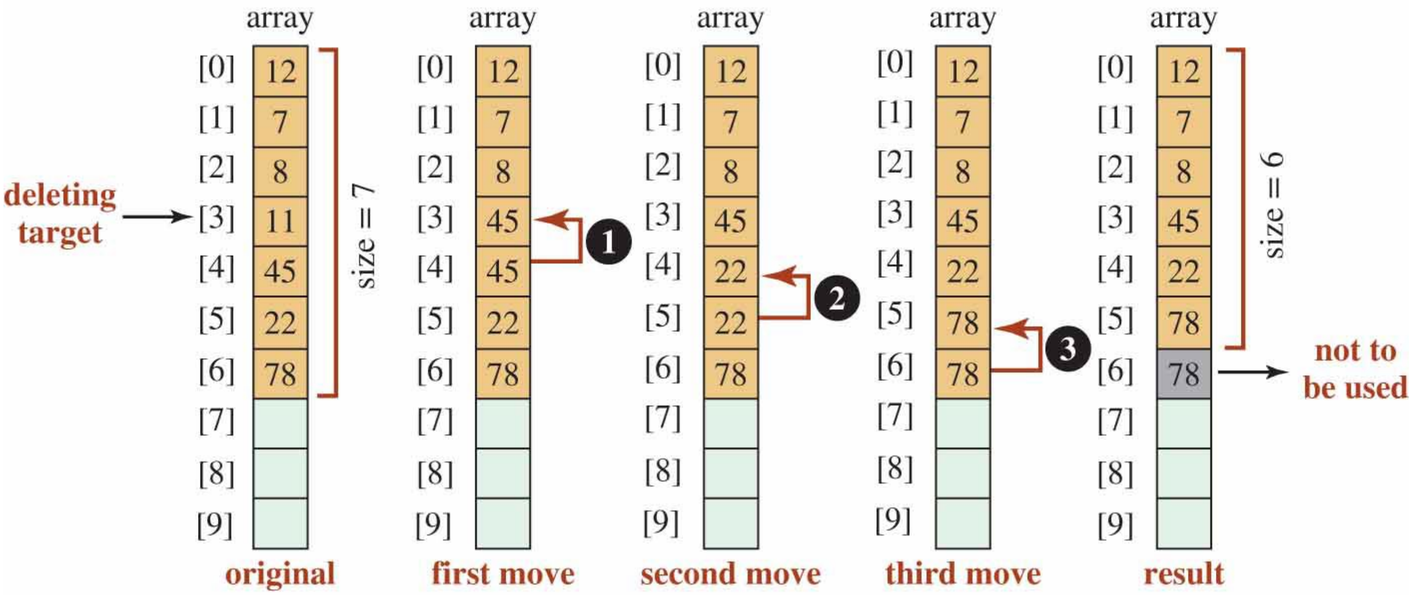
Modifying Operations Part 2

Deleting an Element

Let us assume that we want to delete the element at index 3.

The way to delete an element in the array is to copy (move) all elements after the target index one element toward the beginning of the array.

