Arrays

Hsuan-Tien Lin

Dept. of CSIE, NTU

March 6, 2012

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

```
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

```c
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
  - retrieving: ordered — ???; non-ordered — ???
  - storing: ??? binary-search seq-search
  - 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

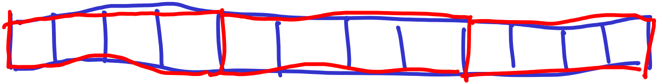
2-D Array: by 1-D Array

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

```
#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: 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

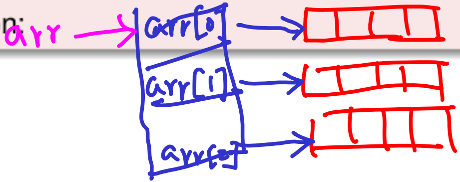
```
#define N (100)
#define M (200)
int twodim[N][M];

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

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

```
#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];}
```

Comparison of Three Implementations

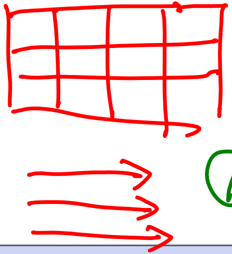
```
int* twodim = new int[N*M];
int twodim[N][M];
  //also, int (*twodim)[M] = new int[N][M];
int** twodim = new int*[N]; // and ...
```

	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.

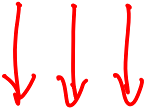
method 2 for static allocating (constant M); method 1 or 3 for dynamic allocating (your choice)

A Tale between Two Programs

```
int rowsum(){
  int i, j;
  int res = 0;
  for(i=0;i<MAXROW;i++)
    for(j=0;j<MAXCOL;j++)
      res += array[i][j];
} return res;
```



```
int colsum(){
  int i, j;
  int res = 0;
  for(j=0;j<MAXCOL;j++)
    for(i=0;i<MAXROW;i++)
      res += array[i][j];
}
```



Sparse Matrix

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

Specialty

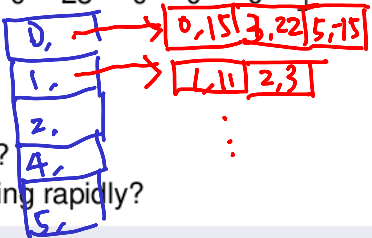
a rectangular 2-D array that contains many common elements (0) that we may not want to repeatedly store

Data Structures for Sparse Matrix

- dense implementation: as 2D dense arrays
- array of array implementation:
  - “(dense 1D) of (sparse 1D)”
  - “(sparse 1D) of (sparse 1D)”
- ordered triples implementation: see next page

Ordered Triples Implementation

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$



Ordered(-by-row-then-by-col) Triples

6	6	8
row	*col*	*value*
0	0	15
0	3	22
0	5	-15

- space?
- retrieving rapidly?

simple exercise: compare to unordered triple implementation