SINAI UNIVERSITY

CSW 232
Computer Programming (1)

SPRING 2024

Lecture 08 - Arrays and Strings

Instructor: Dr. Tarek Abdul Hamid

Data Types

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

Arrays

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

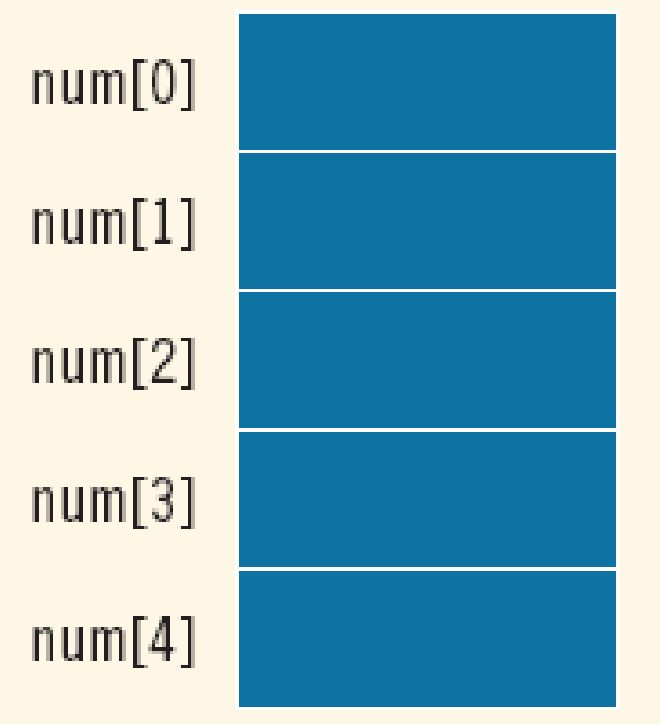
```
dataType arrayName[intExp];
```

`intExp` evaluates to a positive integer

Arrays

- Example:

```
int num[5];
```



Accessing Array Components

- General syntax:

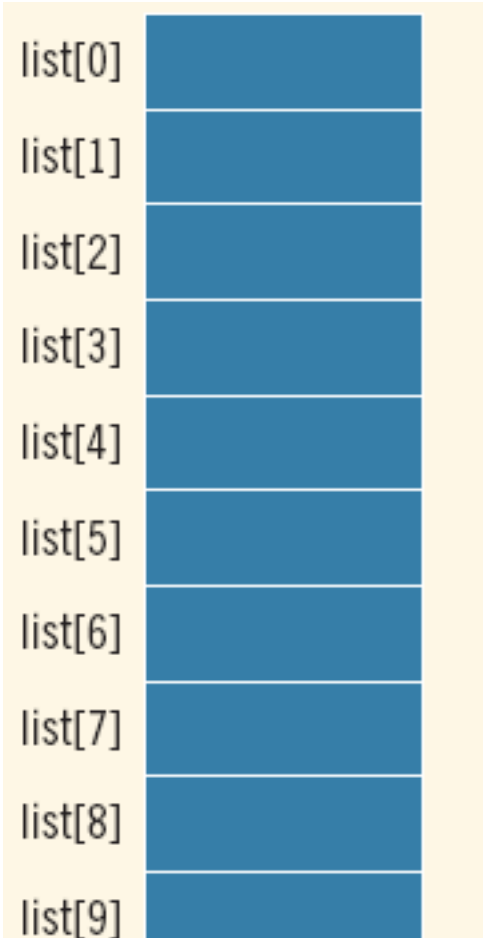
```
arrayName[indexExp]
```

  where `indexExp`, called an **index**, is any expression whose value is a nonnegative integer

- Index value specifies the position of the component in the array
- `[]` is the **array subscripting operator**
- The array index always starts at **0**

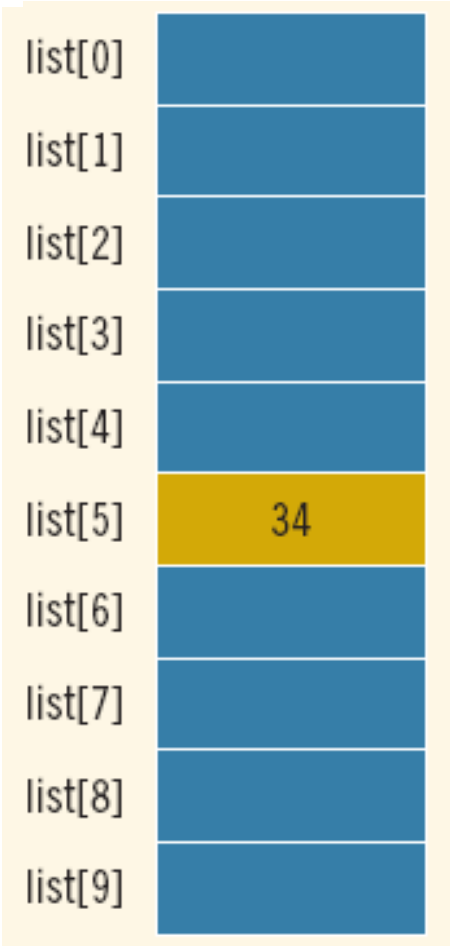
Accessing Array Components

```
int list[10];
```



