CS135; 1D Arrays and Strings

Gael Zarco

April 21, 2025

Structured Data Types contain data where each item is a collection of other data items.

- Simple data structures are the building blocks of structured data types.

1 Arrays

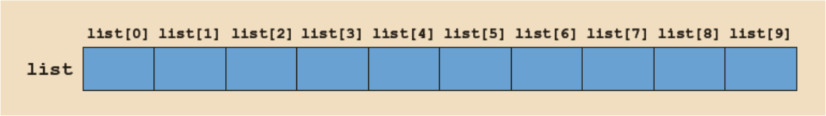
An **Array** is a collection of a fixed number of components (also called elements) all of the same data type and in contigous (adjacent) memory space. A **One-Dimensional Array** is an array in which the components are arranged in a list form.

Listing 1: 1D Array Syntax

```
dataType arrayName[intExp];

// Example
int list[5];     // Declared array ’list’ of 10 elements
```

`intExp` specifies the number of components in the array and can be any constant expression that evaluates to a positive integer. The Example above delcares an array `list` of *10* components.

- The components are `list[0], list[1], ..., list[9]`.
- Declares a total of 10 variables

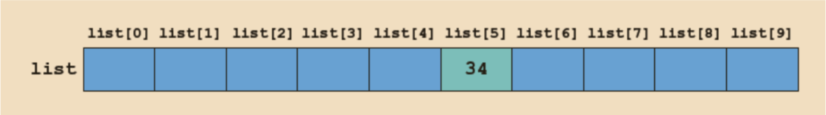


Listing 2: 1D Array Assignment

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

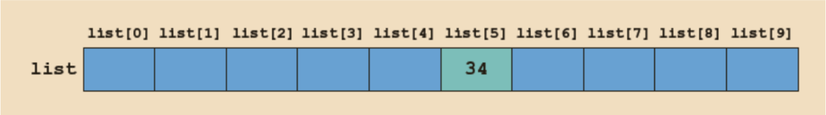
This expression stores *34* in `list[5]`, which is the *sixth* component of the array `list`.

- You can use i to index into an array.
  - list[i]



Listing 3: 1D Array Assignment Cont

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

// Add the contents of list[3] and list[6] and store in list[5]
list[5] = list[3] + list[6];
```



1.1 Processing One-Dimensional Arrays

Listing 4: Read Data Into 1D Array

```
for (i = 0; i < 10; i++)
  cin >> list[i];
```

Listing 5: Print Data From 1D Array

```
for (i = 0; i < 10; i++)
  cout << list[i];
```

Listing 6: Find Sum & Avg From 1D Array

```
int sum = 0;
int avg;

for (i = 0; i < 10; i++)
  sum = sum + list[i];

avg = sum / 10;
```

Listing 7: Find Largest Element in 1D Array

```
int maxIdx = 0;

for (int i = 1; i < 10; i++)
```

```
  if (list[maxIdx] < list[i])
    maxIdx = i;

int largestInt = list[maxIdx];
```

1.2 Array Index Out of Bounds

Listing 8: Array Index Example

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

The component `num[i]` is *valid* or **In Bounds** if `index`:

- $0 \leq \text{index} \leq \text{ARRAY\_SIZE} - 1$.
- `index` is not negative or greater than $\text{ARRAY\_SIZE} - 1$.
  - It is **Out of Bounds** in this event.
  - C++ does not check whether the index value is within range; this is the programmer's responsibility.

1.3 Array Initialization During Declaration

An array can be initialized while being declared

Listing 9: Array Initialization Example

```
double sales[5] = {12.25, 32.50, 16.90, 23, 45.68};

// Not necessary to specify the size when initializing
double sales[] = {12.25, 32.50, 16.90, 23, 45.68};
```

1.4 Partial Initialization of Arrays During Declaration

Listing 10: Partial Array Initialization

```
int list[10] = {5, 6, 3};
```

The first three components of `list` are `list[0] = 5`, `list[1] = 6`, `list[2] = 3`, and the rest are set to the default of 0.

1.5 Restrictions on Array Processing

C++ does not allow **Aggregate Operations** on an array. Aggregate operations on an array are any operations that manipulate the entire array as a single unit.

Listing 11: Illegal Aggregate Operation on Array

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

// illegal
yourList = myList;
```

1.6 Arrays as Parameters to Functions

In C++, arrays are passed as parameters to functions by **Reference Only**. You do **not** use the `&` symbol when declaring an array as a formal parameter.

