Programming Fundamentals I

Chapter 8: Arrays and Strings

Dr. Adriana Badulescu

Objectives

- Learn about arrays
- Learn about C-strings
- Learn about parallel arrays
- Learn about two-dimensional arrays
- Learn about multidimensional arrays

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 wherein 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

should be a constant

Accessing Array Components

- General syntax:

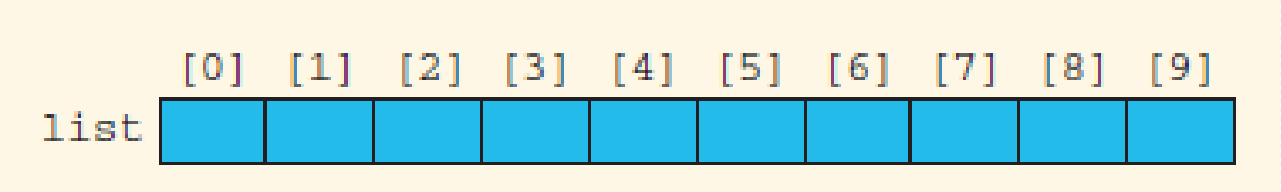
```
ArrayName[indexExp]
```

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

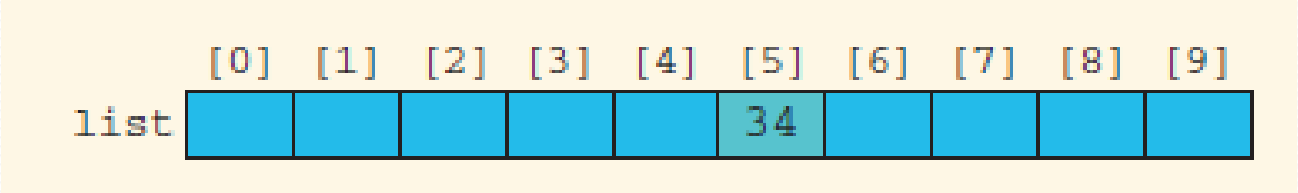
- Index value specifies the position of the component in the array
- **`[ ]`** is the **array subscripting operator**
- The array index always starts at `0` and ends with `intExp-1`

Accessing Array Components

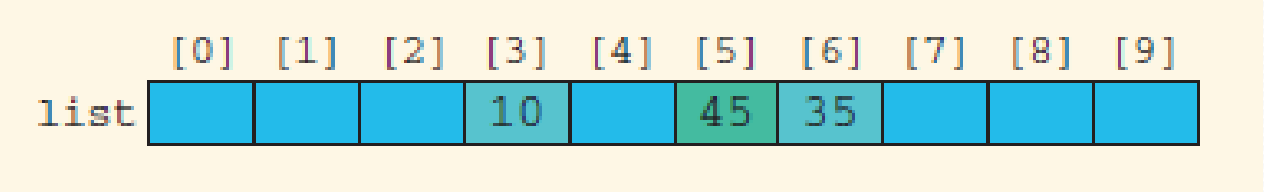
```
int list[10];
```




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




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



Accessing Array Components

- CORRECT

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

- INCORRECT

```
int arraySize;

cout << "Enter the size of the array: ";
cin >> arraySize;
cout << endl;

int list[arraySize];
```

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

Processing One-Dimensional Arrays

- 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++)
     //process list[i]
```

- Example:

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

Processing One-Dimensional Arrays

