Data Structures and Algorithms
(資料結構與演算法)

Lecture 1: Array

Hsuan-Tien Lin (林軒田)

htlin@csie.ntu.edu.tw

Department of Computer Science
& Information Engineering

National Taiwan University
(國立台灣大學資訊工程系)

Intuition behind Array

```
1   int CabnetWithNumber[128];
```

Array as Data Structure

(one-dimensional) array

- holds a list of N elements
- allows users to get the i-th element (efficiently)
- allows users to put to the i-th location (efficiently)

—i called **index**

Basic "Operations" on Data Structure

- construct/destruct
- getBlahBlah
- dynamic operations: sometimes needing **maintain**
  - putBlahBlah
  - removeBlahBlah

Biggest Physical Array

MEMORY!

C/C++ Implementation of Array

```
1 int dense[10] = {1, 3, 0, 0, 0, 0, 0, 0, 0, 2};
```

- dense array: store everything consecutively
  - space: $N * (elem.size)$ for a length-N array
  - getByIndex: constant
  - putByIndex: constant
  - construct: (almost) constant
  - removeByIndex: fill-in or not?

Game Scoreboard Problem

Ordered Array as Data Structure

(one-dimensional) array that

- holds a list of N elements, ordered consecutively by **key**
- allows users to getByKey (efficiently)
- allows users to putByKey (efficiently)

- construct/destruct
- getByKey
- putByKey
- removeByKey (?)

Game Scoreboard with Ordered Array

Binary Search Algorithm for getByKey

Insertion Algorithm for putByKey

Insertion Sort

2D Array as Data Structure

(rectangular) 2-D array

- object specification: (*index*, *element*) pairs with *index* $\in \{(0,0),(0,1),\cdots,(N-1,M-1)\}$
- action specification: get(*index*); put(*index*, *element*); create(N, M), etc.

2D Array by 1D Array: Manual Index Conversion

- object representation: a block of consecutive memory of size $N * M$, with a chunk representing each *element* for each *index*
- action implementation:

```
1 #define N (100) //or "similarly" const int N = 100;
2 #define M (200)
3 int* twodim = new int[N*M];
4
5 int get(int* arr, int n, int m)
6   { return arr[n*M + m]; }
```

2D Array by 1D Array: Automatic Folding

- object representation: a block of consecutive memory of size $N * M$, with a chunk representing each *element* for each *index*
- action implementation:

```
1 #define N (100)
2 #define M (200)
3 int twodim[N][M];
4
5 int get(int arr[][M], int n, int m)
6   { return arr[n][m];}
```

2D Array by 1D Array: Array of Arrays

2-D array by array of arrays in C

- object representation: *N* blocks of consecutive memory of size *M*
- action implementation:

```
1 #define N (100)
2 #define M (200)
3 int** twodim = new int*[N];
4 for(int n=0;n<N;n++)
5   twodim[n] = new int[M];
6 int get(int** arr, int n, int m)
7   { return arr[n][m];}
```

Comparison of Three Implementations in C

```
1 int* twodim = new int[N*M];
2 int twodim[N][M];
3   //also, int (*twodim)[M] = new int[N][M];
4 int** twodim = new int*[N]; // and ...
```

	1	2	3
space	$N * M$ integers	$N * M$ int.	$N * M$ int. + N pointers
type	`int*`	`int*[M]`	`int**`
create	constant	constant	prop. to N
retrieve	arithmetic+dereference	arith.+deref.	deref.+deref.

method 2 for static allocating (constant M); method 1 or 3 for dynamic allocating (your choice)

A Tale between Two Programs

```
1  int rowsum(){
2    int i, j;
3    int res = 0;
4    for(i=0;i<MAXROW;i++)
5      for(j=0;j<MAXCOL;j++)
6        res += array[i][j];
7  }
```

```
1  int colsum(){
2    int i, j;
3    int res = 0;
4    for(j=0;j<MAXCOL;j++)
5      for(i=0;i<MAXROW;i++)
6        res += array[i][j];
7  }
```