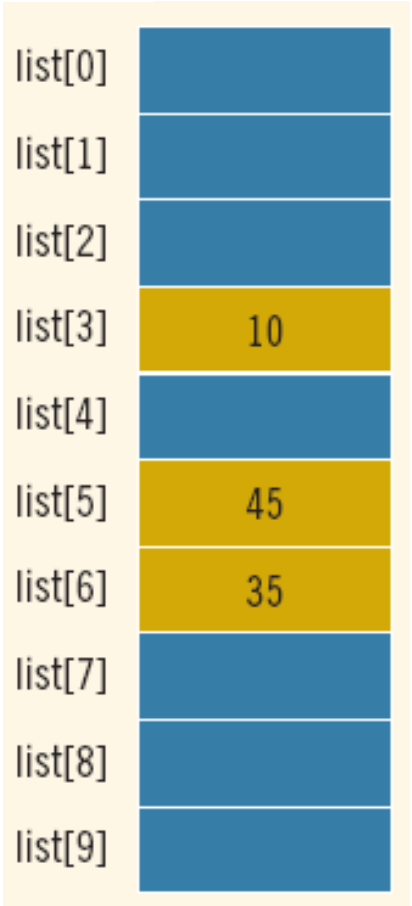
Accessing Array Components

```
list[5] = 34;
```



Accessing Array Components

```
list[3] = 10;
list[6] = 35;
list[5] = list[3] + list[6];
```



Accessing Array Components

You can also declare arrays as follows:

```
const int ARRAY_SIZE = 10;
int list[ARRAY_SIZE];
```

That is, you can first declare a named constant and then use the value of the named constant to declare an array and specify its size.

NOTE

When you declare an array, its size must be known. For example, you cannot do the following:

```
int arraySize;                                  //Line 1

cout << "Enter the size of the array: "; //Line 2
cin >> arraySize;                               //Line 3
cout << endl;                                   //Line 4

int list[arraySize];                            //Line 5; not allowed
```

Processing One-Dimensional Arrays

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

Accessing Array Components

- Consider the declaration

```
int list[100];  //array of size 100
int i;
```

- Using `for` loops to access array elements:

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

- Example:

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

Example

Write a program to :

- **Initialize an array of 10 elements**
- **Reading data into the array**
- **Printing the array elements**
- **Finding the Sum, Average and Max of the array elements.**

Example

```
double sales[10];
int index;
double largestSale, sum, average;
```

Initializing an array:

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

Reading data into an array:

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

Printing an array:

```
for (index = 0; index < 10; index++)
    cout << sales[index] << " ";
```

Finding the sum and average of an array:

```
sum = 0;
for (index = 0; index < 10; index++)
    sum = sum + sales[index];

average = sum / 10;
```

Largest element in the array:

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

Array Index Out of Bounds

- If we have the statements:

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

- The component `num[i]` is valid if `i = 0, 1, 2, 3, 4, 5, 6, 7, 8,` or `9`
- The index of an array is in bounds if the

  `index >=0` and the `index <= ARRAY_SIZE-1`

  - Otherwise, we say the `index` is out of bounds
- In C++, there is no guard against indices that are out of bounds

Array Initialization During Declaration

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

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

Partial Initialization of Arrays During Declaration

- The statement:

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

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

- The statement:

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

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

Partial Initialization of Arrays During Declaration

- The statement:

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

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

- The statement:

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

declares an array of 25 components; initializes `list[0]` to 4 and `list[1]` to 7; all other components are initialized to 0

Some Restrictions on Array Processing

- Consider the following statements:

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

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

```
yourList = myList;  //illegal
```

- Solution:

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

Some Restrictions on Array Processing

- The following is illegal too:

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

- Solution:

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

Arrays as Parameters to Functions

- Arrays are passed by reference only
- The symbol & is *not* used when declaring an array as a formal parameter
- The size of the array is usually omitted
  - If provided, it is ignored by the compiler

Consider the following function:

```
void funcArrayAsParam(int listOne[], double listTwo[])
{
      .
      .
      .
}
```