```
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] << " ";
```

Processing One-Dimensional Arrays

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 and the size is determined by the number of initial values in the curly brackets

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

- If the size is specified the values not specified will be initialized with 0
- 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`

Some Restrictions on Array Processing

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

Operation	C++	Legal or Illegal
Assignment	array1=array2	Illegal
Input	cin>>array1	Illegal
Output	cout << array1	Legal but incorrect
Comparison	(array1<array2)	Legal but incorrect

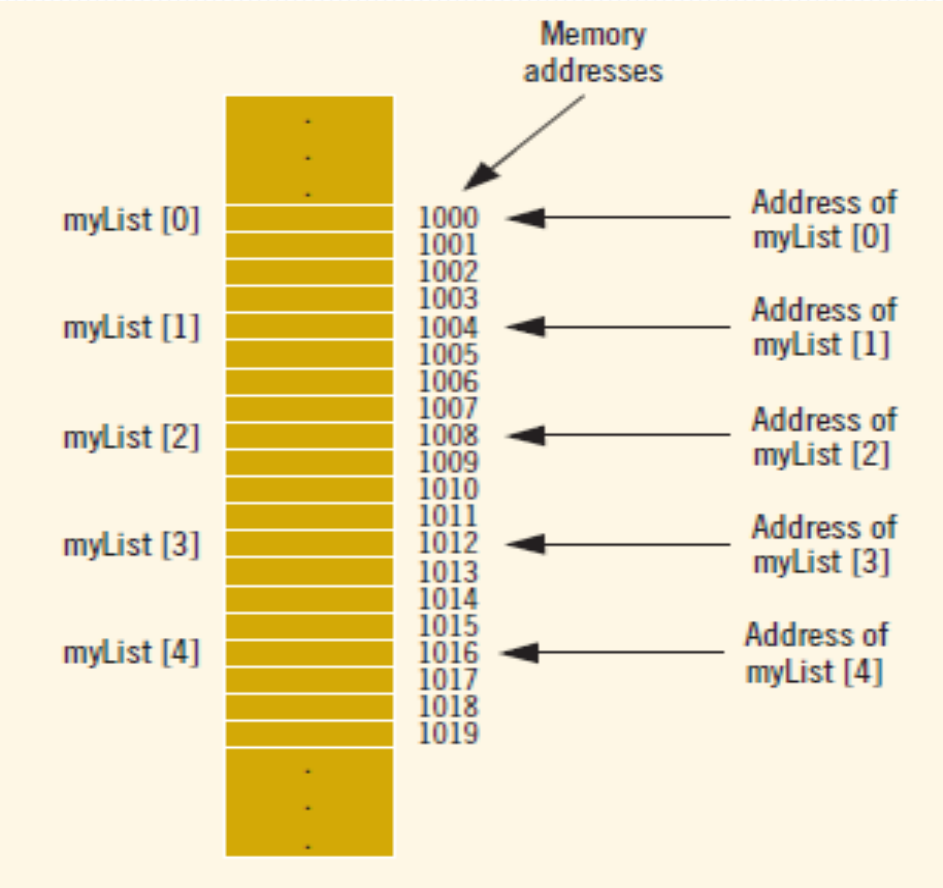
- How to do it? Component by component

Some Restrictions on Array Processing

- **Output**
- The following statements are legal, but do not give the desired results:

```
cout << yourList;

if (myList <= yourList)
.
.
```

- Print the base address of the array



Arrays as Parameters to Functions

- Arrays are passed by reference only
  - The symbol **&** is *not* used when passing an array as a formal parameter
  - The size of the array is usually omitted and It is usually send as another parameter
  - When we pass an array as a parameter, the base address of the actual array is passed to the formal parameter
- C++ does not allow functions to return a value of the type array

```
void initializeArray(int list[], int listSize)
{
    int index;

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

Arrays and Functions

```
//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 read and store the data into an int array.
//The array to store the data and its size are passed as
//parameters. The parameter listSize specifies the number
//of elements to be read.
void fillArray(int list[], int listSize)
{
    int index;

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

Arrays and Functions

```
//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] << " ";
}


//Function to find and return the sum of the
//elements of an int array. The parameter listSize
//specifies the number of elements to be added.
int sumArray(const int list[], int listSize)
{
    int index;
    int sum = 0;

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

    return sum;
}
```

Arrays and Functions

```
//Function to find and return the index of the first
//largest element in an int array. The parameter listSize
//specifies the number of elements in the array.
int indexLargestElement(const int list[], int listSize)
{
    int index;
    int maxIndex = 0; //Assume the first element is the largest

    for (index = 1; index < listSize; index++)
        if (list[maxIndex] < list[index])
            maxIndex = index;

    return maxIndex;
}
```

Arrays and enum

- C++ allows any integral type to be used as an array index
- First element has to be 0
- 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;
```

Arrays and typedef

```
const int SIZE = 50;

double yourList[SIZE];
double myList[SIZE];
```