Figure 8.9 ***Deleting an element in an array***



Using Functions with Arrays

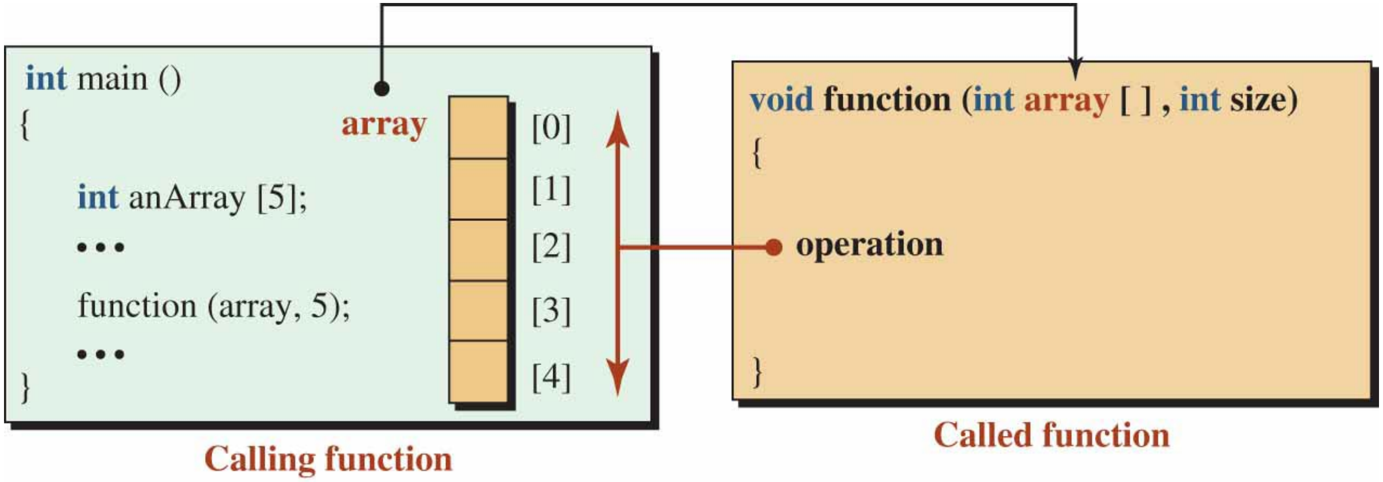
Passing Arrays to Functions

Memory allocation is done only through the calling function, but the called function is allowed to access or modify the element of the array.

The called function needs to know the beginning address and the size of the array using (*int* array []).

The size of the array is defined as a separate parameter.

Figure 8.11 *Passing an array to a function*



Using Functions with Arrays

No Returning Array From Function

C++ does not allow us to return an array from a function. In other words, we cannot have a function prototype such as the following.

```
type [ ] function(const type array [ ], int size); Not allowed
```

When passing arrays to function, we have three choices as shown below:

```
// array will not change.
void function(const type array [ ],  int size);
// array will change.
void function(type array [ ],  int size);
// no change in array1, array 2 is modified version of array1.
void function(const type array1 [ ],  type array2 [ ],  int size);
```

To simulate returning an array from a function, we can use two arrays (one constant and one non-constant).

Using Functions with Arrays

Program 8.8 *Simulating array return by passing two arrays*

```
/******************************************************************
 * Passing two array to a function simulating returning an array. *
 ******************************************************************/
#include <iostream>
using namespace std;
/*****************************************************************
 * Function reverse is a function that takes two arrays. It uses *
 * the first array to reverse the element in the second array.   *
 *****************************************************************/
void reverse (const int array1[], int array2[], int size)
{
     for (int i = 0, j = size - 1; i < size; i++, j--)
     {
         array2 [j] = array1 [i];
     }
     return;
}
/******************************************************************
 * Function print accepts the name and the size of an array.      *
 * It then prints the elements of the array without modifying it. *
```

Using Functions with Arrays

Program 8.8 *Simulating array return by passing two arrays*

```cpp
 ************************************************************/
void print (const int array [], int size)
{
    for (int i = 0; i < size; i++)
    {
        cout << array [i] << " ";
    }
    cout << endl;
    return;
}

int main ( )
{
    // Declaration of two arrays
    int array1 [5] = {150, 170, 190, 110, 130};
    int array2 [5];
    // Calling reverse function to modify array2 to be the reversed of array1
    reverse (array1, array2 , 5);
    // Printing both arrays
    print (array1, 5);
    print (array2, 5);
    return 0;
}
```

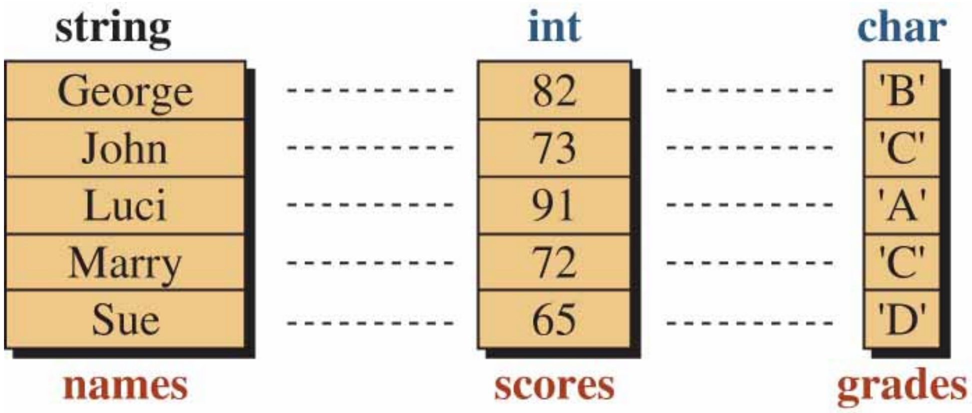
- **Run:**
- 150 170 190 110 130
- 130 110 190 170 150

