COMSATS University
Islamabad
(Lahore Campus)

CSC103- Programming Fundamentals

MS. MAHWISH WAQAS

MAWISH.WAQAS@CUILAHORE.EDU.PK

Chapter 8
Arrays and Strings

Auto Declaration and Range-Based `For` Loops

- C++11 allows auto declaration of variables
  - Data type does not need to be specified

```
auto num = 15;    // num is assumed int
```

```
for (dataType identifier : arrayName)
    statements
```

- Range-based `for` loop

For example, suppose you have the following declarations:

```
double list[25];
double sum;
```

```
sum = 0;
for (double num : list) // For each num
    sum = sum + num;    // in list
```

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) (cont'd.)

- Example:

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

- Since C-strings are null terminated and `name` has 16 components, the largest string it can store has 15 characters
- If you store a string whose length is less than the array size, the last components are unused

C-Strings (Character Arrays) (cont'd.)

- Size of an array can be omitted if the array is initialized during declaration
- Example:

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

  - Declares an array of length 5 and stores the C-string "John" in it
- Useful string manipulation functions
  - `strcpy`, `strcmp`, and `strlen`

String Comparison

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

Reading and Writing Strings

- Most rules for arrays also apply to C-strings (which are character arrays)
- Aggregate operations, such as assignment and comparison, are not allowed on arrays
- C++ does allow aggregate operations for the input and output of C-strings

String Input

- Example:

```
cin >> name;
```

  - Stores the next input C-string into name

- To read strings with blanks, use `get` function:

```
cin.get(str, m+1);
```

  - Stores the next `m` characters into `str` but the newline character is not stored in `str`
  - If input string has fewer than `m` characters, reading stops at the newline character

String Output

- Example:

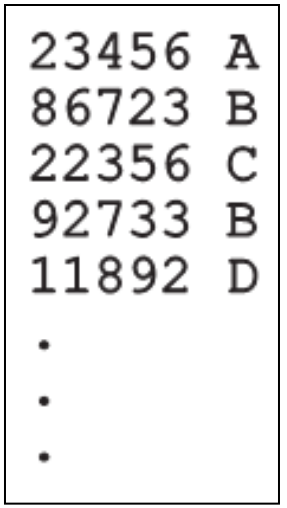
```
cout << name;
```

  - Outputs the content of name on the screen
  - `<<` continues to write the contents of `name` until it finds the null character
  - If `name` does not contain the null character, then strange output may occur
    - `<<` continues to output data from memory adjacent to `name` until a `'\0'` is found

