Arrays and Structures

Data Structures

Ching-Fang Hsu
Department of Computer Science and Information Engineering
National Cheng Kung University

The Array As An Abstract Data Type

- An array is a set of pairs, <index, value>, 範圍 such that each index that is defined has a value associated with it.
  - A *correspondence* or a *mapping*
  - A homogeneous aggregate of data elements 同質性元素組合
- Standard operations provided by most languages (p.52, ADT 2.1)
  - Array creation
  - Value retrieval 讀取array（名稱，值）
  - Value setting 寫入Array（名稱，值，新的參數）

ADT *Array* is

objects: A set of pairs <*index*, *value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions, for example, $\{0, \cdots, n-1\}$ for one dimension, $\{(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)\}$ for two dimensions, etc.

functions:

for all $A \in Array$, $i \in index$, $x \in item$, j, $size \in$ integer

Array Create(j, *list*)	::=	**return** an array of j dimensions where *list* is a j-tuple whose ith element is the the size of the ith dimension. *Items* are undefined.
Item Retrieve(A, i)	::=	**if** ($i \in index$) **return** the item associated with index value i in array A **else return** error
Array Store(A,i,x)	::=	**if** (i in *index*) **return** an array that is identical to array A except the new pair <i, x> has been inserted **else return** error.

end *Array*

ADT 2.1: Abstract Data Type *Array*

The Array As An Abstract Data Type (contd.)

- The implementation of one-dimensional arrays in C
  - When the compiler encounters an declaration for an array with type `t` and size n, it allocates n aconsecutive memory locations, where each one is large enough to hold a type `t` value.
  - The base address α -- the address of the first element of an array
    - The address of the i-th element = $\alpha + (i\text{-}1) * \text{sizeof}(t)$
    - In C, we do not multiply the offset i and sizeof (`t`) to get the appropriate element of the array.

The Array As An Abstract Data Type (contd.)

- `list[i] ≡ *(list + i)` list中第i個元素取值
- Dereferencing -- the pointer is interpreted as an indirect reference （先找出所在位置，再取值）
  - p. 54, Program 2.2

```
void print1(int *ptr, int rows)
{/* print out a one-dimensional array using a pointer */
   int i;
   printf("Address Contents\n");
   for (i = 0; i < rows; i++)
      printf("%8u%5d\n", ptr + i, *(ptr + i));
   printf("\n");
}
```

Program 2.2: One-dimensional array accessed by address

The Polynomial Abstract Data Type

- Ordered / linear lists
  - $(item_0, item_1, \ldots, item_{n-1})$
  - Operations on lists (p. 65)
- Representing an ordered list as an array
  - Associate $item_i$ with the array index i. ➪ a sequential mapping 循序對應：以連續的記憶體空間來儲存陣列
    連續
  - Sequential mapping works well for most operations listed in page 65 in constant time, except insertion and deletion. 會遇到data removal的問題
    Sequential mapping 不適
    - A motivation that leads us to consider nonsequential mappings 使用link list儲存

- Finding the length, n, of a list.
- Reading the items in a list from left to right (or right to left).
- Retrieving the ith item from a list, $0 \leq i < n$.
- Replacing the item in the ith position of a list, $0 \leq i < n$.
- Inserting a new item in the ith position of a list, $0 \leq i \leq n$. The items previously numbered $i, i+1, \cdots, n-1$ become items numbered $i+1, i+2, \cdots, n$.
- Deleting an item from the ith position of a list, $0 \leq i < n$. The items numbered $i+1, \cdots, n-1$ become items numbered $i, i+1, \cdots, n-2$.

The Polynomial Abstract Data Type (contd.)