Parallel Arrays

Sometimes, we need a list in which each row is made of more than one data item, possibly of different types.

For example, we may need to keep name, score, and grade information about each student in a course.

Figure 8.12 *Three parallel arrays of different types*



string	int	char
George	82	'B'
John	73	'C'
Luci	91	'A'
Marry	72	'C'
Sue	65	'D'
names	scores	grades

Note:
The broken line shows associativity among three elements in different arrays.

Arrays and the Student Class Part 1

Program 8.10 *The interface for the Student class*

```
/*****************************************************************
 * This is the interface file for a Student class with three    *
 * private data members and four public member functions.       *
 *****************************************************************/
#ifndef STUDENT_H
#define STUDENT_H
#include <iostream>
#include <string>
using namespace std;

class Student
{
    private:
        string name;
        int score;
        char grade;
    public:
        Student ();
        Student (string name, int score);
        ~Student ();
```

Arrays and the Student Class Part 2

Program 8.10 *The interface for the Student class*

```
    void print();
};
#endif
```

Arrays and the Student Class Part 3

Program 8.11 *The implementation file for the Student class*

```
/*******************************************************************
 * This the implementation for the Student class whose interface *
 * file is given in Program 8-11.                                 *
 ******************************************************************/
#include "student.h"

// Default constructor
Student :: Student()
{
}
// Parameter Constructor
Student :: Student (string nm, int sc)
:name (nm), score (sc)
{
    char temp [ ] = {′F′, ′F′, ′F′, ′F′, ′F′, ′F′, ′D′, ′C′, ′B′, ′A′, ′A′};
    grade = temp [score /10];
}
// Destructor
Student :: ~Student()
{
```

Arrays and the Student Class Part 4

Program 8.11 *The implementation file for the Student class*

```
}
// Print member function
void Student :: print()
{
    cout << setw (12) << left << name;
    cout << setw (8) << right << score;
    cout << setw (8) << right << grade << endl;
}
```

Arrays and Student Class Part 5

Program 8.12 *The application file for Student class*

```
/*****************************************************************
 * The application file to create objects from the Student     *
 * class and print the name, score, and grade of each student  *
 *****************************************************************/
#include "student.h"
#include "iomanip"

int main ( )
{
     // Declaration of an array of Students using default constructors
     Student students [5];
     // Instantiation of five objects using parameter constructors
     students[0] = Student ("George", 82);
     students[1] = Student ("John", 73);
     students[2] = Student ("Luci", 91);
     students[3] = Student ("Mary", 72);
     students[4] = Student ("Sue", 65);
     // Printing students’ name, score, and grade
     for (int i = 0; i < 5; i++)
     {
```

Arrays and Student Class Part 6

Program 8.12 *The application file for Student class*

```
        students[i].print();
        }
        return 0;
    }
```

```
c++ – c students.cpp                     // Compilation of implementation file
c++ – c app.cpp                          // Compilation of application file
c++ – o application student.o app.o      // Linking of two compiled object files
application                              // Running the executable file
```

```
Run:
George    82    B
John      73    C
Luci      91    A
Mary      72    C
Sue       65    D
```

MULTI-DIMENSIONAL ARRAY

Some applications require that a set of values be arranged in a multi-dimensional array.