Parallel Arrays

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

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



Two- and Multidimensional 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];
```

  - `intExp1` and `intExp2` are expressions with positive integer values specifying the number of rows and columns in the array

Accessing Array Components

- Accessing components in a two-dimensional array:

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

  - Where `indexExp1` and `indexExp2` are expressions with positive integer values, and specify the row and column position

- Example:

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

Accessing Array Components (cont'd.)

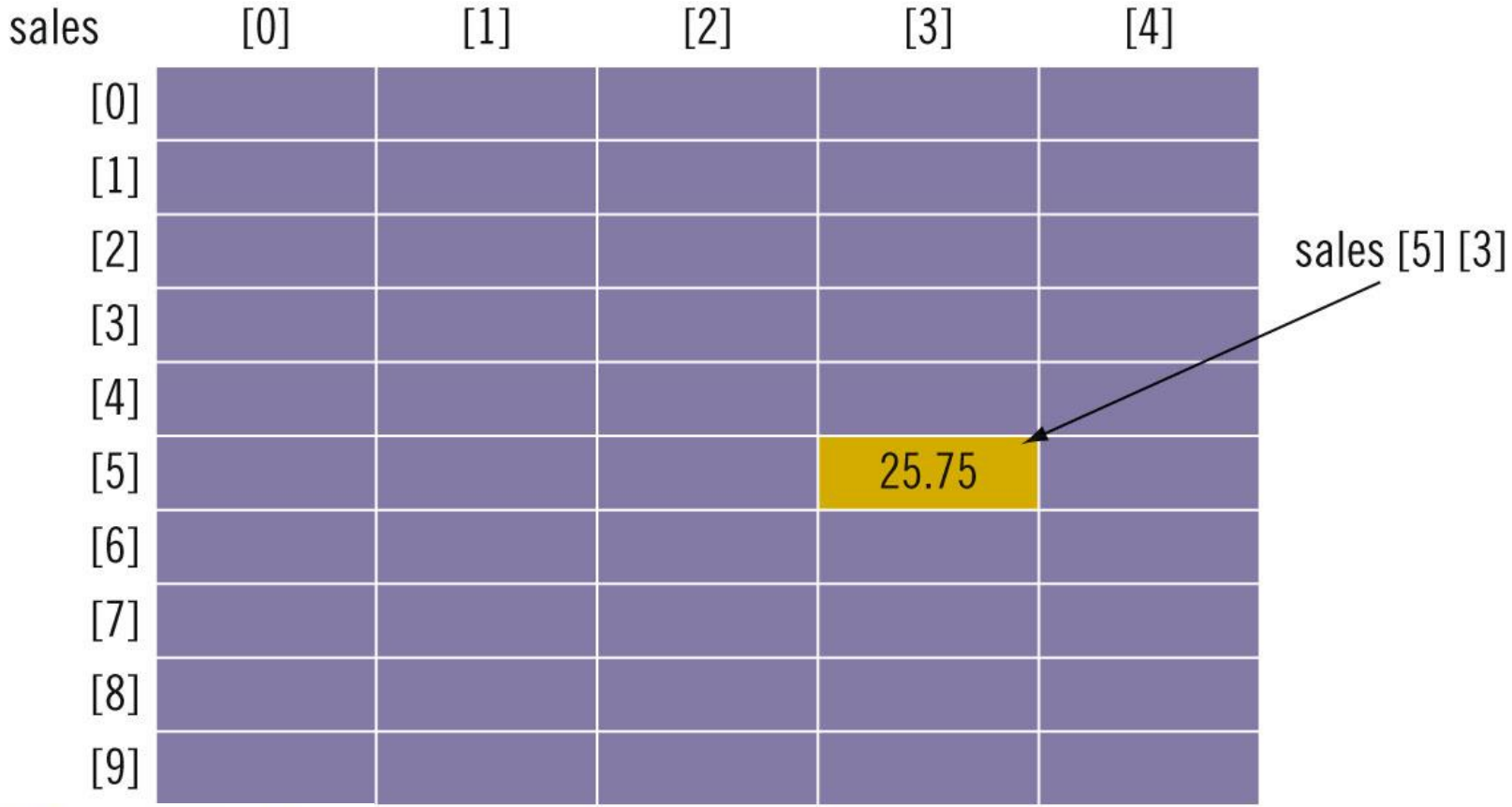


FIGURE 8-14 sales[5][3]

Two-Dimensional Array Initialization During Declaration

- Two-dimensional arrays can be initialized when they are declared:
  - Elements of each row are enclosed within braces and separated by commas
  - All rows are enclosed within braces
  - For number arrays, unspecified elements are set to 0

Two-Dimensional Arrays and Enumeration Types

- Enumeration types can be used for array indices:

```
const int NUMBER_OF_ROWS = 6;
const int NUMBER_OF_COLUMNS = 5;
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;
```

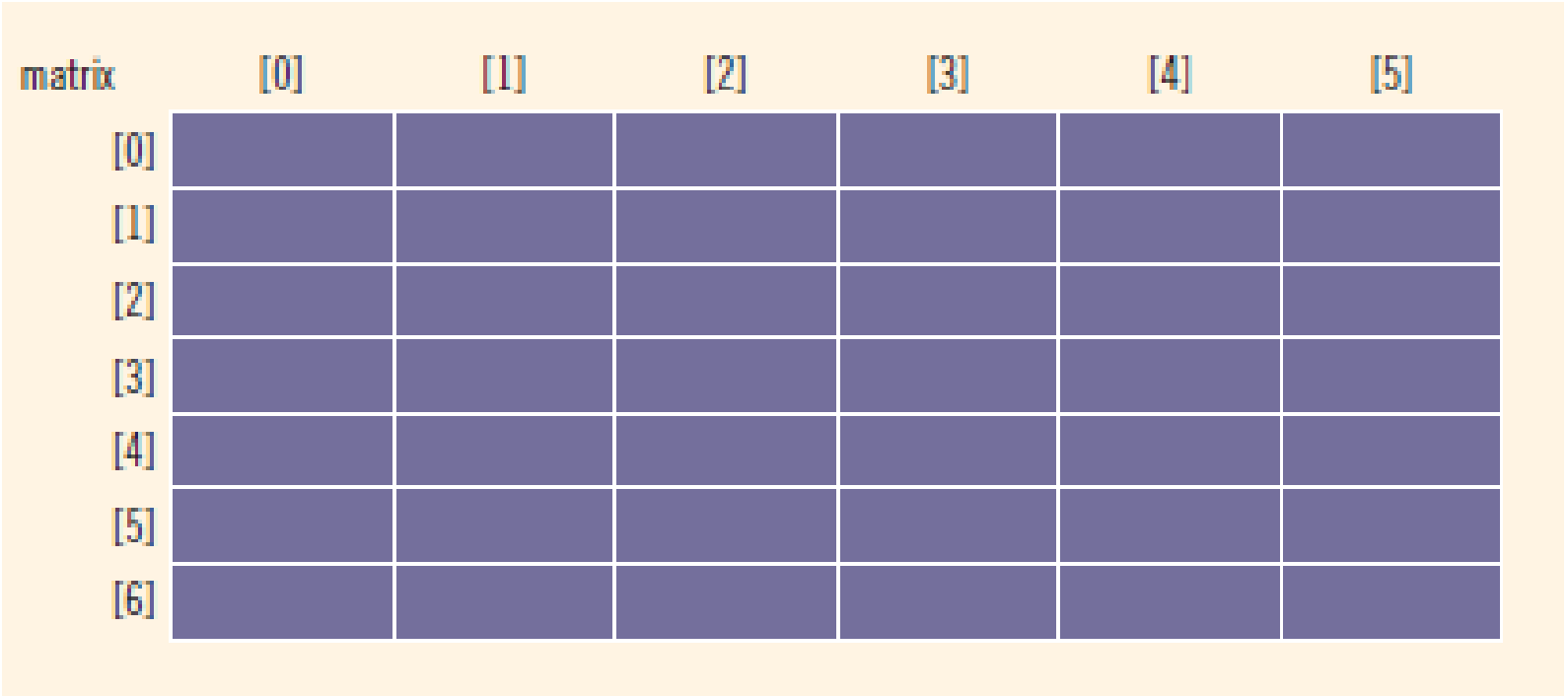
inStock	[RED]	[BROWN]	[BLACK]	[WHITE]	[GRAY]
[GM]					
[FORD]					
[TOYOTA]					
[BMW]					
[NISSAN]					
[VOLVO]					

Processing Two-Dimensional Arrays

- Ways to process a two-dimensional array:
  - Process entire array
  - Row processing: process a single row at a time
  - Column processing: process a single column at a time
- 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

```
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

```
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;
```

- Examples:
  - To initialize row number 4 (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

- Use a nested loop to output the components of a two dimensional array:

```
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

- Examples:
  - To input into row number 4 (fifth row):

```
row = 4;

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

  - 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

- Example:
  - To find the sum of row number 4:

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

Sum by Column

- Example:
  - 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

- Example:
  - To find the largest element in each row:

```
  //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;
}
```

Passing Two-Dimensional Arrays as Parameters to Functions

- Two-dimensional arrays are passed by reference as parameters to a function
  - Base address is passed to formal parameter
- Two-dimensional arrays are stored in row order
- When declaring a two-dimensional array as a formal parameter, can omit size of first dimension, but not the second

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)