- Example: Build a set of functions for manipulation of symbolic polynomials
  - ADT (p.67, ADT 2.2)
- For simplifying operations, exponents are arranged in decreasing order.
  - Operation Add can be achieved by comparing terms from the two polynomials until one or both of the polynomials becomes empty.
    - Initial version of `padd` function (p. 68, Program 2.5)

ADT *Polynomial* is

objects: $p(x) = a_1 x^{e_1} + \cdots + a_n x^{e_n}$; a set of ordered pairs of $<e_i, a_i>$ where a_i in *Coefficients* and e_i in *Exponents*, e_i are integers >= 0

functions:

for all *poly*, *poly*1, *poly*2 $\in$ *Polynomial*, *coef* $\in$ *Coefficients*, *expon* $\in$ *Exponents*

Polynomial Zero()	::=	**return** the polynomial, $p(x) = 0$
Boolean IsZero(*poly*)	::=	**if** (*poly*) **return** *FALSE* **else return** *TRUE*
Coefficient Coef(*poly*,*expon*)	::=	**if** (*expon* $\in$ *poly*) **return** its coefficient **else return** zero
Exponent LeadExp(*poly*)	::=	**return** the largest exponent in *poly*
Polynomial Attach(*poly*, *coef*, *expon*)	::=	**if** (*expon* $\in$ *poly*) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted
Polynomial Remove(*poly*, *expon*)	::=	**if** (*expon* $\in$ *poly*) **return** the polynomial *poly* with the term whose exponent is *expon* deleted **else return** error
Polynomial SingleMult(*poly*, *coef*, *expon*)	::=	**return** the polynomial $poly \cdot coef \cdot x^{expon}$
Polynomial Add(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 + *poly*2
Polynomial Mult(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 $\cdot$ *poly*2

end *Polynomial*

ADT 2.2: Abstract data type *Polynomial*

```
/* d = a + b, where a, b, and d are polynomials */
d = Zero()
while (! IsZero(a) && ! IsZero(b)) do {
   switch COMPARE(LeadExp(a), LeadExp(b)) {
      case -1: d =
         Attach(d,Coef(b,LeadExp(b)),LeadExp(b));
         b = Remove(b,LeadExp(b));
         break;
      case 0: sum = Coef( a, LeadExp(a))
                  + Coef(b, LeadExp(b));
         if (sum) {
            Attach(d,sum,LeadExp(a));
            a = Remove(a,LeadExp(a));
            b = Remove(b,LeadExp(b));
         }
         break;
      case 1: d =
         Attach(d,Coef(a,LeadExp(a)),LeadExp(a));
         a = Remove(a,LeadExp(a));
   }
}
insert any remaining terms of a or b into d
```

Program 2.5: Initial version of *padd* function

The Polynomial Abstract Data Type -- Representation

- Option 1 (p. 66~68)
  - Maximum degree is restricted by `MAX_DEGREE`.

```
#define MAX_DEGREE 101
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
        } polynomial;
```

Array index當exponent使用

  - The main drawback : lower flexibility on space requirement
    - Wasting a lot of space when the degree of the polynomial is much less then `MAX_DEGREE` or the polynomial is sparse

The Polynomial Abstract Data Type -- Representation (contd.)

此方法適用於項數少，次方變化大者，若多都項式是連續的話，將很耗費記憶體空間

- Option 2 (p. 68~69)
  - Representing $a_i x^i$ as a structure and using only one global array of this structure to store all polynomials (p. 68~69)

```
#define MAX_TERMS 100
typedef struct {
        float coef;
        int expon;
        } polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

The Polynomial Abstract Data Type -- Representation (contd.)

$$A(x) = 2x^{1000} + 1 \qquad B(x) = x^4 + 10x^3 + 3x^2 + 1$$

只記有效項，不存在的不花空間配置

	startA	*finishA*	*startB*			*finishB*	*avail*
coef	2	1	1	10	3	1	
expon	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

The Polynomial Abstract Data Type -- Representation (contd.)

- No limit on the number of polynomials stored in the global array
- The index of the first (last) term of polynomial A is given by `starta` (`finisha`).
  - `finisha` = `starta` + *n* - 1, if A has n nonzero terms
- The index of the next free location in the array is given by `avail`.
- The main drawback: About twice as much space as option 1 is needed when all the terms are nonzero.
- The revised function `padd` (p. 70, Program 2.6)