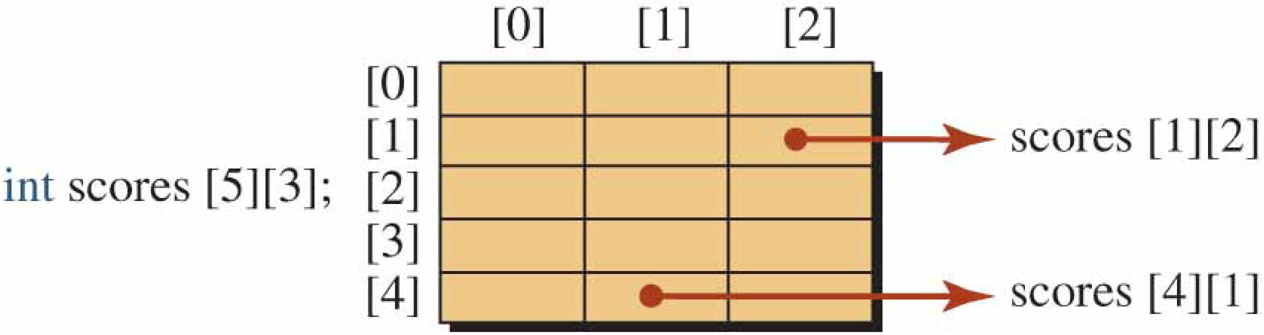
The most common are two-dimensional arrays, but we may encounter three dimensional arrays occasionally.

Two-Dimensional Arrays Part 1

A two-dimensional array defines a structured data type in which two indices are used to define the location of elements in rows and columns.

The first index defines the row; the second index defines the column.

Figure 8.13 *Scores of 5 students in three tests)*



Two-Dimensional Arrays Part 2

Declaration and Initialization

We declare and define a two dimensional array like we did with a one-dimensional array but we have to define two dimensions, rows and columns.

```
int score [5][3];
```

Subscript Operators

In one-dimensional arrays we needed to use a subscript operator.

In two-dimensional arrays, we need to use two subscript operators.

Two-Dimensional Arrays Part 3

Initialization

In one-dimensional arrays we needed to use a subscript operator. In two-dimensional arrays, we need to use two subscript operators.

```
int scores [5][3] = { {82, 65, 72},
                      {73, 70, 80},
                      {91, 76, 40},
                      {72, 72, 68},
                      {65, 90, 80} };
```

To initialize the whole array to zeros (when array is declared locally), we specify only the first value, as shown in the next example.

```
int scores [5][3] = {0};
```

Two-Dimensional Arrays Part 4

Accessing Elements

We can access each element in the array using the exact location of the element defined by the two indexes.

Accessing can be used to store a value in an individual element or retrieve the value of an element.

```
scores[1][0] = 5;        // Storing 5 in row 1 column 0
cin >> scores[2][1];     // Inputting value for row 2 column 1
x = scores [1][2];       // Copying row 1 column 2 into variable x
cout << scores [0][0]; // Outputting value of row 0 column 0
```

Two-Dimensional Arrays Part 5

Passing Two-Dimensional Arrays To Functions

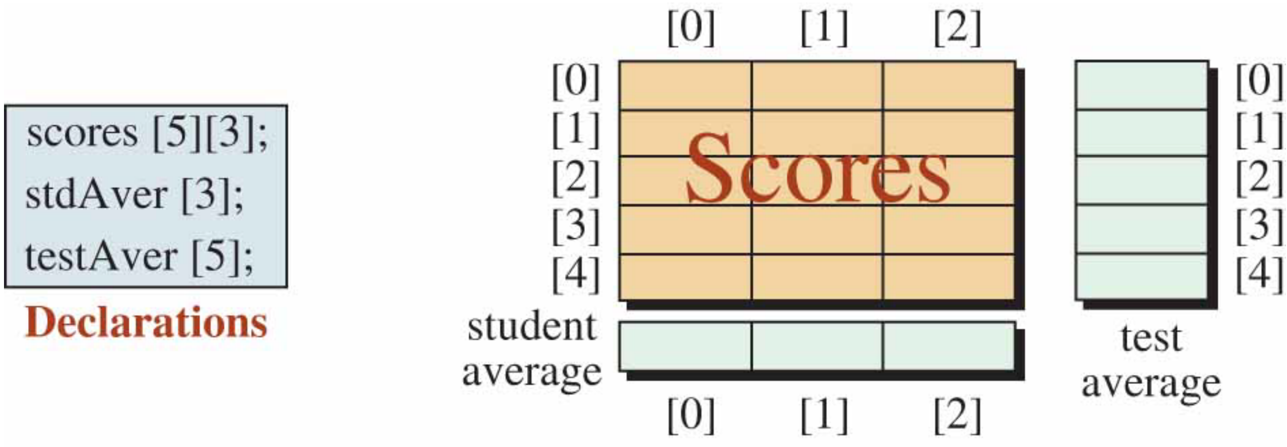
As show below, the first parameter defines the array.