Listing 12: Arrays as Formal Parameters

```
void initialize(int list[], int listSize);
```

1.7 Constant Arrays as Formal Parameters

You can use `const` keyword in the declaration of a formal param to prevent the function from changing the actual param.

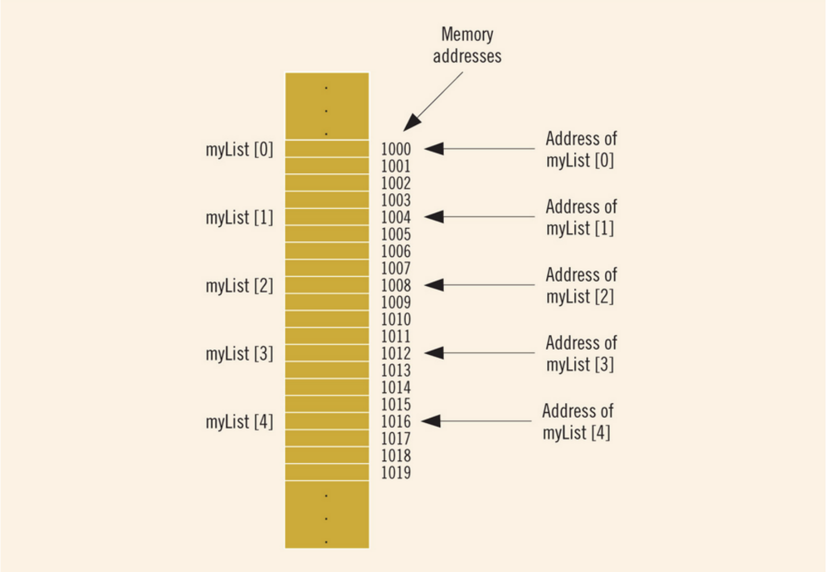
Listing 13: Constant Arrays as Formal Parameters

```
void foo(int x[], const int y[], int sizeX, int sizeY);
```

1.8 Base Address of an Array and Array in Computer Memory

The **Base Address** of an array is the address (memory location) of the first array component.

- In 1D arrays, the base address is list[0];



1.9 Functions Cannot Return a Value of the Type Array

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

1.10 Integral Data Type and Array Indices

C++ allows any integral type to be used as an array index.

Listing 14: Improved Code Readability

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

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

1.11 Other Ways to Declare Arrays

Listing 15: Declaration Using Existing Variable

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

Listing 16: Declaration with `using`

```
const int SIZE = 50;         // Line 1
using list = double[SIZE];   // Line 2
list yourList;               // Line 3
list myList;                 // Line 4
```

2 Searching an Array for a Specific Item

Sequential Search (*linear search*):

1. Searching a list for a given item, starting from the first element.
2. Compare each element in the array with the value being searched.
3. Continue to search until item is found or no more data is left.

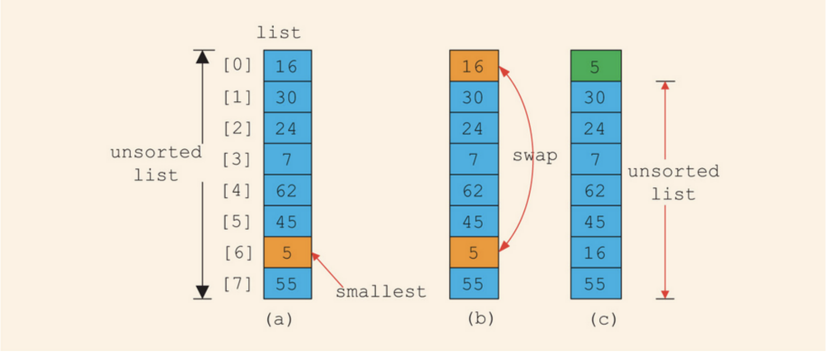
Listing 17: Simple Array Search

```
int seqSearch(const int list[], int listLength, int searchItem) {
  int  loc   = 0;
  bool found = false;

  while (loc < listLength && !found) {
    if (list[loc] == searchItem) {
      found = true;
    } else {
      ++loc;
    }
  }

  return found ? loc : -1;
}
```

3 Sorting

Selection Sort is rearranging the list by selecting an element and moving it to its proper position.

1. Find the smallest element in the unsorted portion of the list.
2. Move it to the top of the unsorted portion by swapping with current element.
3. Start again with the rest of the list.



4 Auto Declaration and Range-Based `for` Loops

Modern C++ allows for **Auto** declaration of variables

- Data type does not need to be specified.

```
auto num = 15;
```

The compiler deduces `num` to be of type `int`.

