Arrays and Structures

Data Structures

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

An Abstract Data Type

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,…, $n-1$} 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 *i*th element is the size of the *i*th 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*

An Abstract Data Type (contd.)

- The implementation of one-dimensional arrays in C
  - When the compiler encounters an array declaration with type `t` and size n, it allocates n consecutive 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 = α + (i-1) $*$ sizeof (`t`)
    - In C, we do not multiply the offset i and sizeof (`t`) to get the appropriate element of the array.

An Abstract Data Type (contd.)

- `list[i]` ≡ `*(list + i)`
- Dereferencing -- the pointer is interpreted as an indirect reference
  - p. 54, Program 2.2

```
void print1( int *ptr, int rows)
{
    int i;
    printf("Address Contents\n");
    for( i = 0; i < rows; i++ )
       printf("%8u%5d\n", ptr+i, *(ptr+i));
    printf("\n");
}
```

dereferencing

The Polynomial Abstract Data Type

- Ordered / linear lists
  - $(item_0, item_1, \ldots, item_{n-1})$
  - Operations on lists (p. 65)

•Length determination
•Scanning from left to right
•Item value retrival/setting
•Item insertion
•Item deletion

- Representing an ordered list as an array
  - Associate $item_i$ with the array index i.
    - ⇨ A sequential mapping
  - Sequential mapping works well for most operations listed in page 65 in constant time, except insertion and deletion.
    - A motivation that leads us to consider nonsequential mappings

ADT *Polynomial* is

objects: $p(x) = a_1 x^{e_1} + \quad + a_n x^{e_n}$; 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 ∈*Polynomial*, *coef* ∈*Coefficients*, *expon*∈*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* ∈ *poly*) **return** its coefficient
else return 0

Exponent LeadExp(*poly*) ::= **return** the largest exponent in *poly*

Polynomial Attach(*poly*, *coef*, *expon*) ::= **if** (*expon* ∈ *poly*) **return** error
else return the polynomial *poly*
with the term <*coef*, *expon*> inserted

Polynomial Remove(*poly*, *expon*) ::= **if** (*expon* ∈ *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 $poly1 + poly2$

Polynomial Mult(*poly*1, *poly*2) ::= **return** the polynomial $poly1 \cdot poly2$

end *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:
            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:
            Attach(d, Coef(a,LeadExp(a)), LeadExp(a));
            a = Remove(a,LeadExp(a));
    }
}
insert any remaining terms of a or b into d
```

Two non-zero polynomials

Comparison between two leading terms

Output and remove b's leading term

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;
```

  - If `a` is of type *`polynomial`* and `n` < `MAX_DEGREE`, $A(x) = \sum_{i=0}^{n} a_i x^i$ can be represented as

```
a.degree = n
a.coef[i] = a_{n-i}, 0 ≤ i ≤ n
```

The Polynomial Abstract Data Type – Representation (contd.)

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

The Polynomial Abstract Data Type – Representation (contd.)

- 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.)

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

	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

透過start 和finish 來分隔每一個多項式，將所有多項式用一個陣列存就可以了

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

若矩陣之間較為稀疏，使用第二種方法紀錄矩陣較省空間。然而若矩陣本身若太為密集，
例如為x^1000+x^999+x^998+....此時要花費太多exponent的空間儲存此係數對應到的次方

- 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)

```
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++;
            break;
      }
      /* 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;
}
```

Two non-zero polynomials

Comparison between two leading terms

Output and remove b's leading term

Append remaining terms to the resulting polynomial

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

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

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

The Sparse Matrix Abstract Data Type

- 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.

ADT *SparseMatrix* is

objects: a set of triples, $\boxed{\langle row, column, value\rangle}$, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item*.

functions:

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

$\boxed{\textit{SparseMatrix}\ \text{Create}(maxRow, maxCol)}$ ::=
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*.

$\boxed{\textit{SparseMatrix}\ \text{Transpose}(a)}$ ::=
return the matrix produced by interchanging the row and column value of every triple.

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

$\boxed{\textit{SparseMatrix}\ \text{Multiply}(a, b)}$ ::=
if number of columns in *a* equals number of rows in *b* according to the formula: $d[i][j] = \Sigma(a[i][k] \cdot b[k][j])$ where $d(i, j)$ is the (i, j) element **else return** error.

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).

SparseMatrix Create(*maxRow*, *maxCol*) ::=

```
#define MAX_TERMS 101   /* maximum number of terms+1 */
typedef struct {
        int col;
        int row;
        int value;
        } term;
term a[MAX_TERMS];
```

The Sparse Matrix Abstra Data Type (contd.)