Here, the first bracket is empty, but the second bracket needs to literally define the size of the second dimension.

The size of the first dimension needs to be passed to the array as a separate parameter.

```
void function(int array[ ] [3] , int rowSize);
```

Figure 8.14 *One two-dimension and two one-dimensional array*



Two-Dimensional Arrays Example

Program 8.13 *Using three arrays*

```
/******************************************************************
 * The program creates student average and test average from the *
 * two-dimensional test scores.                                  *
 *****************************************************************/
#include <iostream>
#include <iomanip>
using namespace std;

/****************************************************************
 * The function takes a two-dimensional array of test scores   *
 * for six students in three tests. It then modifies an array  *
 * in main representing student average.                       *
 ***************************************************************/
void findStudentAverage (int const scores [ ][3],
                 double stdAver [ ], int rowSize, int colSize)
{
    for (int i = 0; i < rowSize; i++)
    {
        int sum = 0;
        for (int j = 0; j < colSize; j++)
```

Two-Dimensional Arrays Example

Program 8.13 *Using three arrays*

```
        {
             sum += scores[i][j];
        }
        double average = static_cast <double> (sum) / colSize;
        stdAver[i] = average;
    }
    return;
}
/******************************************************************
 * The function takes a two-dimensional array of test scores    *
 * for six students in three tests. It then modifies an array   *
 * in main representing test averages.                          *
 ******************************************************************/
void findTestAverage (int const scores [][3],
                  double tstAver [], int rowSize , int colSize)
{
    for (int j = 0; j< colSize; j++)
    {
        int sum = 0;
        for (int i = 0; i < rowSize; i++)
```

Two-Dimensional Arrays Example

Program 8.13 *Using three arrays*

```
        {
            sum += scores [i][j];
        }
        double average = static_cast <double> (sum) / rowSize;
        tstAver[j] = average;
    }
}

int main( )
{
    // Declarations of three arrays and some variables
    const int rowSize = 5;
    const int colSize = 3;
    int scores [rowSize][colSize] = {{82, 65, 72},
                                     {73, 70, 80},
                                     {91, 67, 40},
                                     {72, 72, 68},
                                     {65, 90, 80}};
    double stdAver [rowSize];
    double tstAver [colSize];
```

Two-Dimensional Arrays Example

Program 8.13 *Using three arrays*

```
// Calling two functions to modify two average arrays
findStudentAverage (scores, stdAver, rowSize, colSize);
findTestAverage (scores, tstAver, rowSize, colSize);
// Print headings
cout << " Test Scores stdAver" << endl;
cout << " --------------------------- ------- " << endl;
// Print test scores and student averages
for (int i = 0; i < rowSize ; i++)
{
    for (int j = 0 ; j < colSize; j++)
    {
         cout << setw (12) << scores[i][j];
    }
    cout << fixed << setprecision (2) << " " << stdAver[i] << endl;
}
// Print test averages
cout << "tstAver ";
cout << "--------------------------- ";
for (int j = 0 ; j < colSize; j++)
{
```

Two-Dimensional Arrays Example

Program 8.13 *Using three arrays*

```
          cout << fixed << setprecision (2) << stdAver[j] << " ";
     }
     return 0;
}
```

```
Run:
        Test Scores              stdAver
        ------------------------ -------
        82        65        72    73.00
        73        70        80    74.33
        91        67        40    66.00
        72        72        68    70.67
        65        90        80    78.33
tstAver 73.00    74.33      66.00
```

Two-Dimensional Arrays Operations

Operations

Some of the operations we defined previously for a one-dimensional array can be used with two-dimension arrays.