OR

```
typedef double list[SIZE];

list yourList;
list myList;
```

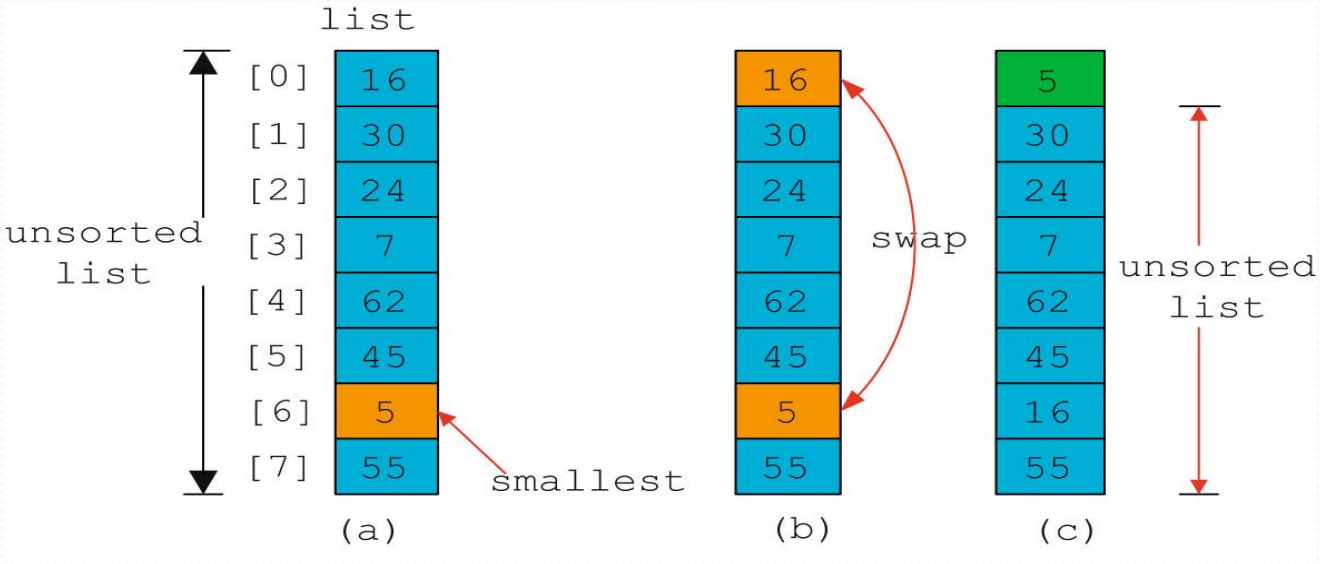
Searching an Array for a Specific Item

- **Sequential search** or **linear search**
  - Searching a list for a given item
  - Starting from the first array element
  - Compare `searchItem` with the elements in the array
  - Continue the search until either you find the item or no more data is left in the `list` to compare with `searchItem`

```
int seqSearch(const int list[], int listLength, int searchItem)
{
    int loc;

    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;

    return -1;

}
```

Sorting

- **Selection sort** rearrange the list by selecting an element and moving it to its proper position in the array
  - Find the smallest element in the unsorted portion of the list
  - Move it to the top of the unsorted portion by swapping with the element currently there
  - Start again with the rest of the list



Sorting

```
void selectionSort(int list[], int length)
{
    int index;
    int smallestIndex;
    int location;
    int temp;

    for (index = 0; index < length - 1; index++)
    {
        smallestIndex = index;
        for (location = index + 1; location < length; location++)
            if (list[location] < list[smallestIndex])
                smallestIndex = location;
        temp = list[smallestIndex];
        list[smallestIndex] = list[index];
        list[index] = temp;
    }
}
```

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
```

- Range-based `for` loop

```
sum = 0;
for (double num : list)
    sum = sum + num;
```

Example: Arrays

- A – array of 5 integers, initialize with 0

```
//Array A of 5 integers
//constant for the size of the array
const int SIZE = 5;
//declare the array
int A[SIZE];
//initialize the array
int k;
for (k = 0; k < SIZE; k++)
        A[k] = 0;
//print the array
cout << "\n\nThe array A is: ";
//cout << A;
for (k = 0; k < SIZE; k++)
        cout << A[k]<<" ";
```

```
The array A is: 0 0 0 0 0
```

Example: Arrays

```
//Array A2 with values 1 2 3 4 5, initialize at declaration
int A2[SIZE] = { 1,2,3,4,5 };
cout << "\n\nThe array A2 is: ";
for (k = 0; k < SIZE; k++)
        cout << A2[k] << " ";
```

```
The array A2 is: 1 2 3 4 5
```

```
//Array A3 with values 0 1 2 3 4, assign the values to components
int A3[SIZE];
A3[0] = 0;
A3[1] = 1;
A3[2] = 2;
A3[3] = 3;
A3[4] = 4;
//A3[5] = 5;
cout << "\n\nThe array A3 is: ";
for (k = 0; k < SIZE; k++)
        cout << A3[k] << " ";
```

```
The array A3 is: 0 1 2 3 4
```

Example: Arrays

```
//Aray A4 with values 0 1 2 3 4, with a formula/loop
int A4[SIZE];
for (k = 0; k < SIZE; k++)
        A4[k] = k;
cout << "\n\nThe array A4 is: ";
for (k = 0; k < SIZE; k++)
        cout << A4[k] << " ";
```

```
The array A4 is: 0 1 2 3 4
```

- Easiest to declare and initialize
- Scales well

Example: Arrays

- Search for 9 in A4

```
int search = 9;
bool found = false;
for (k = 0; k < SIZE; k++)
        if (A4[k] == search)
        {
                found = true;
                break;
        }
if (found)
        cout << "\n\nWe found " << search << " in A4 at position " << k;
else
        cout << "\n\nWe did not find " << search << " in A4";
```

```
We did not find 9 in A4
```

Example: Arrays

- Read values from the user

```
//Array A5 with values from the user
cout << "\n\nEntering the values for the array A5\n";

int A5[SIZE];
for (k = 0; k < SIZE; k++)
{
        cout << "\nEnter a value for position " << k << ": ";
        cin >> A5[k];
}
cout << "\n\nThe array A5 is: ";
for (k = 0; k < SIZE; k++)
        cout << A5[k] << " ";
```