```c
void padd(int startA,int finishA,int startB, int finishB,
                                 int *startD,int *finishD)
{/* add A(x) and B(x) to obtain D(x) */
  float coefficient;
  *startD = avail;
  while (startA <= finishA && startB <= finishB)
    switch(COMPARE(terms[startA].expon,
                   terms[startB].expon)) {
      case -1: /* a expon < b expon */
            attach(terms[startB].coef,terms[startB].expon);
            startB++;
            break;
      case 0: /* equal exponents */
            coefficient = terms[startA].coef +
                          terms[startB].coef;
            if (coefficient)
              attach(coefficient,terms[startA].expon);
            startA++;
            startB++;
            break;
      case 1: /* a expon > b expon */
            attach(terms[startA].coef,terms[startA].expon);
            startA++;
  }
```
如果A剩下，for執行m次，如果是B剩下，for執行n次
```c
  /* add in remaining terms of A(x) */
  for(; startA <= finishA; startA++)
    attach(terms[startA].coef,terms[startA].expon);
  /* add in remaining terms of B(x) */
  for( ; startB <= finishB; startB++)
    attach(terms[startB].coef, terms[startB].expon);
  *finishD = avail-1;
}
```

Program 2.6: Function to add two polynomials

The Polynomial Abstract Data Type -- Representation (contd.)

- Analysis of Program 2.6
  - Each iteration of the while-loop: $\mathrm{O}(1)$
  - The number of iterations: bounded by $\overset{A}{m} + \overset{B}{n} - 1 \Rightarrow \mathrm{O}(n+m)$
    - The worst case (p. 71)
  - The time for two for-loops: bounded by $\mathrm{O}(n+m)$

$\Rightarrow$ The asymptotic time of the algorithm for operation Add is $\mathrm{O}(n+m)$.

The Sparse Matrix Abstract Data Type

使用一維array儲存sparse matrix，節省記憶體空間

- A matrix containing many zero entries is called a *sparse matrix*.
  - Difficult to determine exactly whether a matrix is sparse or not
- The standard representation of a matrix is a two-dimensional array, but not appropriate for a sparse matrix due to a waste of space.
  - Storing only non-zero elements is a feasible solution for a sparse matrix.

The Sparse Matrix Abstract Data Type (contd.)

- A minimal set of matrix operations
  - Creation
  - Addition
  - Transpose
  - Multiplication
- The ADT of a sparse matrix (p. 74, ADT 2.3)
- Using the triple <*row*, *col*, *value*> to characterize an element within a matrix.
  - A sparse matrix $\equiv$ an array of triples

ADT *SparseMatrix* is

objects: a set of triples, <*row*, *column*, *value*>, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item*.

functions:

for all $a, b \in SparseMatrix$, $x \in item$, $i, j, maxCol, maxRow \in index$

SparseMatrix Create(*maxRow*, *maxCol*) ::=
: **return** a *SparseMatrix* that can hold up to $maxItems = maxRow \times maxCol$ and whose maximum row size is *maxRow* and whose maximum column size is *maxCol*.

SparseMatrix Transpose(a) ::=
: **return** the matrix produced by interchanging the row and column value of every triple.

SparseMatrix Add(a, b) ::=
: **if** the dimensions of a and b are the same **return** the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.
: **else return** error

SparseMatrix Multiply(a, b) ::=
: **if** number of columns in a equals number of rows in b
: **return** the matrix d produced by multiplying a by b according to the formula: $d[i][j] = \sum(a[i][k] \cdot b[k][j])$ where $d(i, j)$ is the (i, j)th element
: **else return** error.

ADT 2.3: Abstract data type *SparseMatrix*

The Sparse Matrix Abstract Data Type (contd.)

- For efficient transpose operation, the triples are ordered by rows and within rows by columns.
- With the triple definition, the number of rows and columns, and the number of nonzero elements, the Create operation can be derived (p. 75).

The Sparse Matrix Abstract Data Type -- Transposing a Matrix

- A simple algorithm for transposing

```
for all elements in column j
    place element <i, j, value> in
    element <j, i, value>
```

  - p. 77, Program 2.8
  - Time complexity: $O(columns \cdot elements)$
  - cf. $O(rows \cdot columns)$ with a two-dimensional array representation

```
for (j = 0; j < columns; j++)
  for (i = 0; i < rows; i++)
     b[j][i] = a[i][j];
```