- Figure 2.5

Global information: row dimension, column dimension, and # of non-zero entries

$$\begin{array}{c|cccccc} & col0 & col1 & col2 & col3 & col4 & col5 \\ \hline row0 & 15 & 0 & 0 & 22 & 0 & -15 \\ row1 & 0 & 11 & 3 & 0 & 0 & 0 \\ row2 & 0 & 0 & 0 & -6 & 0 & 0 \\ row3 & 0 & 0 & 0 & 0 & 0 & 0 \\ row4 & 91 & 0 & 0 & 0 & 0 & 0 \\ row5 & 0 & 0 & 28 & 0 & 0 & 0 \end{array}$$

	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

(a)

	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

(b)

a[0]存放這個array的dimension 以及非0的部分有幾個

```
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++;
              }
    }
}
```

Global information setting

The index of the term to be set in b

Scanning all non zero terms

Fill in b[current] with a[j]

Proceed to the next vacant location in b

```c
void fastTranspose(term a[], term b[])
{  /* the transpose of a is placed in b */
   int rowTerms[MAX_COL], startingPos[MAX_COL];
                  numCols = a[0].col, numTerms = a[0].value;
                  = numCols;
                  = 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;
     }
   }
}
```

Starting position of each row of b

\# of terms per row of b

The initialization of rowTerms

The calculation of # of terms per row of b

Determination of startPos per row

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 $O(columns + elements)$.

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

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

	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

	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

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

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

```
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, 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;
            }
```

The starting index of the currently processed row of A

The index of the currently processed row of A

Step 1: Matrix transposing

$O(colsB + totalB)$

Step 2: Merging operation

```
            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:  /* go to next term in b */
                        j++;
            }
        }  /* end of for j <= totalB+1 */
        for (; a[i].row ==row; i++)
            ;
        rowBegin = i, row = a[i].row;
    } /* end of for i <= totalA */
    d[0].row = rowsA;
    d[0].col = colsB;
    d[0].value = totalD;
}
```

The Sparse Matrix Abstract Data Type -- Matrix Multiplication (contd.)