```
Enter a value for position 0: 5

Enter a value for position 1: 3

Enter a value for position 2: 8

Enter a value for position 3: 2

Enter a value for position 4: 1


The array A5 is: 5 3 8 2 1
```

Example: Arrays

- Sort the array A5 using selection sort

```
for (k = 0; k < SIZE - 1; k++)
{
        //find the smallest after k
        int smallest = k;
        int n;
        for (n = k + 1; n < SIZE; n++)
                if (A5[n] < A5[smallest])
                        smallest = n;
        //swap the smallest with A5[k]
        int temp;
        temp = A5[k];
        A5[k]= A5[smallest];
        A5[smallest]=temp;
}
cout << "\n\nThe array A5 is: ";
for (k = 0; k < SIZE; k++)
        cout << A5[k] << " ";
```

```
The array A5 is: 5 3 8 2 1

The array A5 is: 1 2 3 5 8
```

C–Strings (Character Arrays)

- **Character array**: an array whose components are of type `char`
- **C-strings** are null-terminated (`'\0'` - **null character**) 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)

- Initialize at declaration

```
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. Equivalent to

```
char name[] = {'J', 'o', 'h', 'n', '\0'};
```

C–Strings (Character Arrays)

- **Functions**

Function	Effect	Example
strcpy(s1,s2)	Copies the c-string s2 into s1	strcpy(s,"John Doe");
strcmp(s1,s2)	Return a negative number is s1 is less that s2 0 if S1 and S2 are the same A positive number, if S1 is larger than S2	strcmp("an", "air") -> 1 strcmp("Bill","Billy") -> -1
strlen(s)	Returns the number of characters from s encluding the null character	strlen("John Doe") -> 8

Reading and Writing C−Strings

- Most rules that apply to arrays apply to C-strings as well
- Aggregate operations, such as assignment and comparison, are not allowed on arrays
- Even the input/output of arrays is done component-wise
- The one place where C++ allows aggregate operations on arrays is the input and output of C-strings (that is, character arrays)

C–Strings Input and Output

Statement	Effect
cin >> name;	Inputs the next input C-string into name (characters until the next whitespace)
cin.get(str, m+1);	Inputs the next m characters into str (including space and tabs) but the newline character is not stored in str If the input string has fewer than m characters, the reading stops at the newline character
cout << name;	Outputs the content of name on the screen until it finds the null character

Example: C-strings

```
//C1 Mary , intialize at declaration
char C1[SIZE] = "Mary";
//print the c-string
cout << "\n\nThe C-String C1 is: " << C1;

//C2 Jon , assign to components
char C2[SIZE];
C2[0] = 'J';
C2[1] = 'o';
C2[2] = 'n';
C2[3] = '\0';
cout << "\n\nThe C-String C2 is: " << C2;

//C3 David , strcpy
char C3[SIZE+1];
strcpy_s(C3, "David");
cout << "\n\nThe C-String C3 is: " << C3;
```

```
The C-String C1 is: Mary

The C-String C2 is: Jon

The C-String C3 is: David
```

C-Strings vs String

	C-Strings	Strings
Format	C A T \0	C A T
Declare	char S[10]	string S
Initialize at declaration	char S[10]="CAT" char S[10]={'C', 'A', 'T', '\0'}	String S="CAT"
Assignment	strcpy(S, "DOG")	S="DOG"
Comparison	strcmp(S, "CAT") -> 0 strcmp(S, "DOG") -> -1	S=="CAT" -> true S=="DOG" -> false
Length	strlen(S)	S.length() S.size()
Input	cin >> S getline(cin,S) cin.get(S,11)	cin>>S getline(cin,S)
Output	cout<<S	cout<<S

C-Strings vs String

	C-Strings	Strings
Concatenation	Component-by-component	S+"DUCK"
Search	Component-by-component	S.find("DUCK") S.find("DOG",10)
Append	Component-by-component	S.append(S, "&DOG") S+="&DOG"
Insert	Component-by-component	S.insert(0, "DOG")
Erase	Component-by-component	S.erase(1,1)
Replace	Component-by-component	S.replace(0,3, "DUCK")
Swap	Component-by-component	S.swap("CAT")

Specifying Input/Output Files at Execution Time

- You can let the user specify the name of the input and/or output file at execution time:

```
ifstream infile;
ofstream outfile;

char fileName[51];     //assume that the file name is at most
                       //50 characters long
cout << "Enter the input file name: ";
cin >> fileName;

infile.open(fileName);   //open the input file
.
.
.

cout << "Enter the output file name: ";
cin >> fileName;

outfile.open(fileName);  //open the output file
```

- Argument to the function `open` must be a null-terminated string (a `C`-string)

`string` Type and Input/Output Files

- If we use a variable of type `string` to read the name of an I/O file, the value must first be converted to a C-string before calling `open`
- Using `strVar.c_str()` where `strVar` is a variable of type `string`

