Lecture 5 – Arrays

CST238 – Intro to Data Structures
YoungJoon Byun
ITCD

Lecture Objectives

- After completion of this lecture, you will be able to

Chapter 3: Data Structures and Abstract Data Types

- **<u>3.1 Data Structures, Abstract Data Types and Implementations (skip)</u>**
- 3.2 Static Arrays
- 3.3 Multidimensional Arrays
- 3.4 Dynamic Arrays
- 3.5 C-Style Structs
- 3.6 Procedural Programming

Chapter 3: Data Structures and Abstract Data Types

- 3.1 Data Structures, Abstract Data Types and Implementations
- **<u>3.2 Static Arrays</u>**
- 3.3 Multidimensional Arrays
- 3.4 Dynamic Arrays
- 3.5 C-Style Structs
- 3.6 Procedural Programming

Array

- A sequence of variables of the same data type.
- Why do we need an array?
  - To keep several variables of the same type together
  - Example: You can store final exam scores of 100 students in an array called *double score[100]*.
- Static array
  - A compiler can determine the memory size needed before starting the execution.

One Dimension Arrays

- Syntax

```
type arrayName [CAPACITY];
type arrayName [CAPACITY] =
                    { initializer_list };
```

- Example

```
double score[100];
int b[10]={0,11,22,33,44,55,66,77,88,99};
```

	b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	b[7]	b[8]	b[9]
b	0	11	22	33	44	55	66	77	88	99

Accessing an Array Element

- To access an element of an array, a programmer should provide the array name and index.
- First element always starts from index 0.

```
int score[45]
score[0] = 76;
score[10] = 89;
score[-1] = 99;        // Not good.
score [45] = 100;      // Not good.
// But the compiler doesn't produce an error
message for the out of range problem.
```


Subscript Operation – [ ]

- Consider the **[ ]** to be an operator
  - It performs address translation.
- In fact, an **array name** is the **starting address of an array**.
  - In other words, an array name is a constant that indicates the base address of an array.

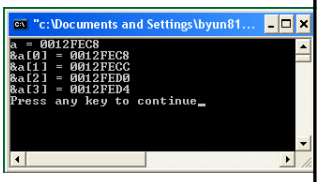
Example – Subscript Operation

```
int a[10];
```

Example

```
1.  #include <iostream>
2.  using namespace std;

3.  int main()
4.  {
5.      int a[4];
6.
7.      cout << "a = " << a << endl;
8.      cout << "&a[0] = " << &a[0] << endl;
9.      cout << "&a[1] = " << &a[1] << endl;
10.     cout << "&a[2] = " << &a[2] << endl;
11.     cout << "&a[3] = " << &a[3] << endl;
12.}
```



Passing an Array to a Function

- Example: determine an average of 25 students

```
1. int main () {
2.   double score[25], average;
3.   average = get_avg(score, 25);
4. }

5. // Function definition
6. double get_avg(double nums[], int size)
7. {
8.   …
9. }
```


Memory Layout

Rename array type using **`typedef`**

- You already learned
  typedef double real;
  typedef int * IntPtr;
- typedef for an array
  typedef type ArrayType[CAPACITY];
  ArrayType x; // same as type x[CAPACITY];
- Example

```
typedef int IntArray[100];
IntArray a;  // same as int a[100];
```


Out of Range Error

- Most C++ compilers do not by default check indices for out of range
- Results of out of range array access
  – Program can exceed allowed memory area
  – Program can give puzzling results

Example – Out of Range Error /* Fig. 3.5 */

```
#include <iostream>
using namespace std;

const int CAPACITY = 4;
typedef int IntArray[CAPACITY];

void read(IntArray theArray, int capacity, int numValues);
void display(IntArray theArray, int numValues);

int main() {
   IntArray a, b, c;

   cout << "Enter " << CAPACITY << " integer values for:\n";
   cout << "Array a: ";
   read(a, CAPACITY, CAPACITY);
   cout << "Array b: ";
   read(b, CAPACITY, CAPACITY);
   cout << "Array c: ";
   read(c, CAPACITY, CAPACITY);
```

```
cout << "The arrays are:\n";
cout << "a: ";    display(a, CAPACITY);
cout << "b: ";    display(b, CAPACITY);
cout << "c: ";    display(c, CAPACITY);

// Change array elements in b with out-of-range indices.
int below = -3, above = 6;
b[below] = -999;
b[above] = 999;

cout << "\nThe arrays after out-of-range errors are:\n";
cout << "a: ";    display(a, CAPACITY);
cout << "b: ";    display(b, CAPACITY);
cout << "c: ";    display(c, CAPACITY);
cout << endl;
}
```



```
void read(IntArray theArray, int capacity, int numValues)
{
   for (int i = 0; i < numValues; i++)
      cin >> theArray[i];
}

void display(int theArray[], int numValues)
{
  for (int i = 0; i < numValues; i++)
    cout << theArray[i] << "  ";
  cout << endl;
}
```


Problems with Static Arrays

- Capacity can't be changed.
- An array is not an object
  - (in the OOP sense)
- Virtually no predefined operations for non-**char** arrays.
- The Deeper Problem:
  - C-style arrays aren't self-contained
  - Data, functions, and size not encapsulated

Chapter 3: Data Structures and Abstract Data Types

- 3.1 Data Structures, Abstract Data Types and Implementations
- 3.2 Static Arrays
- **3.3 Multidimensional Arrays (will be covered later)**
- 3.4 Dynamic Arrays
- 3.5 C-Style Structs
- 3.6 Procedural Programming

Summary

- Static arrays (chap. 3.2)
- Next Lecture
  - Dynamic arrays (chap. 3.4)

References

- Larry Nyhoff, *ADTs, Data Structures, and Problem Solving with C++*, 2nd Edition, Prentice-Hall, 2005
- Walter Savitch, *Problem Solving with C++*, 6th Edition, Addison-Wesley, 2006
- Dr. Meng Su's Lecture Notes
  http://cs.bd.psu.edu/~mus11/122Fa06/cse122Fa06.htm