```
void transpose(term a[], term b[])
{/* b is set to the transpose of a */
   int n,i,j, currentb;
   n = a[0].value;          /* total number of elements */
   b[0].row = a[0].col; /* rows in b = columns in a */
   b[0].col = a[0].row; /* columns in b = rows in a */
   b[0].value = n;
   if (n > 0 )   { /* non zero matrix */
      currentb = 1;
      for (i = 0; i < a[0].col; i++)
      /* transpose by the columns in a */
         for (j = 1; j <= n; j++)
         /* find elements from the current column */
            if (a[j].col == i) {
            /* element is in current column, add it to b */
               b[currentb].row = a[j].col;
               b[currentb].col = a[j].row;
               b[currentb].value = a[j].value;
               currentb++;
            }
   }
}
```

缺點：對非0元素沒排開

Program 2.8: Transpose of a sparse matrix

$$A = \begin{bmatrix} 15 & X & X & 22 & X & -15 \\ X & 11 & 3 & X & X & X \\ X & X & X & -6 & X & X \\ X & X & X & X & X & X \\ 91 & X & X & X & X & X \\ X & X & 28 & X & X & X \end{bmatrix} \qquad A^T = \begin{bmatrix} 15 & X & X & X & 91 & X \\ X & 11 & X & X & X & X \\ X & 3 & X & X & X & 28 \\ 22 & X & -6 & X & X & X \\ X & X & X & X & X & X \\ -15 & X & X & X & X & X \end{bmatrix}$$

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

(1)
(3)

Program 2.8

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

(2)

每1行非零元素的個數

	[0]	[1]	[2]	[3]	[4]	[5]
rowTerms =	2	1	2	2	0	1
startingPos =	1	3	4	6	8	8

+ +

固定將第一行需轉置至列的元素記為 startingPos[0] = 1，其餘 startingPos 看下一個rowTerms得知

Program 2.9

The Sparse Matrix Abstract Data Type -- Transposing a Matrix (contd.)

- The $O(columns \cdot elements)$ time becomes $O(columns^2 \cdot rows)$ when the number of elements is of the order *columns* · *rows*.
  - An improved version: *fast_transpose* (p. 78, Program 2.9) with $O(columns + elements)$ complexity

```
void fastTranspose(term a[], term b[])
{/* the transpose of a is placed in b */
  int rowTerms[MAX_COL], startingPos[MAX_COL];
  int i,j, numCols = a[0].col, numTerms = a[0].value;
  b[0].row = numCols;  b[0].col = a[0].row;
  b[0].value = numTerms;
  if (numTerms > 0) { /* nonzero matrix */
    for (i = 0; i < numCols; i++)
      rowTerms[i] = 0;
    for (i = 1; i <= numTerms; i++)
      rowTerms[a[i].col]++;
    startingPos[0] = 1;
    for (i = 1; i < numCols; i++)
      startingPos[i] =
                 startingPos[i-1] + rowTerms[i-1];
    for (i = 1; i <= numTerms; i++) {
      j = startingPos[a[i].col]++;
      b[j].row = a[i].col;   b[j].col = a[i].row;
      b[j].value = a[i].value;
    }
  }
}
```

The initialization of rowTerms

The calculation of nonzero number of terms per row of b

Program 2.9: Fast transpose of a sparse matrix

The Sparse Matrix Abstract Data Type -- Transposing a Matrix (contd.)

- Analysis of Program 2.9
  - The 1st for-loop: $O(columns)$
    - `row_terms` initialization
  - The 2nd for-loop: $O(elements)$
    - calculating # of non-zero elements within each column
  - The 3rd for-loop: $O(columns)$
    - starting positions calculations
  - The 4th for-loop: $O(elements)$
    - value setting for array `b`

$\Rightarrow$ The time complexity of *fast_transpose* is $O(columns + elements)$.

The Sparse Matrix Abstract Data Type -- Matrix Multiplication

- **Definition**: Given A and B where A is $m \times n$ and B is $n \times p$, the <i, j> element of the product matrix D is

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

- Step 1: Compute the transpose of B.
- Step 2: Do a merge operation similar to that used in the polynomial addition.
- p. 81~82, Program 2.10, 2.11
  - The for-loop: $O(\sum_{row} (colsB \bullet termsRow + totalB))$

$$= O(colsB * totalA + rowsA * totalB)$$