`ifstream infile;`
`char fileName[50];`
`cin >> fileName;`
`infile.open(fileName);`
	`ifstream infile;`
	`string fileName;`
	`cin >> fileName;`
	`infile.open(fileName.c_str());`

Example: Strings and Searches

- Create a string with all the digits between 0 and 9, without hardcoding it

```
char c;
string S;
for (c = '0'; c <= '9'; c++)
        S += c;
cout << "\nThe string S is " << S;
```

- Append 10 more of 0123456789 to the string s

```
int k;
for (k = 1; k <= 10; k++)
        S.append("0123456789");
cout << "\nThe string S is " << S;
```

```
The string S is 0123456789
The string S is 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
```

Example: Strings and Searches

- Search for “123" in S

```
//search for 123 in S
Search = "123";
Pos = S.find(Search);
//check if we found it
if (Pos != string::npos)
        cout << "\n\n" << Search << " is at position " << Pos << " in " << S;
else
        cout << "\n\n" << Search << " is not in " << S;
```

```
123 is at position 1 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
```

- search for the second “123" in S

```
Search = "123";
Pos = S.find(Search, Pos+1);
//check if we found it
if (Pos != string::npos)
        cout << "\n\n" << Search << " is at position " << Pos << " in " << S;
else
```

```
123 is at position 1 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
123 is at position 11 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
```

Example: Strings and Searches

- Find how many times/all occurences of "123" in S

```
int Number = 0;
Search = "123";
Pos = -1;
do
{
    Pos = S.find(Search, Pos + 1);
    //check if we found it
    if (Pos != string::npos)
    {
        cout << "\n\n" << Search << " is at position " << Pos << " in " << S;
        Number++;
    }
} while (Pos != string::npos);

if (Number==0)
    cout << "\n\n" << Search << " is not in " << S;
else
    cout << "\n\n" << Search << " is in " << S << "\n" << Number << " times";
```

Example: Strings and Searches

```
123 is at position 1 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
123 is at position 11 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
123 is at position 21 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
123 is at position 31 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
123 is at position 41 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
123 is at position 51 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
123 is at position 61 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
123 is at position 71 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
123 is at position 81 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
123 is at position 91 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
123 is at position 101 in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
123 is in 0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678911 times
```

Example: Strings and Searches

```
string A;
char c;
for (c = 'A'; c <= 'Z'; c++)
        A+=c;
cout << "\nThe string A is " << A;
```

```
The string A is ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
//find first consonant in A
Pos = A.find_first_not_of("AEIOU");
cout << "\n\nThe first consonant ("<<A[Pos]<<") is at position " << Pos;
//find last vowel in A
Pos = A.find_last_of("AEIOU");
cout << "\n\nThe last vowel (" << A[Pos] << ") is at position " << Pos;
```

```
The first consonant (B) is at position 1

The last vowel (U) is at position 20
```

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];
```

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- and Multidimensional Arrays

int matrix[5][3];

matrix	[0]	[1]	[2]
[0]			
[1]			
[2]			
[3]			
[4]			

Accessing Array Components

- Syntax: `ArrayName[indexExp1][indexExp2];`

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

matrix	[0]	[1]	[2]
[0]	0	1	2
[1]	10	11	12
[2]	20	21	22
[3]	30	31	32
[4]	40	41	42

matrix[2][1]

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, if all components of a row aren't specified, unspecified ones are set to 0

```
int matrix[5][3]={
        { 0 , 1 , 2 },
        { 10 , 11 , 12 },
        { 20 , 21 , 22 },
        { 30 , 31 , 32 },
        { 40 , 41 , 42 }
}
```

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 = 5;
- const int NUMBER_OF_COLUMNS = 3;
- int matrix[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];

matrix	[0]	[1]	[2]
[0]	0	1	2
[1]	10	11	12
[2]	20	21	22
[3]	30	31	32
[4]	40	41	42

Processing

```
//BEFORE TABLE
for (row=0; row<NUMBER_OF_ROWS; row++)
{
        //BEFORE ROW
        for(col=0; col<NUMBER_OF_COLS; col++)
        {
                //BEFORE CELL/COLUMN
                //process matrix[row][col]
                //AFTER CELL/COLUMN
        }
        //AFTER ROW
}
//AFTER TABLE
```

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;
```

Input and Output

- 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];
```

- 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;
}
```