Others need to be modified to be applicable to two-dimensional arrays.

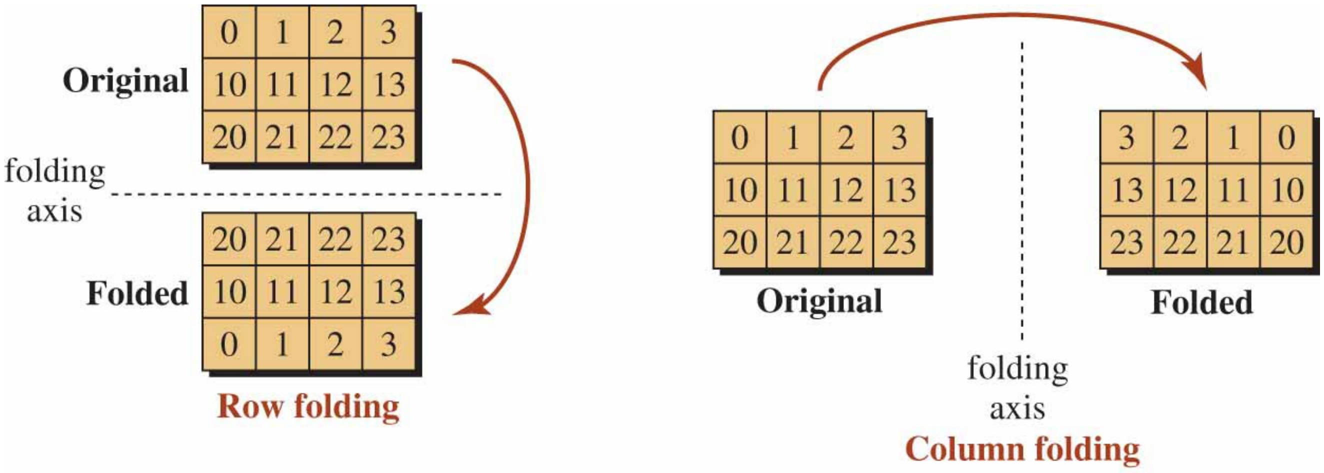
However, there are some operations that can be applied specifically to two dimensional arrays.

Two-Dimensional Arrays Operations

Folding

We can fold a two-dimensional array around a horizontal axis (row folding) or a vertical axis (column folding)

Figure 8.15 ***Folding a two-dimensional array***



Row folding — Original

0	1	2	3
10	11	12	13
20	21	22	23

Row folding — Folded

20	21	22	23
10	11	12	13
0	1	2	3

Column folding — Original

0	1	2	3
10	11	12	13
20	21	22	23

Column folding — Folded

3	2	1	0
13	12	11	10
23	22	21	20

Two-Dimensional Arrays Operations

The following shows how we can use a nested loop to do row folding.

```
for (int i = 0 ; i < rowSize ; i++)
{
    for (int j = 0 ; j < colSize ; j++)
    {
        foldedArray [rowSize -1 - i][j] = originalArray [i][j];
    }
}
```

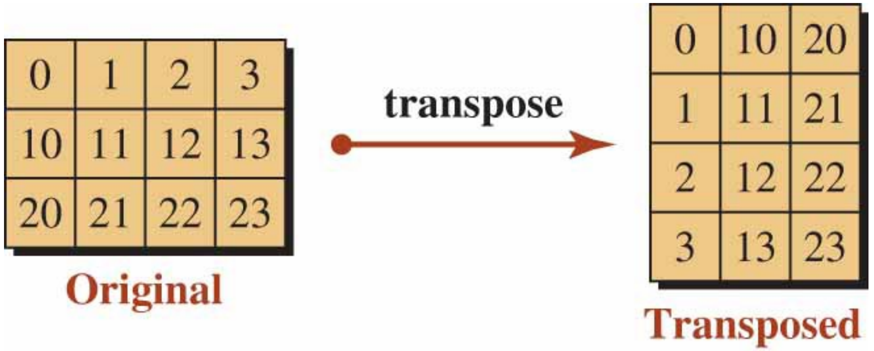
Two-Dimensional Arrays Operations

Transposing

When working with a two-dimensional array, we may need to transpose it.