```
void mmult(term a[], term b[], term d[])
{ /* multiply two sparse matrices */
  int i, j, column, totalB = b[0].value, totalD = 0;
  int rowsA = a[0].row, colsA = a[0].col,
  totalA = a[0].value; int  colsB = b[0].col,
  int rowBegin = 1, row = a[1].row, sum = 0;
  int newB[MAX_TERMS][3];
  if (colsA != b[0].row) {
    fprintf(stderr,"Incompatible matrices\n");
    exit(EXIT_FAILURE);
  }
  fastTranspose(b,newB);
  /* set boundary condition */
  a[totalA+1].row = rowsA;
  newB[totalB+1].row = colsB;
  newB[totalB+1].col = 0;
  for (i = 1; i <= totalA; ) {
    column = newB[1].row;
    for (j = 1; j <= totalB+1;) {
    /* multiply row of a by column of b */
      if (a[i].row != row) {
        storeSum(d,&totalD,row,column,&sum);
        i = rowBegin;
        for (; newB[j].row == column; j++)
          ;
        column = newB[j].row;
      }
      else if (newB[j].row != column) {
        storeSum(d, &totalD, row, column, &sum);
        i = rowBegin;
        column = newB[j].row;
      }
      else switch (COMPARE(a[i].col, newB[j].col)) {
        case -1: /* go to next term in a */
              i++; break;
        case 0: /* add terms, go to next term in a and b*/
              sum += ( a[i++].value * newB[j++].value);
              break;
        case 1 : /* advance to next term in b */
              j++;
      }
```

The starting index of the currently processed row of A

The index of the currently processed of A

`fastTranspose(b,newB);` — O(colsB + totalB)

`for (i = 1; i <= totalA; ) {` — Max. # of iterations = colsB + colsB * termsRow + totalB

`for (j = 1; j <= totalB+1;) {` — O(colsB * termsRow + totalB)

`i = rowBegin;` — At most colsB times

`for (; newB[j].row == column; j++)` — O(totalB)

resetting (`i = rowBegin;` … `column = newB[j].row;`)

resetting (`i = rowBegin;` `column = newB[j].row;`)

`i++; break;` — Maximum increment is colsB * termsRow

`break;` (case 0) — Maximum increment is totalB

```
        }  /* end of for j <= totalB+1 */
        for (; a[i].row == row; i++)   O( totalB )
          ;
        rowBegin = i; row = a[i].row;
    } /* end of for i<=totalA */
    d[0].row = rowsA;
    d[0].col = colsB; d[0].value =  totalD;
}
```

Program 2.10: Sparse matrix multiplication