Sum by Row and Column

- To find the 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;
}
```

- To find the 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

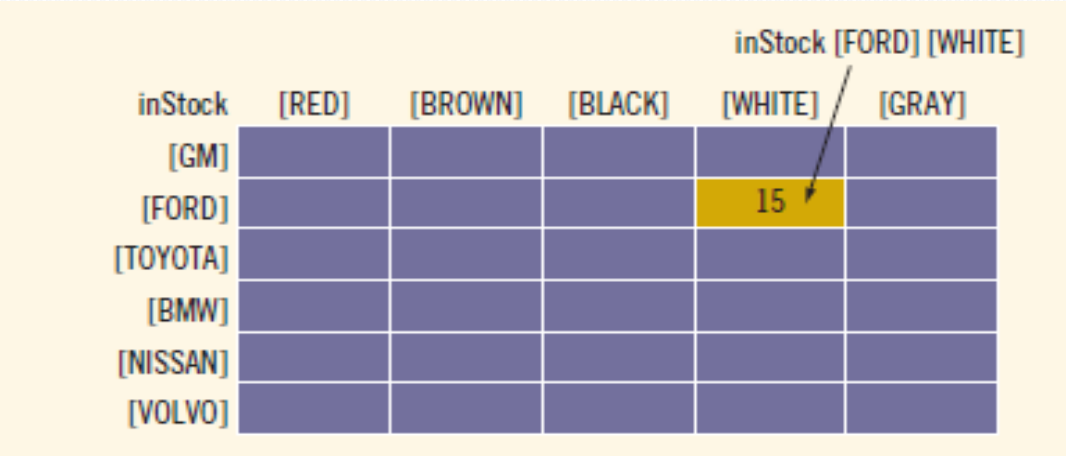
```
  //Largest element in each row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
    largest = matrix[row][0]; //Assume that the first element
                              //of the row is the largest.
    for (col = 1; col < NUMBER_OF_COLUMNS; col++)
        if (largest < matrix[row][col])
        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

Passing Two-Dimensional Arrays as Parameters to Functions

- Two-dimensional arrays can be passed as parameters to a function
  - Pass by reference
    - Base address (address of first component of the actual parameter) 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

Two-Dimensional Arrays and Enumeration Types

```
const int NUMBER_OF_ROWS = 7;
const int NUMBER_OF_COLUMNS = 6;
enum CarType {GM, FORD, TOYOTA, BMW, NISSAN, VOLVO};
enum ColorType { RED, BROWN, BLACK, WHITE, GRAY};
```




```
typedef int STOCK[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
STOCK inStock1;
STOCK inStock2;
cout << inStock1[TOYOTA][BLACK];
cout << inStock2[BMW][RED];
```

Two-Dimensional Arrays and typedef

- 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;
```

Example: Two-Dimensional Array

```
const int NROWS = 5;
const int NCOLS = 3;

int M[NROWS][NCOLS]=
{
        {  0, 1, 2 },
        { 10,11,12 },
        { 20,21,22 },
        { 30,31,32 },
        { 40,41,42 }
};

//print the matrix
int row, col;
cout << "\n\nThe matrix is: \n";
for (row = 0; row < NROWS; row++)
{
        for (col = 0; col < NCOLS; col++)
                cout << setw(4) << M[row][col];
        cout << "\n";
}
```

```
The matrix is:
   0   1   2
  10  11  12
  20  21  22
  30  31  32
  40  41  42
```

Example: Two-Dimensional Array

```
//initialize with formula
for (row = 0; row < NROWS; row++)
        for (col = 0; col < NCOLS; col++)
                M[row][col] = row * 10 + col;

cout << "\n\nThe matrix is: \n";
for (row = 0; row < NROWS; row++)
{
        for (col = 0; col < NCOLS; col++)
                cout << setw(4) << M[row][col];
        cout << "\n";
}
```

```
The matrix is:
   0   1   2
  10  11  12
  20  21  22
  30  31  32
  40  41  42
```

Example: Two-Dimensional Array

```
//search for 31 in M
search = 31;
found = false;
   for (row = 0; (row < NROWS) && (found==false) ; row++)
      for (col = 0; col < NCOLS; col++)
         if (M[row][col] == search)
         {
            found = true;
            cout << "\n\n" <<search<< " is in the matrix on row " <<row<< " and column " <<col;
            break;
         }
if (found==false)
      cout << "\n\n" << search << " is not in the matrix!";
```

```
The matrix is:
   0   1   2
  10  11  12
  20  21  22
  30  31  32
  40  41  42


31 is in the matrix on row 3 and column 1
```

Example: Two-Dimensional Array

```
//sum of all values in the matrix
int Sum = 0;
for (row = 0; row < NROWS; row++)
       for (col = 0; col < NCOLS; col++)
              Sum = Sum + M[row][col];
cout << "\n\nThe sum of all the values from the matrix is: " << Sum;

//sum of all values from each row

for (row = 0; row < NROWS; row++)
{
       Sum = 0;
       for (col = 0; col < NCOLS; col++)
              Sum = Sum + M[row][col];
       cout << "\n\nThe sum of all the values from the matrix from row "<<row<<" is: " << Sum;
}

//sum of all values from each column
for (col = 0; col < NCOLS; col++)
{
       Sum = 0;
       for (row = 0; row < NROWS; row++)
              Sum = Sum + M[row][col];
       cout << "\n\nThe sum of all the values from the matrix from column "<<col<<" is: " << Sum;
}
```

Example: Two-Dimensional Array

