Module 3

Arrays and Strings
Objectives

• In this chapter, you will:
  – Learn about arrays
  – Declare and manipulate data into arrays
  – Learn about “array index out of bounds”
  – Learn about the restrictions on array processing
  – Pass an array as a parameter to a function
Objectives (cont’d.)

– Learn about C-strings
– Input data into—and output data from—a C-string
– String manipulation
– Manipulate data in a two-dimensional array
– Learn about multidimensional arrays
Arrays

- **Array**: a collection of a fixed number of components, all of the same data type
- One-dimensional array: components are arranged in a list form
- Syntax for declaring a one-dimensional array:
  
  ```
  dataType arrayName[intExp];
  ```

- **intExp**: any constant expression that evaluates to a positive integer
Accessing Array Components

• General syntax:

\[
arrayName[indexExp]
\]

• \textit{indexExp}: called the \textit{index}
  – An expression with a nonnegative integer value
• Value of the index is the position of the item in the array
• \([\ ]\): \textit{array subscripting operator}
  – Array index always starts at 0
Processing One-Dimensional Arrays

• Basic operations on a one-dimensional array:
  – Initializing
  – Inputting data
  – Outputting data stored in an array
  – Finding the largest and/or smallest element

• Each operation requires ability to step through elements of the array
  – Easily accomplished by a loop
Processing One-Dimensional Arrays (cont’d.)

- Given the declaration:

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

- Use a `for` loop to access array elements:

```cpp
for (i = 0; i < 100; i++) //Line 1
    cin >> list[i];       //Line 2
```
Array Index Out of Bounds

- Index of an array is in bounds if the index is $\geq 0$ and $\leq \text{ARRAY\_SIZE}-1$
  - Otherwise, 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
  - Values are placed between curly braces
  - 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:
  ```cpp
  int list[10] = {0};
  ```
  – Declares an array of 10 components and initializes all of them to zero

• The statement:
  ```cpp
  int list[10] = {8, 5, 12};
  ```
  – Declares an array of 10 components and initializes
    `list[0]` to 8, `list[1]` to 5, `list[2]` to 12
    – All other components are initialized to 0
Some Restrictions on Array Processing

• **Aggregate operation**: any operation that manipulates the entire array as a single unit
  – Not allowed on arrays in C++

• Example:

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

• Solution:

```cpp
for (int index = 0; index < 5; index ++)
    yourList[index] = myList[index];
```
Arrays as Parameters to Functions

- Arrays are passed by reference only
- Do not use symbol & when declaring an array as a formal parameter
- Size of the array is usually omitted
  - If provided, it is ignored by the compiler
- Example:

```c++
void funcArrayAsParam(int listOne[], double listTwo[])
```
Constant Arrays as Formal Parameters

• Can prevent a function from changing the actual parameter when passed by reference
  – Use `const` in the declaration of the formal parameter

• Example:

```c
void example(int x[], const int y[], int sizeX, int sizeY)
```
Base Address of an Array and Array in Computer Memory

• **Base address** of an array: address (memory location) of the first array component

• Example:
  – If `list` is a one-dimensional array, its base address is the address of `list[0]`

• When an array is passed as a parameter, the base address of the actual array is passed to the formal parameter
Functions Cannot Return a Value of the Type Array

• C++ does not allow functions to return a value of type array
Other Ways to Declare Arrays

• Examples:

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

```c
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:

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

  – Declares an array of length 5 and stores the C-string "John" in it
String Operations

• `#include <cstring>`
• Examples of operations:
  – Assignment: `strcpy()`
  – Concatenation: `strcat()`
  – Comparison: `strcmp()`
  – Length: `strlen()`
```cpp
#include <iostream>
#include <cstring>
using namespace std;
const int MAX_LENGTH=100;
int main()
{
    char string1[MAX_LENGTH];
    char string2[MAX_LENGTH];
    strcpy(string1, "Hello World!");
    strcpy(string2, string1);
    return 0;
}
```

string1: <garbage>
string2: <garbage>
```cpp
#include <iostream>
#include <cstring>
using namespace std;
const int MAX_LENGTH=100;
int main()
{
    char string1[MAX_LENGTH];
    char string2[MAX_LENGTH];
    strcpy(string1, "Hello World!");
    strcpy(string2, string1);
    return 0;
}

string1: "Hello World!"
string2: <garbage>
```
```cpp
#include <iostream>
#include <cstring>
using namespace std;
const int MAX_LENGTH=100;
int main()
{
    char string1[MAX_LENGTH];
    char string2[MAX_LENGTH];

    strcpy(string1, "Hello World!");
    strcpy(string2, string1);

    return 0;
}
```

**String Operation: Assignment**

string1: "Hello World!"
string2: "Hello World!"
Common Mistake:
Wrong Assignment