Transposing means changing the role of the rows and columns

Figure 8.16 *Transpose operation on a two-dimensional array*



0	1	2	3
10	11	12	13
20	21	22	23

Original

0	10	20
1	11	21
2	12	22
3	13	23

Transposed

Two-Dimensional Arrays Operations

The following shows how we can use a nested loop to do transposing.

```
for (int i = 0 ; i < orgRowSize ; i++)
{
    for (int j = 0 ; j < orgColSize ; j++)
    {
        trasposedArray [j][i] = originalArray [i][j];
    }
}
```

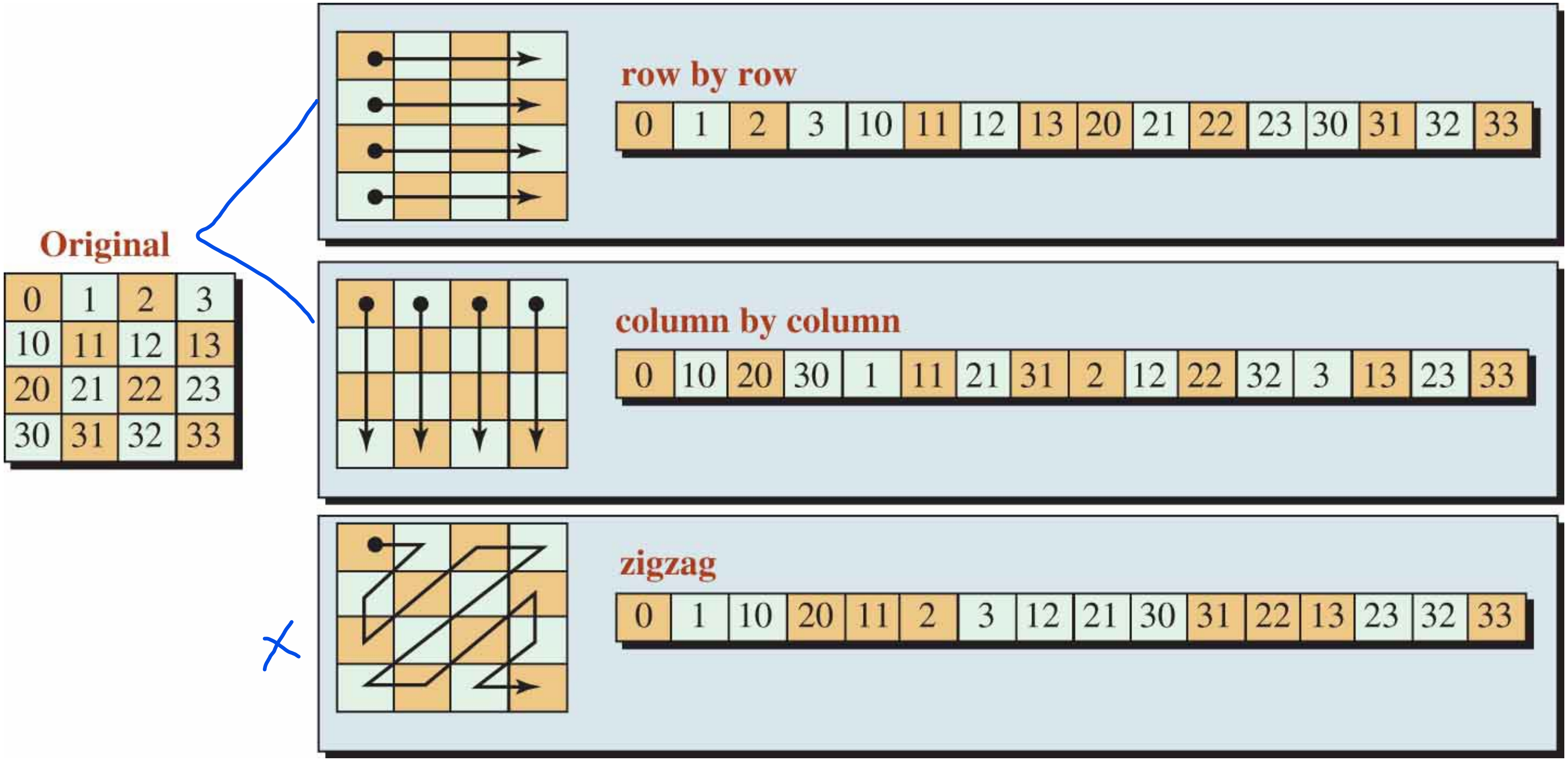
Linearizing

We may need to send the contents of a two-dimensional array through a network (for example, when we send a video).