```
void storeSum(term d[], int *totalD, int row, int column,
                                     int *sum)
{/* if *sum != 0, then it along with its row and column
    position is stored as the *totalD+1 entry in d */
   if (*sum)
      if (*totalD < MAX_TERMS) {
         d[++*totalD].row = row;
         d[*totalD].col = column;
         d[*totalD].value = *sum;
         *sum = 0;
      }
      else {
         fprintf(stderr,"Numbers of terms in product
                                 exceeds %d\n",MAX_TERMS);
         exit(EXIT_FAILURE);
      }
}
```

Program 2.11: *storeSum* function

$$
\begin{array}{c c}
 & \begin{array}{cccccc} \texttt{i} & & & \texttt{i} & & \texttt{i} \end{array} \\
\begin{array}{c} \texttt{row} \rightarrow \\ \\ A = \\ \\ \\ \end{array} & \begin{bmatrix} 15 & X & X & 22 & X & -15 \\ X & 11 & 3 & X & X & X \\ X & X & X & -6 & X & X \\ 91 & X & X & X & X & X \\ X & X & 28 & X & X & X \end{bmatrix}
\end{array}
$$

$$
\begin{array}{c c}
 & \begin{array}{cccccc} \texttt{j} & \texttt{j} & & & \texttt{j} & \end{array} \\
\begin{array}{c} \\ \texttt{column} \rightarrow \\ \\ B^T = \\ \\ \\ \\ \end{array} & \begin{bmatrix} X & X & X & X & X & X \\ X & 7 & X & X & -9 & X \\ -1 & X & X & 13 & X & X \\ X & X & 23 & X & X & 2 \\ X & X & X & X & X & X \\ X & -5 & X & 12 & 5 & 3 \\ X & X & 6 & X & X & X \end{bmatrix}
\end{array}
$$

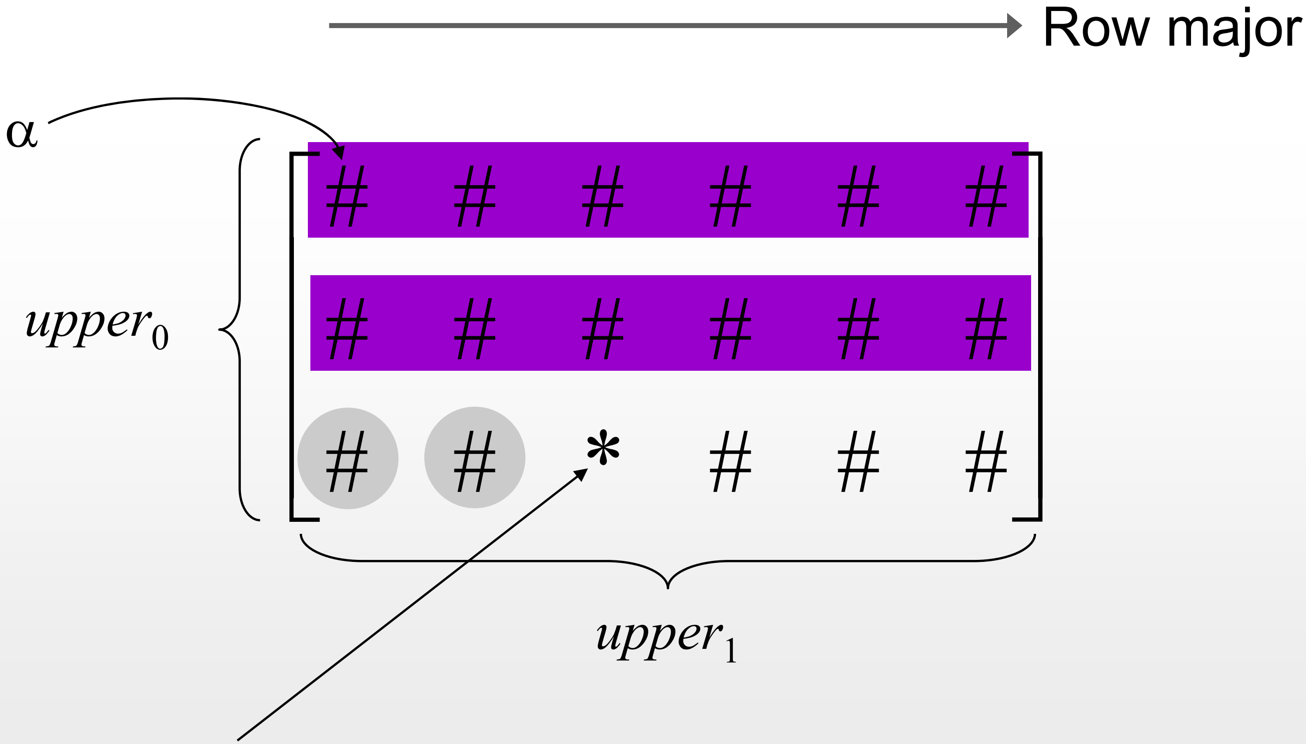
rowBegin

Representation of Multidimensional Arrays

用於計算memory對應位置（compiler）

- Two common ways
  - Row major order 先走列再換行（由左至右，由上至下） row index小的先存
    - Storing multidimensional arrays by rows
  - Column major order 先走行再換列（由上至下，由左至右） column index小的先存
- Assume that α is the starting address of a n-dimensional array $A[upper_0][upper_1]\ldots[upper_{n-1}]$.
  - The address for $A[i_0][i_1]\ldots[i_{n-1}]$ is: 在n維中，a後就需+n項

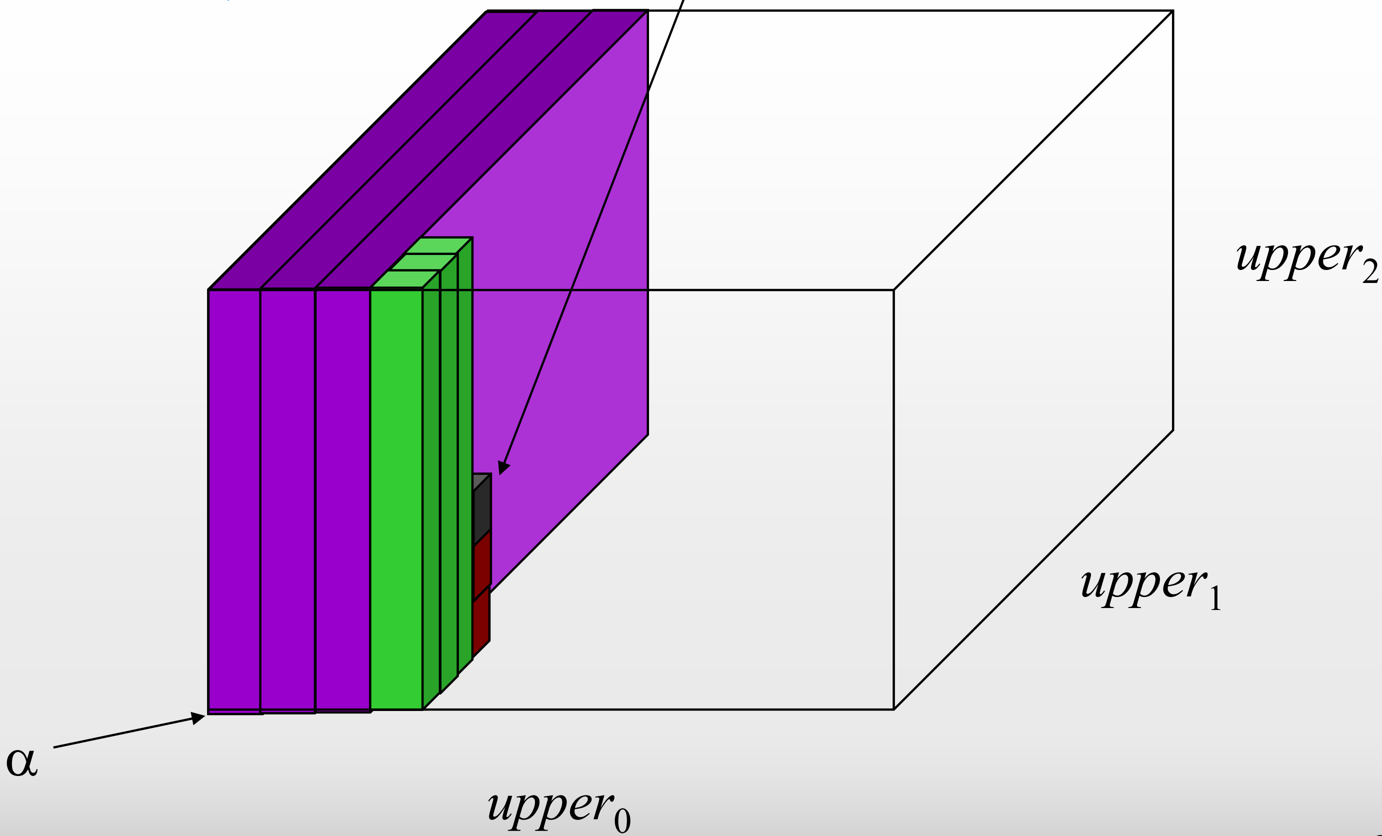
$$\alpha + \sum_{j=0}^{n-1} i_j a_j \text{ where } a_j = \begin{cases} \prod_{k=j+1}^{n-1} upper_k & 0 \le j < n-1 \\ 1 & j = n-1 \end{cases}$$



$$A[i_0][i_1] \rightarrow \alpha + \underline{i_0 * upper_1} + \underline{i_1 * 1}$$

$$A[i_0][i_1][i_2] \rightarrow \alpha + i_0 * upper_1 * upper_2 + i_1 * upper_2 + i_2 * 1$$

Row Major



Representation of Multidimensional Arrays (contd.)

- A compiler will initially take the declared bounds (i.e., $upper_k$, $0 \le k \le n\text{-}1$) and use them to compute the constants a_j, $0 \le j \le n\text{-}2$.
- The computation of the address of $A[i_0][i_1]\ldots[i_{n\text{-}1}]$ requires n-1 more multiplications and n additions.