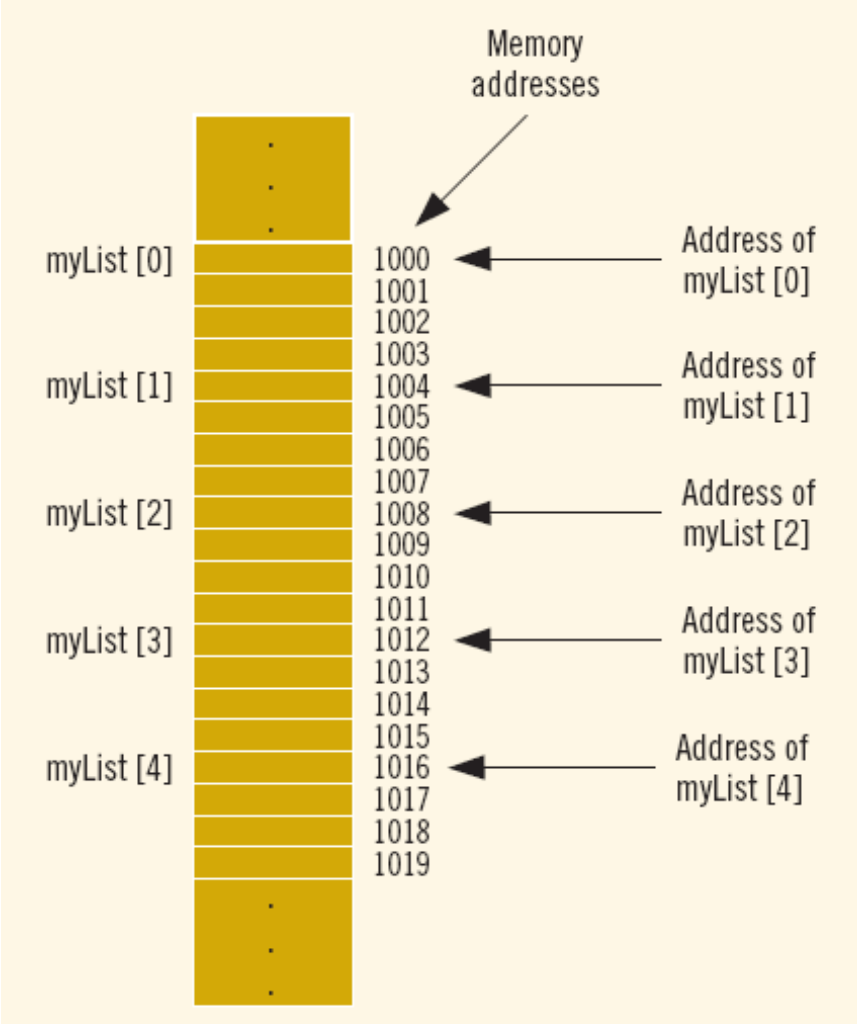
Constant Arrays as Formal Parameters

```
    //Function to initialize an int array to 0.
    //The array to be initialized and its size are passed
    //as parameters. The parameter listSize specifies the
    //number of elements to be initialized.
void initializeArray(int list[], int listSize)
{
    int index;

    for (index = 0; index < listSize; index++)
        list[index] = 0;
}


    //Function to print the elements of an int array.
    //The array to be printed and the number of elements
    //are passed as parameters. The parameter listSize
    //specifies the number of elements to be printed.
void printArray(const int list[], int listSize)
{
    int index;

    for (index = 0; index < listSize; index++)
        cout << list[index] << " ";
}
```

Base Address of an Array and Array in Computer Memory

- The base address of an array is the address, or memory location of the first array component
- If `list` is a one-dimensional array, its base address is the address of `list[0]`
- When we pass an array as a parameter, the base address of the actual array is passed to the formal parameter

Base Address of an Array and Array in Computer Memory



Functions Cannot Return a Value of the Type Array

- C++ does not allow functions to return a value of the type array.

Integral Data Type and Array Indices

- C++ allows any integral type to be used as an array index
- Example:

```
enum paintType {GREEN, RED, BLUE, BROWN, WHITE, ORANGE, YELLOW};
double paintSale[7];
paintType paint;

for (paint = GREEN; paint <= YELLOW;
                    paint = static_cast<paintType>(paint + 1))
    paintSale[paint] = 0.0;

paintSale[RED] = paintSale[RED] + 75.69;
```

Other Ways to Declare Arrays

```
const int NO_OF_STUDENTS = 20;
int testScores[NO_OF_STUDENTS];

const int SIZE = 50;              //Line 1
typedef double list[SIZE];        //Line 2

list yourList;                    //Line 3
list myList;                      //Line 4
```

C-Strings (Character Arrays)

- **Character array:** an array whose components are of type `char`
- C-strings are null-terminated (`'\0'`) character arrays
- Example:
  - `'A'` is the character `A`
  - `"A"` is the C-string `A`
    - `"A"` represents two characters, `'A'` and `'\0'`

C-Strings (Character Arrays)

- Consider the statement

```
char name[16];
```

- Since C-strings are null terminated and `name` has 16 components, the largest string that it can store has 15 characters
- If you store a string of length, say 10 in `name`
  - The first 11 components of `name` are used and the last five are left unused

C-Strings (Character Arrays)

- The statement

```
char name[16] = "John";
```

declares an array `name` of length 16 and stores the C-string `"John"` in it

- The statement

```
char name[] = "John";
```

declares an array `name` of length 5 and stores the C-string `"John"` in it

C-Strings (Character Arrays)

Function	Effect
`strcpy(s1, s2)`	Copies the string `s2` into the string variable `s1` The length of `s1` should be at least as large as `s2`
`strcmp(s1, s2)`	Returns a value < 0 if `s1` is less than `s2` Returns 0 if `s1` and `s2` are the same Returns a value > 0 if `s1` is greater than `s2`
`strlen(s)`	Returns the length of the string `s`, excluding the null character

String Comparison

- C-strings are compared character by character using the collating sequence of the system
- If we are using the ASCII character set
  - `"Air" < "Boat"`
  - `"Air" < "An"`
  - `"Bill" < "Billy"`
  - `"Hello" < "hello"`

Example

Suppose you have the following statements:

```
char studentName[21];
char myname[16];
char yourname[16];
```

The following statements show how string functions work:

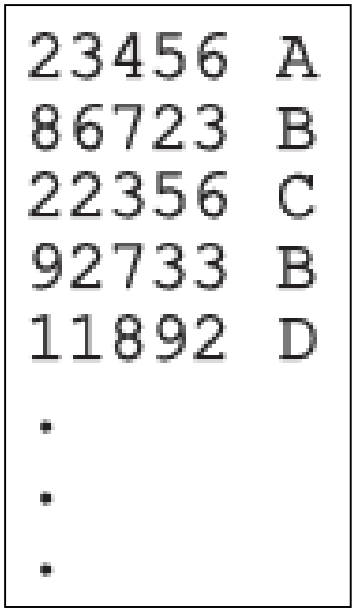
Statement	Effect
`strcpy(myname, "John Robinson");`	`Myname = "John Robinson"`
`strlen("John Robinson");`	Returns 13, the length of the string `"John Robinson"`
`int len;` `len = strlen("Sunny Day");`	Stores 9 into `len`
`strcpy(yourname, "Lisa Miller");` `strcpy(studentName, yourname);`	`yourname = "Lisa Miller"` `studentName = "Lisa Miller"`
`strcmp("Bill", "Lisa");`	Returns a value < 0

`strcpy(yourname, "Kathy Brown");`
`strcpy(myname, "Mark G. Clark");`
`strcmp(myname, yourname);`
	`yourname = "Kathy Brown"`
	`myname = "Mark G. Clark"`
	Returns a value > 0

Parallel Arrays

- Two (or more) arrays are called parallel if their corresponding components hold related information
- Example:

```
int studentId[50];
char courseGrade[50];
```



Two-Dimensional Arrays

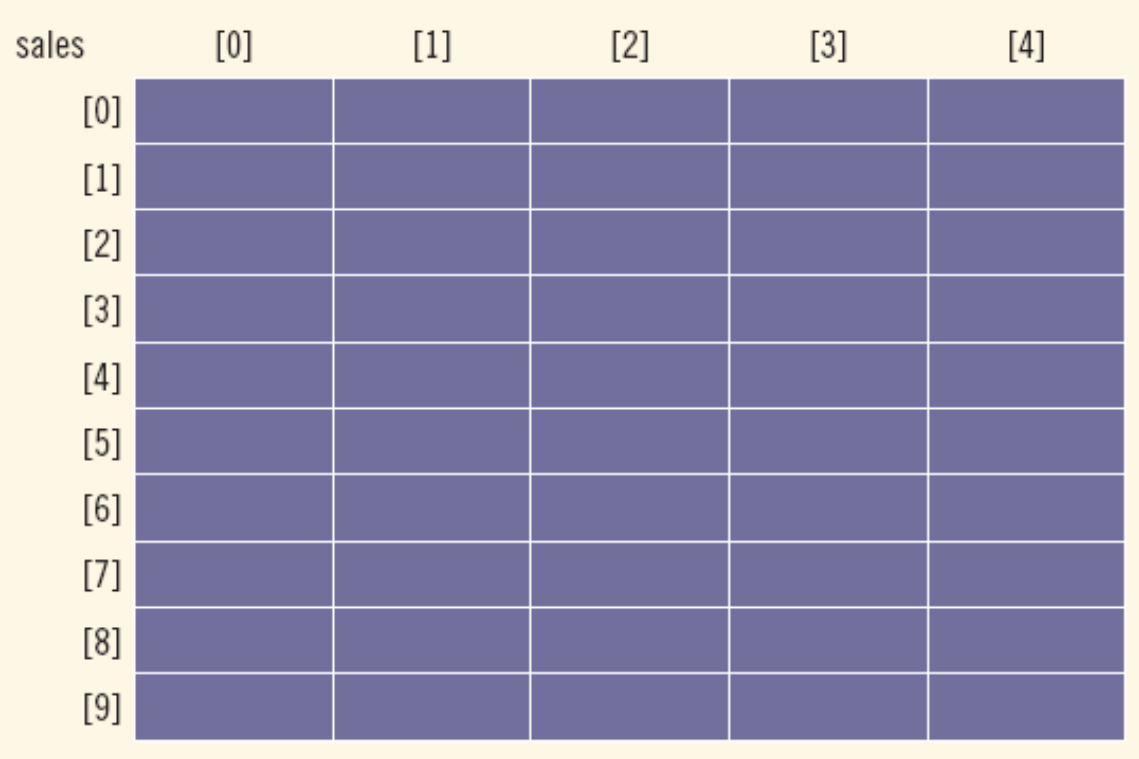
- Two-dimensional array: collection of a fixed number of components (of the same type) arranged in two dimensions
  - Sometimes called matrices or tables
- Declaration syntax:

```
dataType  arrayName[intExp1][intExp2];
```

where `intexp1` and `intexp2` are expressions yielding positive integer values, and specify the number of rows and the number of columns, respectively, in the array