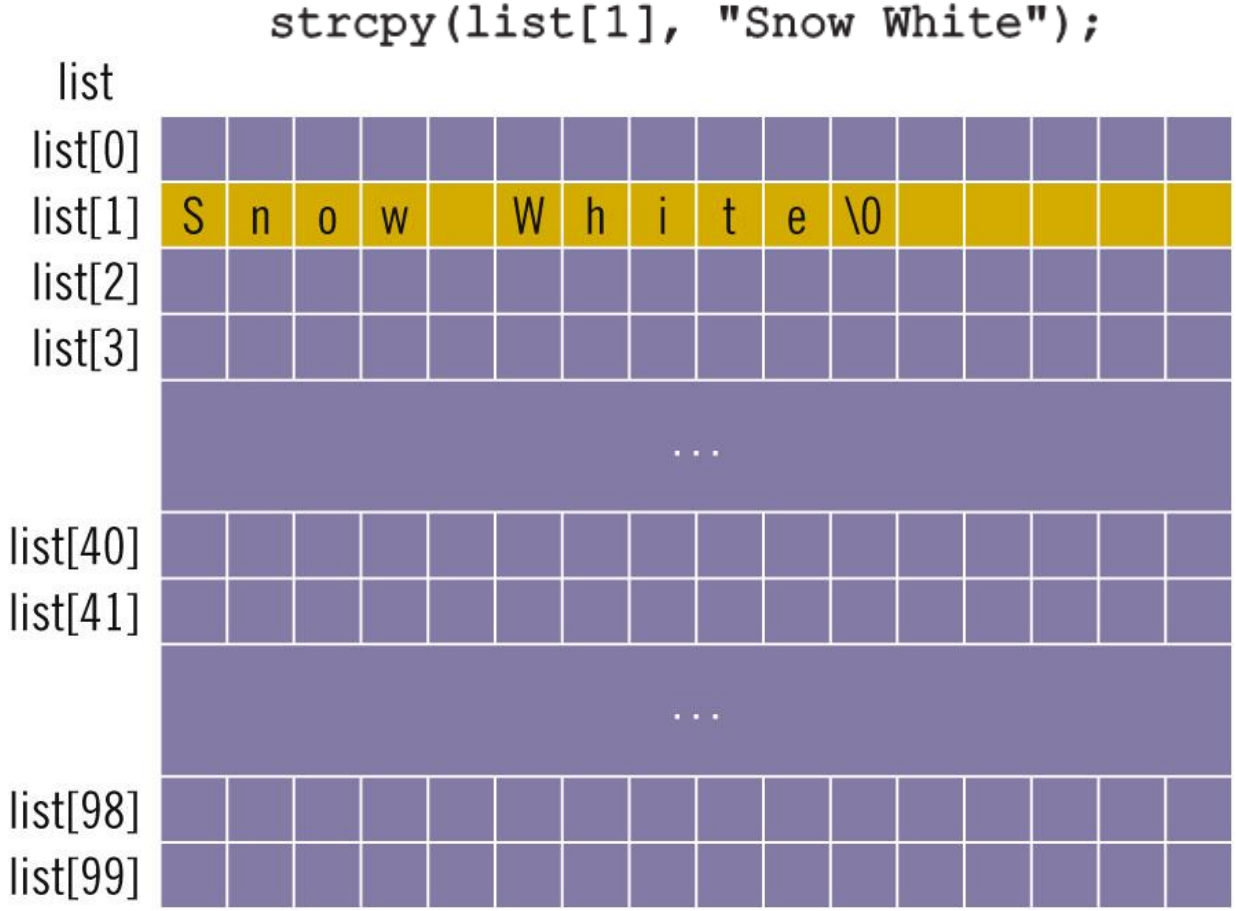


FIGURE 8-20 Array list, showing list[1]

Multidimensional Arrays

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

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

- To access a component:

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

Summary

- Array: structured data type with a fixed number of components of the same type
  - Components are accessed using their relative positions in the array
- Elements of a one-dimensional array are arranged in the form of a list
- An array index can be any expression that evaluates to a nonnegative integer
  - Must always be less than the size of the array

Summary (cont'd.)

- The base address of an array is the address of the first array component
- When passing an array as an actual parameter, use only its name
  - Passed by reference only
- A function cannot return an array type value
- C++11 allows auto declaration of variables

Summary (cont'd.)

- In C++, C-strings are null terminated and are stored in character arrays
- Commonly used C-string manipulation functions include:
  - `strcpy`, `strcmp`, and `strlen`
- Parallel arrays hold related information
- In a two-dimensional array, the elements are arranged in a table form

Summary (cont'd.)

- To access an element of a two-dimensional array, you need a pair of indices:
  - One for row position, one for column position
- In row processing, a two-dimensional array is processed one row at a time
- In column processing, a two-dimensional array is processed one column at a time