Listing 18: Range-Based `for` Loop

```
double list[25];
double sum = 0;

for (double num : list) {  // read as "for each num in list"
    sum += num;
}
```

5 C-Strings (Character Arrays)

A **Character Array** is an array whose components are of type `char`.

- C-strings are *null-terminated* ('
  `0'`) character arrays.
- Examples
  - 'A' is the character A
  - "A" is the C-string A
  - *Note*: "A" represents two characters. 'A' and '
    `0'`.

Listing 19: C-String Declaration

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

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.

Listing 20: Omitting Size of Array During Initialization

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

Declares an array of length *5* and stores the C-string "John" in the array. Useful `string` functions:

- `strcopy`

- `strncpy`
- `strcmp`
- `strlen`

5.1 `string` Comparison

C-Strings are compared character by character using the collating system sequence. Use the `strcmp` function to compare strings.

If using ASCII char set:

- ”Air” < ”Boat”
- ”Air” < ”An”
- ”Bill” < ”Billy”
- ”Hello” < ”hello”

5.2 Reading and Writing Strings

Most array rules apply to C-strings (which are `char` arrays). However, C++ **DOES** allow aggregate ops for the input and output of C-strings.
n

5.3 `string` Input

Listing 21: String Input Example

```
cin >> name;
```

Stores the next input C-string into `name`.

Listing 22: Read Strings with Blanks with `get`

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

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

5.4 string Output

Listing 23: String Output Example

```
cout << name;
```

- << continues to write the contents of name until it finds a null character.
- If name does not contain a null character, then strange output may occur as it will continue to output data from memory until a null character is found.

5.5 string Type and Input/Output Files

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

- If using a string var for the name of an I/O file, the value must first be converted to a C-string before calling open.
- Use the c_str function to convert.

Listing 24: c_str Syntax

```
strVar.c\_str();
```

Where strVar is a variable of type string.

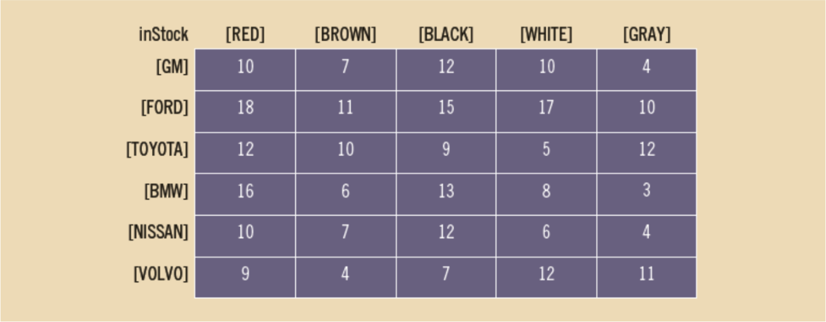
6 Parallel Arrays

Two (or more) arrays are called **Parallel** if their corresponding comonents hold related information.

Listing 25: Parallel Array Example

```
// Stores the values from a 2 column data set file into 2 separate
     arrays.
// Each index of each array refers to the same student (therefore
    parallel).
int noOfStudents = 0;

infile >> studentId[noOfStudents] >> courseGrade[noOfStudents];

while (inFile && noOfStudents < 50) {
  noOfStudents++;
  inFile >> studentId[noOfStudents]
  >> courseGrade[noOfStudents];
}
```

7 Two- and Multidimensional Arrays



inStock	[RED]	[BROWN]	[BLACK]	[WHITE]	[GRAY]
[GM]	10	7	12	10	4
[FORD]	18	11	15	17	10
[TOYOTA]	12	10	9	5	12
[BMW]	16	6	13	8	3
[NISSAN]	10	7	12	6	4
[VOLVO]	9	4	7	12	11

Storing the data set above would require a 1D array of 30 components of type `int`. The first five of which would store the first row of the table, then the next 5, and so on.

- This allows you to simulate the table within a 1D array.
- Not efficient and can be messy and hard to manage.

Two-dimensional Arrays are a collection of a fixed number of components arranged in rows and columns, where all components are of the same type.