Two-Dimensional Arrays

```
double sales[10][5];
```



Accessing Array Components

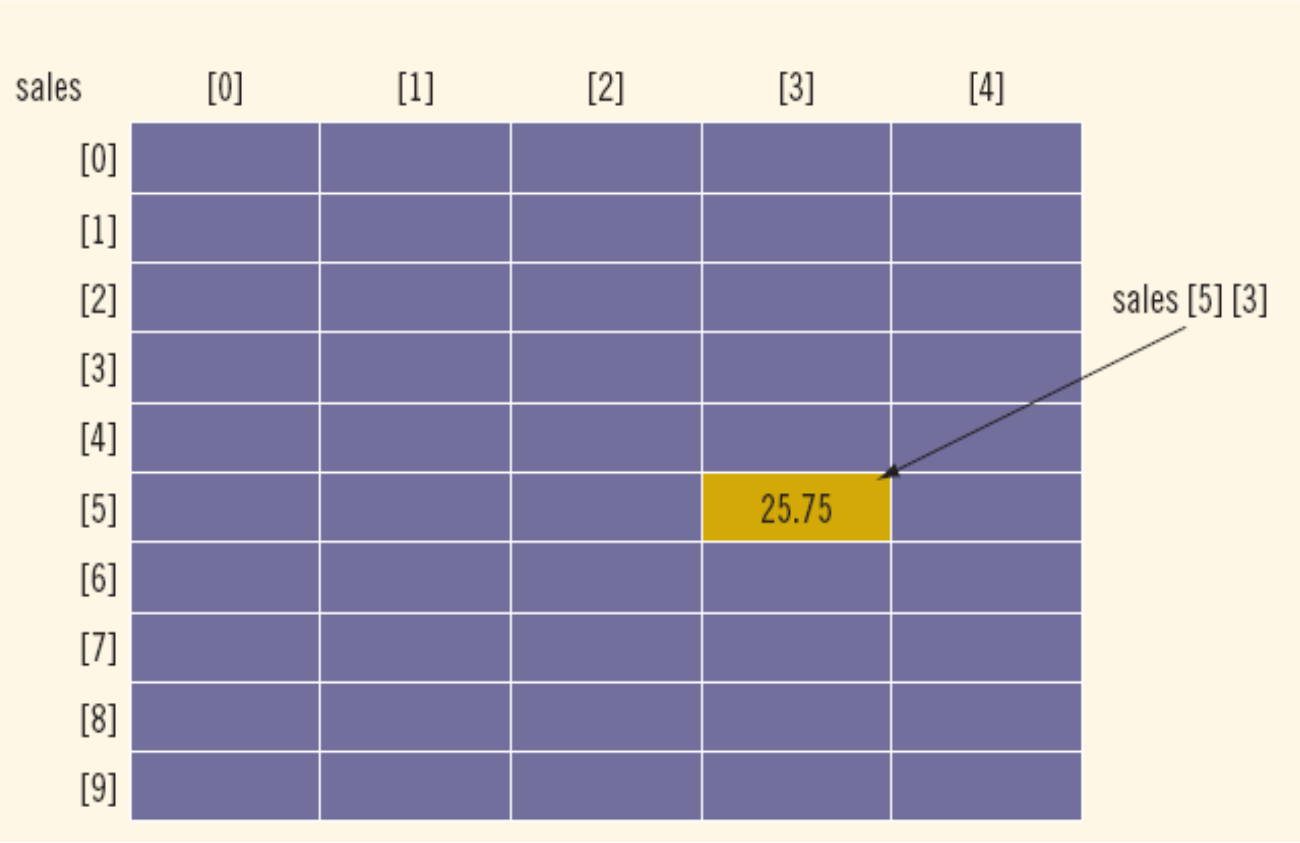
- Syntax:

```
arrayName[indexExp1][indexExp2]
```

where `indexexp1` and `indexexp2` are expressions yielding nonnegative integer values, and specify the row and column position

Accessing Array Components

```
sales[5][3] = 25.75;
```



Two-Dimensional Array Initialization During Declaration

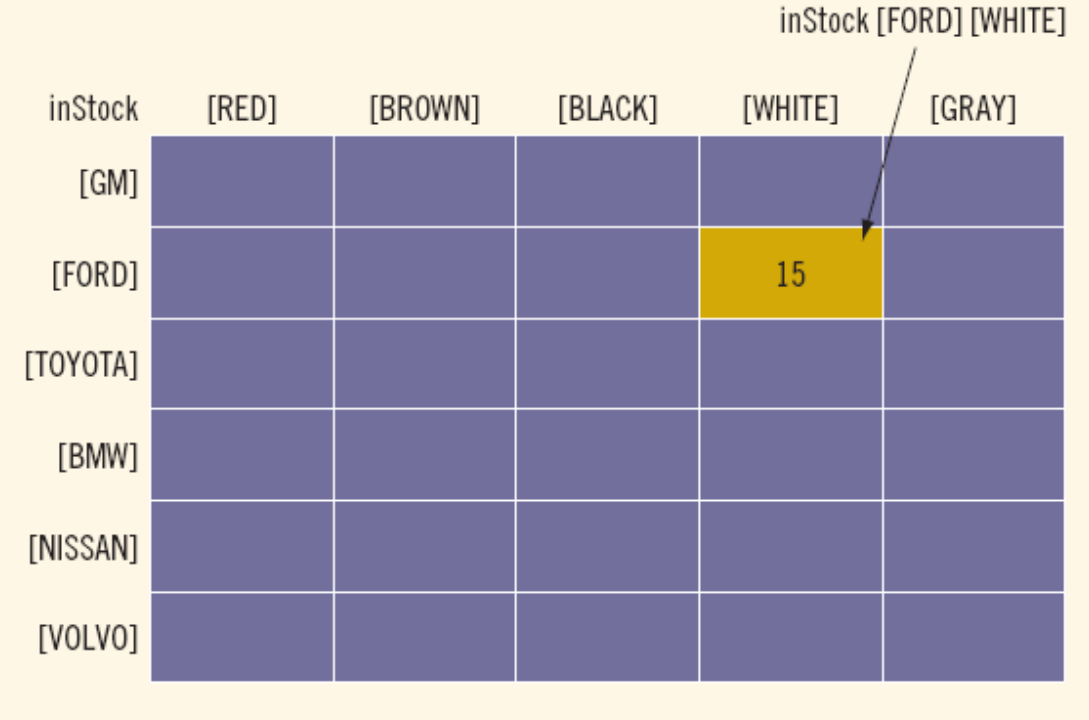
- Two-dimensional arrays can be initialized when they are declared:

```
int board[4][3] = {{2, 3, 1},
                   {15, 25, 13},
                   {20, 4, 7},
                   {11, 18, 14}};
```