```
The sum of all the values from the matrix is: 315

The sum of all the values from the matrix from row 0 is: 3

The sum of all the values from the matrix from row 1 is: 33

The sum of all the values from the matrix from row 2 is: 63

The sum of all the values from the matrix from row 3 is: 93

The sum of all the values from the matrix from row 4 is: 123

The sum of all the values from the matrix from column 0 is: 100

The sum of all the values from the matrix from column 1 is: 105

The sum of all the values from the matrix from column 2 is: 110
```

Arrays of Strings

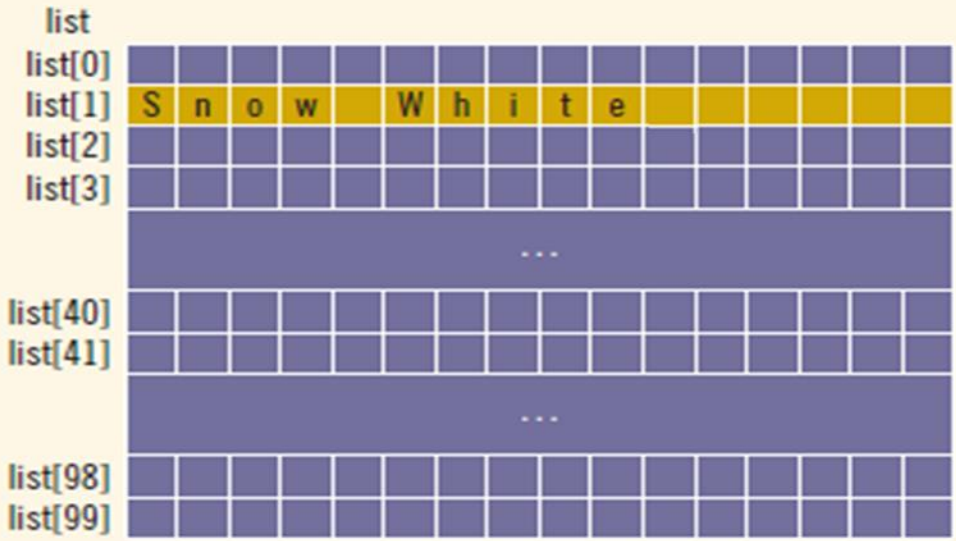
- Strings in C++ can be manipulated using either the data type string or character arrays (`C`-strings)
- On some compilers, the data type `string` may not be available in Standard C++ (i.e., non-ANSI/ISO Standard C++)
- To declare an array of 100 components of type `string`:

```
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

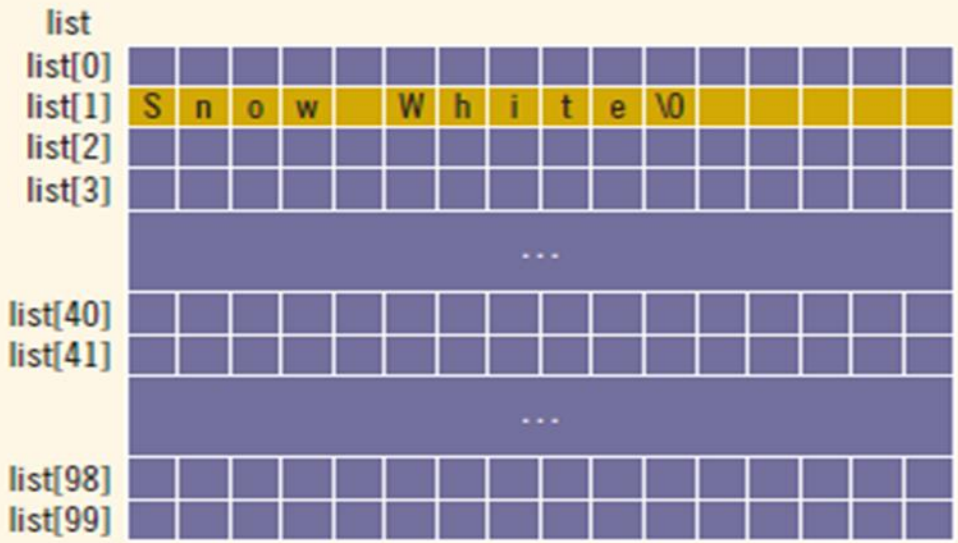
Arrays of Strings and `string` Type

```
string list[100];
list[1]="Snow White";
```



Arrays of Strings and C-Strings (Character Arrays)