- The for-loop: $O(\sum_{row}(colsB \bullet termsRow + totalB))$

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

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

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

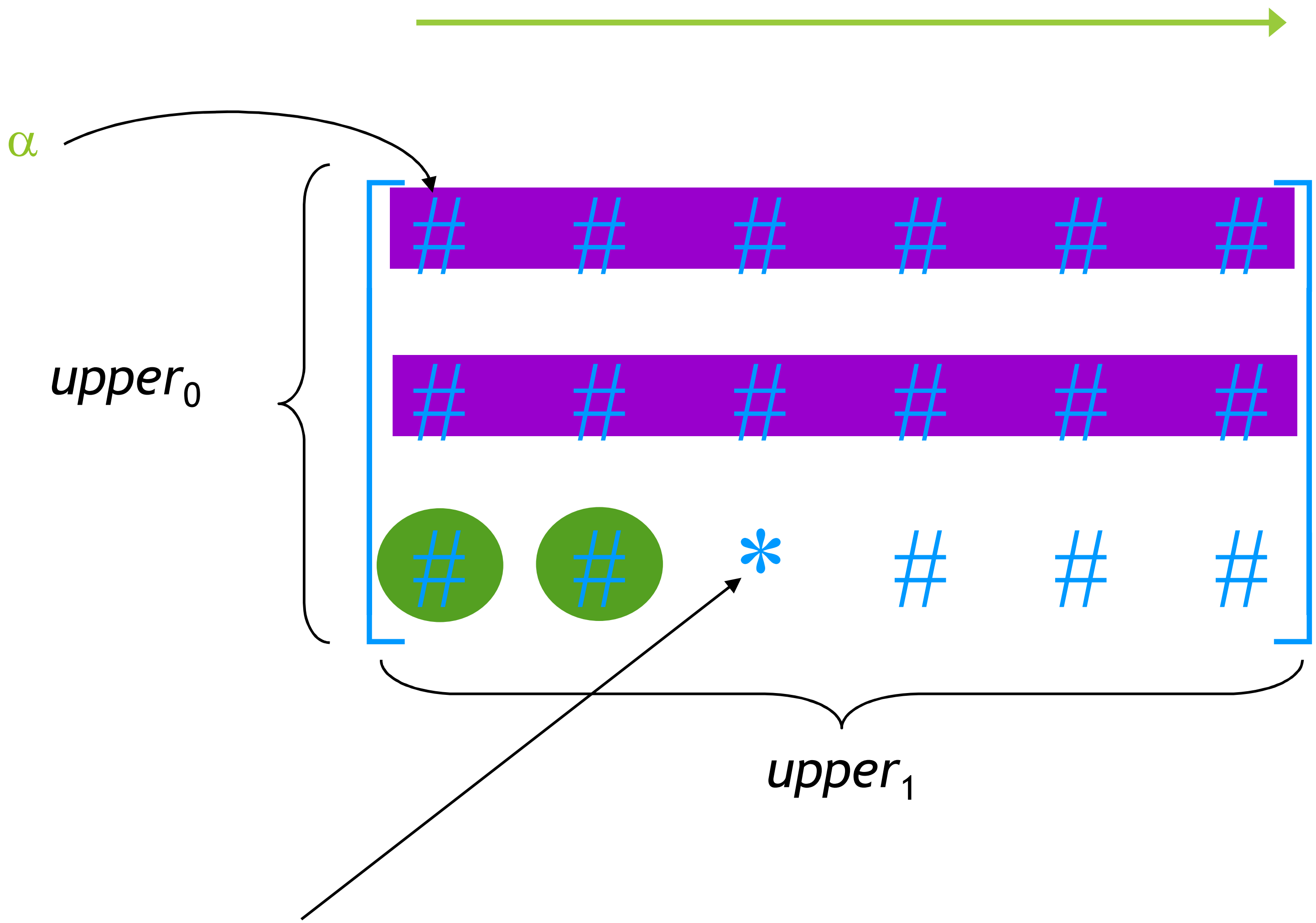
rowBegin

Representation of Multidimensional Arrays

- Two common ways
  - Row major order
    - Storing multidimensional arrays by rows
  - Column major order
- 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:

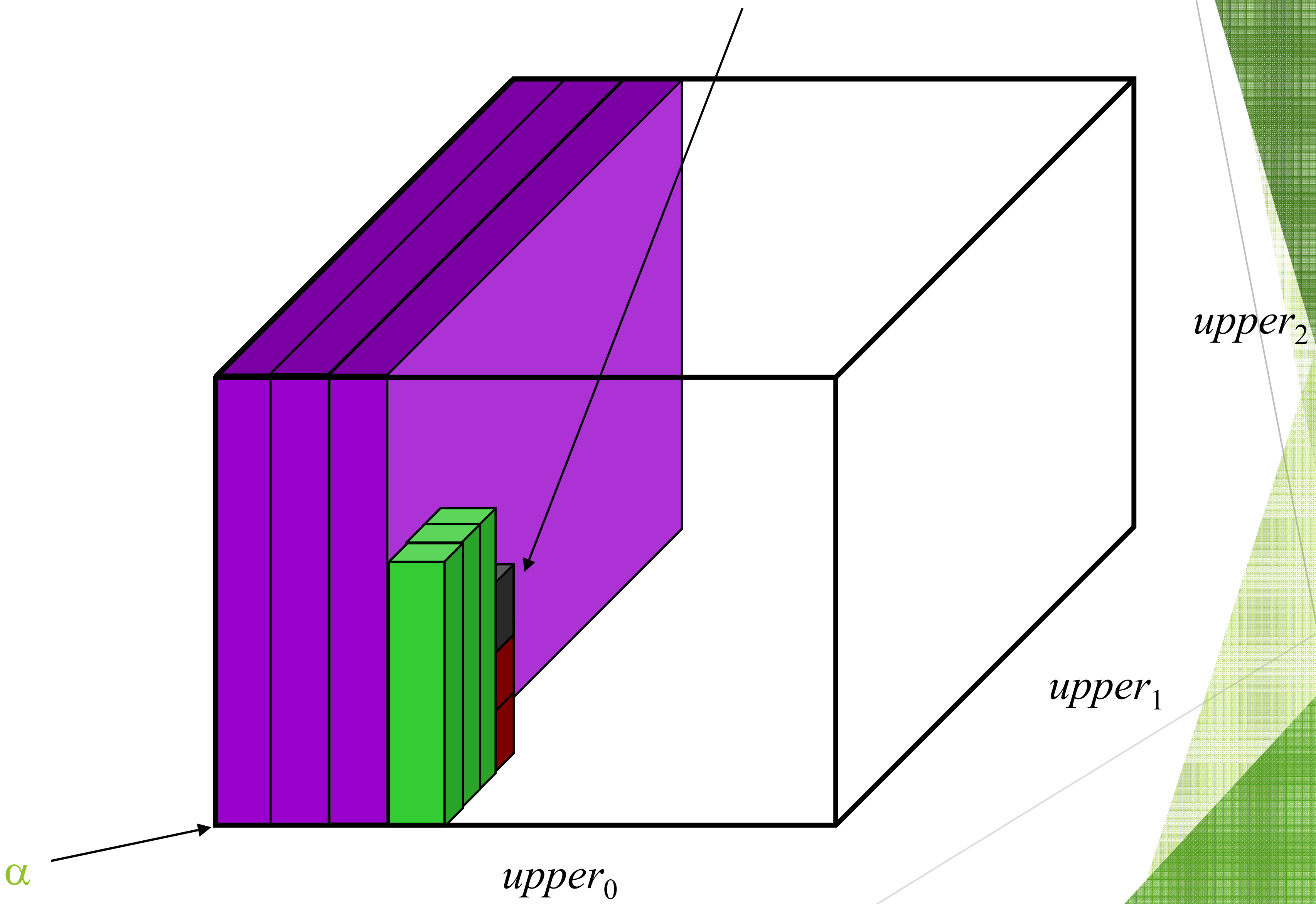
$$\alpha + \sum_{j=0}^{n-1} i_j a_j \text{ where } a_j = \begin{cases} \prod_{k=j+1}^{n-1} upper_k & 0 \le j < n-1 \\ 1 & j = n-1 \end{cases}$$

Row major



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

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



Representation of Multidimensional Arrays (contd.)

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