Before transmission, the array needs to be changed to a one-dimensional array (called linearization).

Two-Dimensional Arrays Operations

Figure 8.17 *Linearizing an array*



Original

0	1	2	3
10	11	12	13
20	21	22	23
30	31	32	33

row by row

0	1	2	3	10	11	12	13	20	21	22	23	30	31	32	33

column by column

0	10	20	30	1	11	21	31	2	12	22	32	3	13	23	33

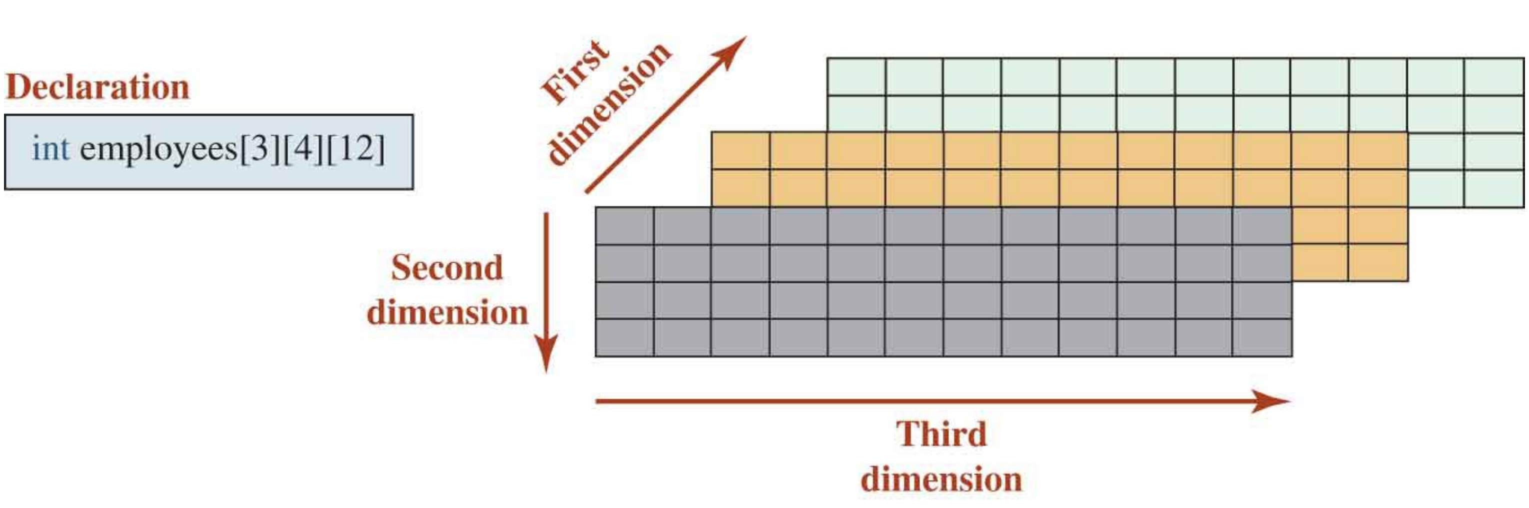
zigzag

0	1	10	20	11	2	3	12	21	30	31	22	13	23	32	33

Three-Dimensional Arrays

C++ does not limit arrays to two dimensions. However, arrays more than three dimensions are rare.

Figure 8.18 *A three-dimensional Array*



Declaration

```
int employees[3][4][12]
```

The figure assumes there is a business that operates in three states, it has up to 4 offices in each state, and there are up to 12 employees in each office.

What’s Next?

Reading Assignment

- ❑ **Read Chap. 9. References, pointers, and Memory Management**
- ❑ **Read Chap. 10. Strings**

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr