Arrays

Hsuan-Tien Lin

Dept. of CSIE, NTU

March 10, 2015

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 retrieve the k-th element
- allows users to store to the k-th location

An Abstract View

Abstract (one-dimensional) array

- holds a list of N elements
- allows users to retrieve the k-th element
- allows users to store to the k-th location

different implementations:
different space/time trade-off

Dense Array

dense implementation of the abstract array

```
1 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
  - retrieving: constant
  - storing: constant
  - creating: constant

Sparse Array

```
1 int dense[10] = {1, 3, 0, 0, 0, 0, 0, 0, 0, 2};
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
  - retrieving: ordered — ???; non-ordered — ???
  - storing: ???
  - creating: ???

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:
  retrieve(*index*); store(*index*, *element*); create(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:

2-D Array by 1-D Array

```
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]; }
```

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: retrieve(*index*); store(*index*, *element*); create(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

```
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];}
```

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: retrieve(*index*); store(*index*, *element*); create(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

```
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

```
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  }
```