  - Elements of each **row** are enclosed within braces and separated by commas
  - All rows are enclosed within braces
  - For number arrays, if all components of a row aren't specified, unspecified ones are set to 0

Two-Dimensional Arrays and Enumeration Types

```
enum carType {GM, FORD, TOYOTA, BMW, NISSAN, VOLVO};
enum colorType {RED, BROWN, BLACK, WHITE, GRAY};

int inStock[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
 inStock[FORD][WHITE] = 15;
```

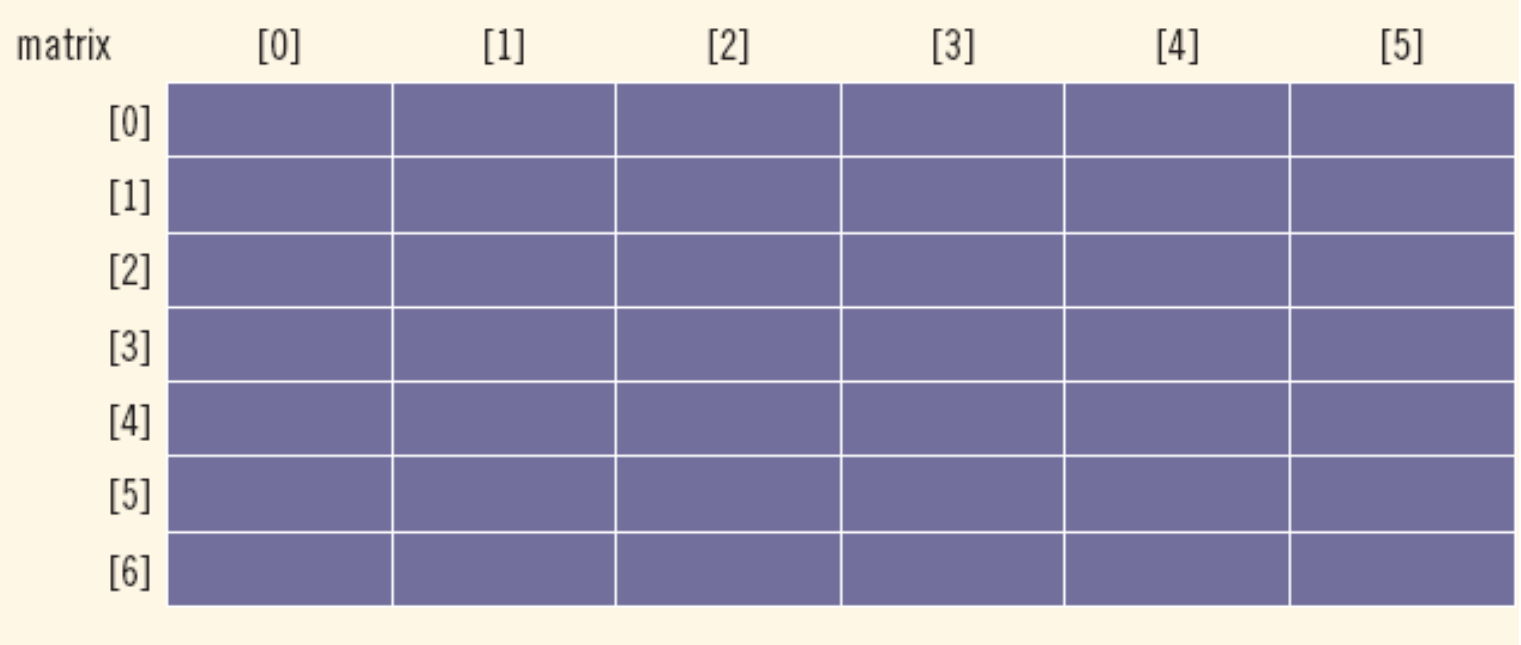


Processing Two-Dimensional Arrays

- Ways to process a two-dimensional array:
  - Process the entire array
  - Process a particular row of the array, called row processing
  - Process a particular column of the array, called column processing
- Each row and each column of a two-dimensional array is a one-dimensional array
  - To process, use algorithms similar to processing one-dimensional arrays

Processing Two-Dimensional Arrays

```
const int NUMBER_OF_ROWS = 7;     //This can be set to any number.
const int NUMBER_OF_COLUMNS = 6; //This can be set to any number.

int matrix[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
int row;
int col;
int sum;
int largest;
int temp;
```



Initialization

- To initialize row number 4 (i.e., fifth row) to 0

```
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    matrix[row][col] = 0;
```

- To initialize the entire matrix to 0:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        matrix[row][col] = 0;
```

Print

- To output the components of `matrix`:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cout << setw(5) << matrix[row][col] << " ";

    cout << endl;
}
```

