Arrays

Hsuan-Tien Lin

Dept. of CSIE, NTU

March 5, 2013

Arrays: from Implementation to Abstraction

C++ Implementation View

(One-dimensional) array is a block of consecutive memory that

- holds a list of N elements
- allows users to **get** the k-th element
- allows users to **put** to the k-th location
- "constant" time to **construct** (`malloc`, `new`)
- nothing much to **maintain**

An Abstract View

Abstract (one-dimensional) array

- holds a list of N elements
- allows users to get the k-th element
- allows users to put to the k-th location
- construction and maintenance?

Dense Array

dense implementation of the abstract array

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

- dense array: store everything (consecutively), needs 10 positions
  - space: $N * (elem.size)$ for a length-N array
  - get: constant
  - put: constant
  - construct: constant

YOUR MEMORY

- pointer: index to the array location
- type of pointer: # bytes (consecutive slots) to be fetched

Sparse Array

```
int dense[10] = {1, 3, 0, 0, 0, 0, 0, 0, 0, 2};
int sparse[3][2] = {{0, 1}, {1, 3}, {9, 2}};
```

- sparse array: store only non-zero (index, element) pairs, needs 3 pairs
  - space: $E * (indexsize + elem.size)$ for E elements, better than $N * (elem.size)$ if E small
  - get: ordered — ???; non-ordered — ???
  - put: ???
  - construct: ???

note: often use **array** to mean dense array only

STL Vector: A Dense Array that Dynamically Grows

learn about its use now (very useful),
discuss about the actual implementation later

2-D Array: by 1-D Array

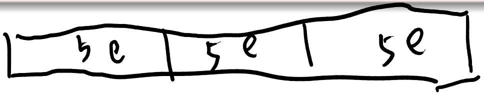
abstract rectangular 2-D array

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

2-D array by 1-D array in C

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

3 × 5



2-D Array by 1-D Array

```
#define N (100) //or "similarly" const int N = 100;
#define M (200)
int* twodim = new int[N*M];

int get(int* arr, int n, int m)
  { return arr[n*M + m]; }
```

2-D Array: by 1-D Array with Constant Folding

abstract 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*); construct(*N*, *M*), etc.

2-D array by 1-D array with constant folding in C

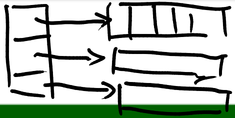
- object representation: a block of consecutive memory of size $N * M$, with a chunk representing each *element* for each *index*
- action implementation:

2-D Array: by 1-D Array with Constant Folding

```c
#define N (100)
#define M (200)
int twodim[N][M];

int get(int arr[][M], int n, int m)
  { return arr[n][m];}
```

2-D Array: by Array of Arrays



abstract 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*); construct(*N*, *M*), etc.

2-D array by array of arrays in C

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

2-D Array: by Array of Arrays

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