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



Example: Array of C-Strings

```
//  o
// /|\
// / \

char M[3][4] =
{
        {' ','o',' ','\0'},
        { '/','|','\\' ,'\0' },
        { '/',' ','\\' ,'\0' }
};
//print it out as 2D array
int row, col;
cout << "\n\nThe 2D array is:\n";
for (row = 0; row < 3; row++)
{
        for (col = 0; col < 3; col++)
                cout << M[row][col] ;
        cout << "\n";
}

//print it out as array of C-strings
cout << "\n\nThe C-strings array is:\n";
for (row = 0; row < 3; row++)
        cout << M[row] << "\n";
```

```
The 2D array is:
 o
/|\
/ \


The C-strings array is:
 o
/|\
/ \
```

Example: Array of Strings

```
//S - array of strings
string S[3] =
{
        " o ",
        "/|\\",
        "/ \\"
};
cout << "\n\nThe strings array is:\n";
for (row = 0; row < 3; row++)
        cout << S[row] << "\n";
```

```
The strings array is:
 o
/|\
/ \
```

Parallel Arrays

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

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

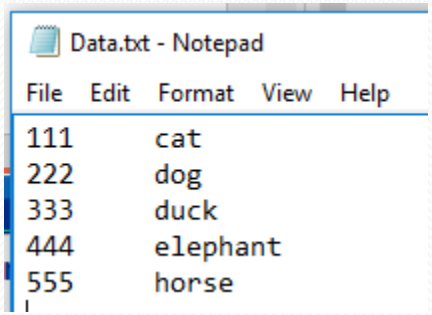
```
23456 A
86723 B
22356 C
92733 B
11892 D
.
.
.
```

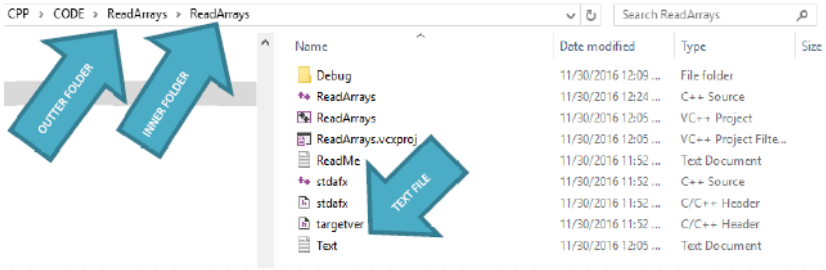
Example: Read Concepts from File

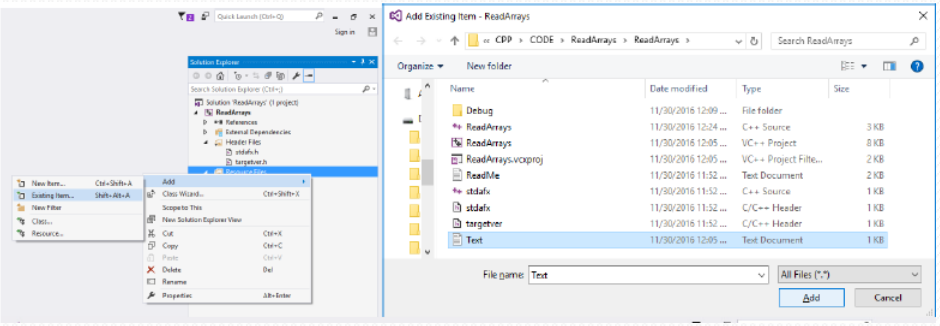
- Write a program that contains the read data from a formatted file (that has 2 columns, one with an integral number and one with a string), reads the integral numbers into an array of int and the strings into an array of strings (parallel arrays) and print them and then sort them based on the integral values and print the sorted arrays again.

Example: Read Concepts from File

- Save the file on disk in the project inner folder (same folder where the CPP file is in)
- Add the file to the project as a resource file
- In your code, open the file (make sure you spell the name of the file correctly including caps), check if the file opened correctly, read from the file, close the file







Example: Read Concepts from File into Array

```
ifstream in("Data.txt");
if (!in)
        cout << "\n\nCannot open the Data.txt file!";
else
{
        const int SIZE = 100;
        int NUMBERS[SIZE];
        string NAMES[SIZE];
        int Position = 0;
        cout << "\n\nReading from the Data.txt file...";
        do
        {
                //read number
                int Number;
                in >> Number;
                if (in.eof())
                        break;
                //read name
                string Name;
                in >> Name;
```

Example: Read Concepts from File into Array

```
            //add to arrays
            NUMBERS[Position] = Number;
            NAMES[Position] = Name;
            Position++;
        } while (in);
        in.close();

        //print arrays
        cout << "\n\nThe parallel arrays are: ";
        for (k = 0; k < Position; k++)
            cout << "\n" << setw(10) << left<< NAMES[k] << setw(5) << NUMBERS[k];
        //cout << "\nDONE";

}
```

```
The parallel arrays are:
cat       111
dog       222
duck      333
elephant  444
horse     555
```

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

Summary

- Enumeration types
- Anonymous types
- Typedef statements
- Namespaces
- Strings
- Arrays
- C-Strings