Listing 26: Two-Dimensional Array Syntax

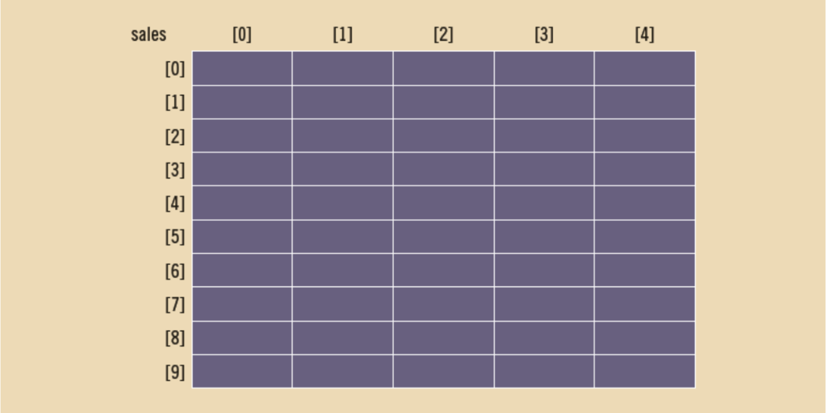
```
dataType arrayName[intExp1] [intExp2];
```

Where, `intExp1` and `intExp2` are constant expressions yielding positive integer values.

Listing 27: Two-Dimensional Array Example

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

- The *rows* are numbered 0-9
- The *columns* are numbered 0-4



7.1 Accessing Array Components

Listing 28: Two-Dimensional Array Indexing Syntax

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

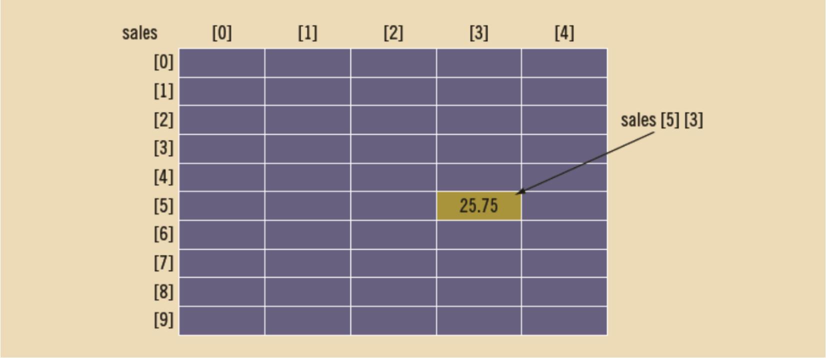
Where, `intExp1` and `intExp2` are constant expressions yielding positive integer values.

- `intExp1` specifies the row position.
- `intExp2` specifies the column position.

Listing 29: Two-Dimensional Array Indexing Example

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

Stores `25.75` into row number 5 and column number 3 of the array /textttsales.



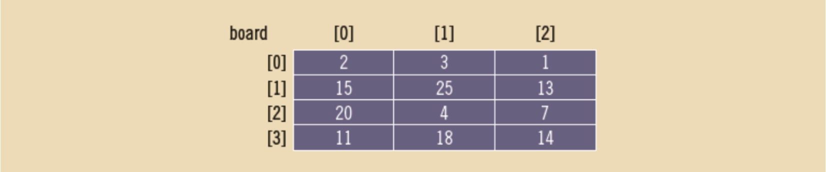
Listing 30: Two-Dimensional Array Indexing With Vars Example

```
sales[i][j] = 25.75;
```

7.2 Two-Dimensional Array Initialization During Declaration

Listing 31: Two-Dimensional Array Initialization Example

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



board	[0]	[1]	[2]
[0]	2	3	1
[1]	15	25	13
[2]	20	4	7
[3]	11	18	14

To initialize a two-dimensional array when it is declared:

1. The elements of each row are all enclosed within one set of curly braces, separated by commas.

2. Set of all rows is enclosed in curly braces.

3. For num arrays, if all components od a row are not specified, the unspecified components are initialized to `0`. At least one value is needed to initialize all the components of a row.

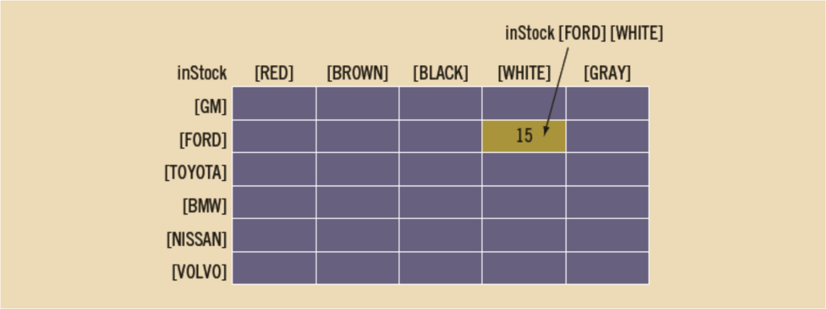
7.3 Two-Dimensional Arrays and Enumeration Types

You can also use enumeration type for array indices.