```c
char name1[5] = "Ann";
char name2[5] = "Dave";

name2 = name1;
```

Error! It will not compile
String Operation: Concatenation

```c
char string1[MAXLENGTH];
char string2[MAXLENGTH];

strcpy(string1, "Goodbye");
strcpy(string2, ", Cruel ");

strcat(string1, string2);
strcat(string1, string2);
strcat(string1, "World!");

string1: “Goodbye”
string2: “, Cruel “
```
String Operation: Concatenation

char string1[MAXLENGTH];
char string2[MAXLENGTH];

strcpy(string1, "Goodbye");
strcpy(string2, ", Cruel ");

strcat(string1, string2);
strcat(string1, string2);
strcat(string1, "World!");

string1: “Goodbye, Cruel”
string2: “, Cruel “
String Operation: Concatenation

```c
char string1[MAXLENGTH];
char string2[MAXLENGTH];

strcpy(string1, "Goodbye");
strcpy(string2, ", Cruel ");

strcat(string1, string2);
strcat(string1, string2);
strcat(string1, "World!");
```

string1: “Goodbye, Cruel, Cruel”
string2: “, Cruel “
String Operation: Concatenation

```c
char string1[MAXLENGTH];
char string2[MAXLENGTH];

strcpy(string1, "Goodbye");
strcpy(string2, ", Cruel ");

strcat(string1, string2);
strcat(string1, string2);
strcat(string1, "World!");

string1: “Goodbye, Cruel , Cruel World!”
string2: “, Cruel “
```
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"
```c
strcpy(string1, "Apple");
strcpy(string2, "Wax");

if (strcmp(string1, string2) < 0) {
    cout << string1 << " " << string2;
} else {
    cout << string2 << " " << string1;
}
```

output: Apple Wax
String Operation: Comparison

```c
strcpy(string1, "Apple");
strcpy(string2, "Wax");

if (strcmp(string1, string2) < 0) {
    cout << string1 << " " << string2;
} else {
    cout << string2 << " " << string1;
}
```

**Returns:**
- **negative** if `string1 < string2`
- **zero** if `string1 == string2`
- **positive** if `string1 > string2`
Common Mistake:
Wrong Comparison

```c
strcpy(string1, "Apple");
strcpy(string2, "Wax");

if (string1 < string2)
{
    cout << string1 << " " << string2;
}
else
{
    cout << string2 << " " << string1;
}
```
char string1[100];
strcpy(string1, "Apple");
cout << strlen(string1);

output: 5

Number of char's before the '\0'
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:

```cpp
    cin >> name;
```

– Stores the next input C-string into name

• To read strings with blanks, use `getline` function:

```cpp
    cin.getline(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:

\texttt{cout \textless\textgreater \ name;}

– Outputs the content of name on the screen
– \texttt{\textless\textgreater} 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
  • \texttt{\textless\textgreater} continues to output data from memory adjacent to name until a '\0' is found
Library `cctype` contain useful functions to manipulate characters

- `int isalnum(int); // non-zero if arg is alpha or num`  
- `int iscntrl(int); // non-zero if arg is a ctrl character`  
- `int isdigit(int); // non-zero if arg is '0' ~ '9'`  
- `int islower(int); // non-zero if arg is lowercase letter`  
- `int isspace(int); // non-zero if arg is whitespace`  
- `int isupper(int); // non-zero if arg is uppercase letter`  
- `int toupper (int); // if arg is lower case returns upper`  
  // case, otherwise returns arg`
Character Functions

**Attention:**

- The character functions, although listed with int arguments and returning an int, are usually used with char as argument and returning type.

- Remember that whitespace is space, newline, line feed, formfeed, carriage return, tab or vertical tab.
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:

\[
\text{arrayName[indexExp1][indexExp2]}
\]

– Where \text{indexExp1} and \text{indexExp2} are expressions with positive integer values, and specify the row and column position

• Example:

\[
\text{sales[5][3] = 25.75;}
\]
### Accessing Array Components (cont’d.)

<table>
<thead>
<tr>
<th>sales</th>
<th>[0]</th>
<th>[1]</th>
<th>[2]</th>
<th>[3]</th>
<th>[4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[5]</td>
<td></td>
<td></td>
<td></td>
<td>25.75</td>
<td></td>
</tr>
<tr>
<td>[6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[8]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[9]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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
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
Initialization

• 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:

```cpp
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):

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

  – To input data into each component of matrix:

```cpp
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:

```c++
//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;
}
```
• Example:

– To find the largest element in each row:

```cpp
//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
Another Way to Declare a Two-Dimensional Array

- Can use `typedef` to define a two-dimensional array data type:

```cpp
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:

```cpp
tableType matrix;
```
Multidimensional Arrays

• *n*-dimensional array: collection of a fixed number of elements arranged in *n* dimensions (*n* \( \geq 1 \))

• Declaration syntax:

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

• To access a component:

```
arrayName[indexExp1][indexExp2] ... [indexExpn]
```
Let us now look at workshop 3 question