Input

- To input data into each component of `matrix`:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        cin >> matrix[row][col];
```

Sum by Row

- To find the sum of row number 4 of `matrix`:

```
sum = 0;
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    sum = sum + matrix[row][col];
```

- To find the sum of each individual row:

```
  //Sum of each individual row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    sum = 0;
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        sum = sum + matrix[row][col];

    cout << "Sum of row " << row + 1 << " = " << sum << endl;
}
```

Sum by Column

- To find the sum of each individual column:

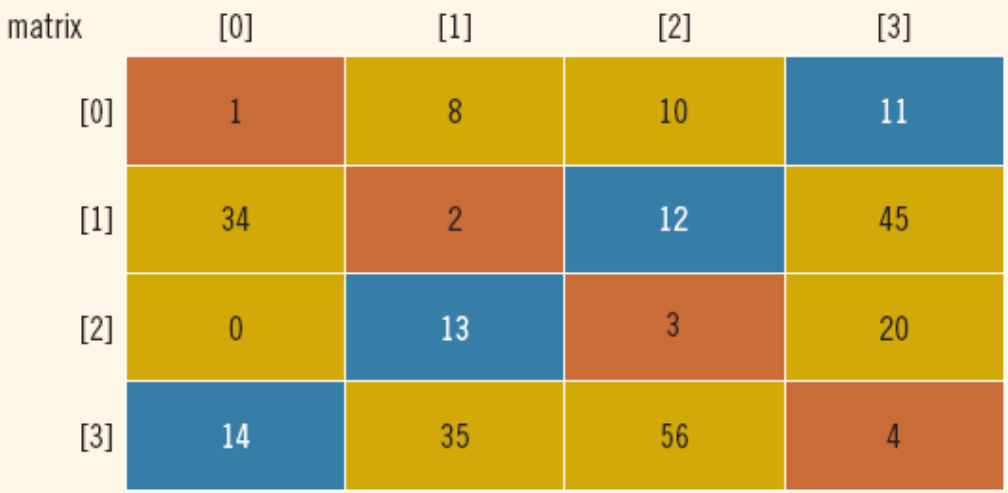
```
  //Sum of each individual column
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
{
     sum = 0;
     for (row = 0; row < NUMBER_OF_ROWS; row++)
          sum = sum + matrix[row][col];

     cout << "Sum of column " << col + 1 << " = " << sum
          << endl;
}
```

Largest Element in Each Row and Each Column

```
  //Largest element in each row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    largest = matrix[row][0]; //Assume that the first element
                              //of the row is the largest.
    for (col = 1; col < NUMBER_OF_COLUMNS; col++)
        if (largest < matrix[row][col])
            largest = matrix[row][col];

    cout << "The largest element in row " << row + 1 << " = "
         << largest << endl;
}

  //Largest element in each column
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
{
    largest = matrix[0][col]; //Assume that the first element
                              //of the column is the largest.
    for (row = 1; row < NUMBER_OF_ROWS; row++)
        if (largest < matrix[row][col])
            largest = matrix[row][col];

    cout << "The largest element in column " << col + 1
         << " = " << largest << endl;
}
```

Reversing Diagonal

- Before:



matrix	[0]	[1]	[2]	[3]
[0]	1	8	10	11
[1]	34	2	12	45
[2]	0	13	3	20
[3]	14	35	56	4

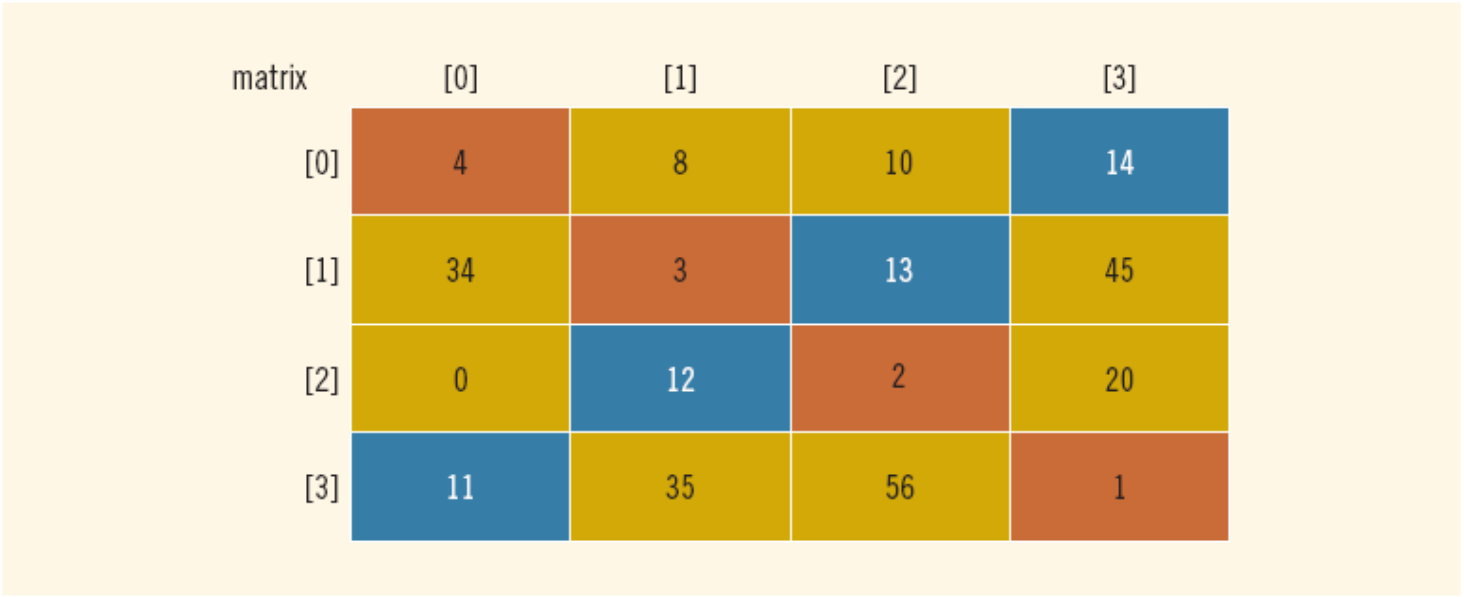
Reversing Diagonal

- To reverse both the diagonals:

```
  //Reverse the main diagonal
for (row = 0; row < NUMBER_OF_ROWS / 2; row++)
{
    temp = matrix[row][row];
    matrix[row][row] =
      matrix[NUMBER_OF_ROWS - 1 - row][NUMBER_OF_ROWS - 1 - row];
    matrix[NUMBER_OF_ROWS - 1 - row][NUMBER_OF_ROWS - 1 - row]
          = temp;
}

  //Reverse the opposite diagonal
for (row = 0; row < NUMBER_OF_ROWS / 2; row++)
{
    temp = matrix[row][NUMBER_OF_ROWS - 1 - row];
    matrix[row][NUMBER_OF_ROWS - 1 - row] =
                    matrix[NUMBER_OF_ROWS - 1 - row][row];
    matrix[NUMBER_OF_ROWS - 1 - row][row] = temp;
}
```

Reversing Diagonal

- After:



matrix	[0]	[1]	[2]	[3]
[0]	4	8	10	14
[1]	34	3	13	45
[2]	0	12	2	20
[3]	11	35	56	1

Arrays of Strings

- Strings in C++ can be manipulated using either the data type string or character arrays (C-strings)
- On some compilers, the data type `string` may not be available in Standard C++ (i.e., non-ANSI/ISO Standard C++)

Arrays of Strings and the `string` Type

- To declare an array of 100 components of type `string`:

```
string list[100];
```

- Basic operations, such as assignment, comparison, and input/output, can be performed on values of the `string` type
- The data in `list` can be processed just like any one-dimensional array

Arrays of Strings and C-Strings (Character Arrays)

```
char list[100][16];
strcpy(list[1], "Snow White");
```

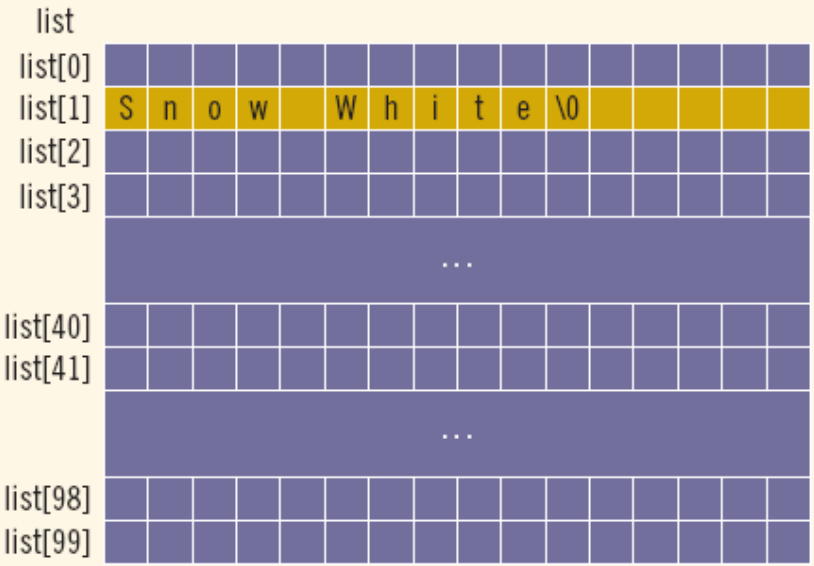


FIGURE 9-17 Array `list`, showing `list[1]`

```
for (j = 0; j < 100; j++)
    cin.get(list[j], 16);
```

Another Way to Declare a Two-Dimensional Array

- Consider the following:

```
const int NUMBER_OF_ROWS = 20;
const int NUMBER_OF_COLUMNS = 10;

typedef int tableType[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
```

- To declare an array of 20 rows and 10 columns:

```
tableType matrix;
```

Multidimensional Arrays

- Multidimensional array: collection of a fixed number of elements (called components) arranged in *n* dimensions (n >= 1)
  - Also called an *n*-dimensional array
- Declaration syntax:

```
dataType arrayName[intExp1][intExp2] ... [intExpn];
```

- To access a component:

```
arrayName[indexExp1][indexExp2] ... [indexExpn]
```

Multidimensional Arrays

- When declaring a multidimensional array as a formal parameter in a function
  - Can omit size of first dimension but not other dimensions
- As parameters, multidimensional arrays are passed by reference only
- A function cannot return a value of the type array
- There is no check if the array indices are within bounds


Thanks!