Listing 32: Enumeration Type in Two-Dimensional Arrays Example

```
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];

// Inserting a value into the 2D array
inStock[1][3] = 15;
```

```
// This is equivalent to the above
inStock[FORD][WHITE] = 15;
```



inStock	[RED]	[BROWN]	[BLACK]	[WHITE]	[GRAY]
[GM]					
[FORD]				15	
[TOYOTA]					
[BMW]					
[NISSAN]					
[VOLVO]					

7.4 Processing Two-Dimensional Arrays

A two-dimensional array can be processed in four ways:

1. Process a single element.
2. Process the entire array.
3. Process a particular row of the array, called **Row Processing**
4. Process a particular column of the array, called **Column Processing**

Each row and column of a two-dimensional array is a one-dimensional array.

- To process, use algorithms similar to processing one-dimensional arrays.

7.5 Initialization

Listing 33: Initialize Row Number 4

```
row = 4;

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

Listing 34: Initialize Entire Matrix

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

7.6 Print

Listing 35: Output 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 << '\n';
  }
}
```

7.7 Input

Listing 36: Adding Input to 4th Row

```
row = 4;

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

Listing 37: Adding Input to Each Component of Matrix

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

7.8 Sum by Row

Listing 38: Find the Sum of Row Number 4

```
sum = 0;
row = 4;

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

7.9 Sum by Column

Listing 39: Find Sum of Individual Columns

```
// Sum of each individual row
for (row = 0; row < NUMBER_OF_ROWS; ++row) {
  sum = 0;

  for (col = 0; col < NUMBER_OF_COLUMNS; ++col) {
    sum += matrix[row][col];
  }

  cout << "Sum of row " << (row + 1) << " = " << sum << '\n';
}
```

7.10 Largest Element in Each Row and Each Column

Listing 40: Algorithm To Find Largest Element in Each Row

```
// Largest element in each row
for (row = 0; row < NUMBER_OF_ROWS; ++row) {
  largest = matrix[row][0]; // Assume the first element is largest

  for (col = 1; col < NUMBER_OF_COLUMNS; ++col) {
    if (matrix[row][col] > largest) {
      largest = matrix[row][col];
    }

  cout << "The largest element in row " << row << " = " << largest
       << '\n';
}
```

7.11 Passing Two-Dimensional Arrays as Parameters to Functions

Two-dimensional arrays are passed by reference as parameters to a function.

- The base address is passed to the formal parameter.
- Stored in **Row Order Form**.

When declaring the array as a formal parameter, omit the size of the first dimension, but not the second.

7.12 Array of Strings

Strings in C++ can be manipulated using either the data type `string` or character arrays (C-strings).

7.13 Array of Strings and the `string` Type

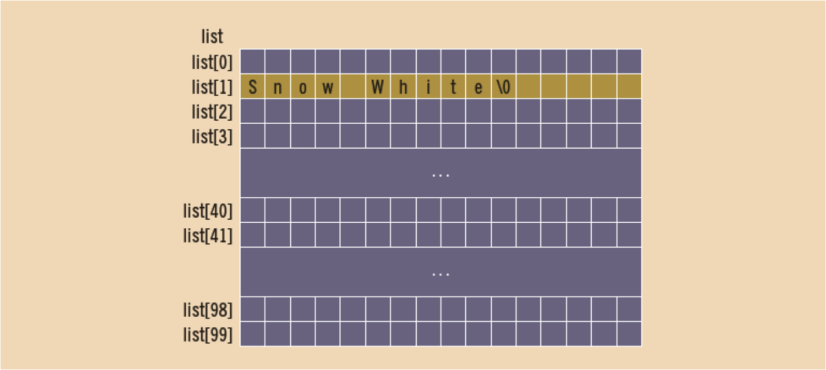
Listing 41: Array of Strings of 100 Elements

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

7.14 Array of Strings and C-Strings

Listing 42: C-String Array

```
char list[100][16];

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



7.15 Another Way to Declare a Two-Dimensional Array

Can use `using` (or `typedef`) to define a two-dimensional array data type:

Listing 43: `using` Two-Dimensional Array

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

using tableType = int[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];

// Declares array of 20 rows and 10 columns
tableType matrix;
```

7.16 Multi-Dimensional Arrays

N-Dimensional Array is a collection of a fixed number of elements arranged in n dimensions, $n >= 1$.

Listing 44: Multi-Dimensional Array Syntax

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

Listing 45: Accessing a Multi-Dimensional Array Component

```
arrayName[indexExp1][indexExp2]...[indexExpN]
```