DATA STRUCTURES

UE17MC453

Dr. Veena S

sveena@pes.edu

Introduction

A **data structure** is a scheme for **organizing data** in the memory of a computer.

Some of the **more commonly used data structures include lists, arrays, stacks, queues, heaps, trees, and graphs.**

The way in which the data is organized affects the **performance** of a program for different tasks.

Computer programmers decide which data structures to use based on the nature of the data and the processes that need to be performed on that data.

Introduction

- Some examples
- Arrays - homogeneous data
- Stacks – LIFO (Pile of Trays)
- Queues – Ticket Counters
- Trees – Searching
- **Computer Science is all about studying Data (information) and its organization**

Information and its Meaning

- Information is data that is
  - Processed
  - Accurate and timely
  - Specific and organized for a purpose
  - Presented within a context that gives it meaning and relevance
  - Increases understandability and decreases uncertainty.

Information and its Meaning Contd..

- In computer science the basic unit of information is **bit**.
  - It only asserts one of two mutually exclusive possibilities. (a switch could be in on or off position) (1 or 0)
  - More switches gives us more exclusive possibilities.
  - N switches gives 2^n possibilities

Information and its Meaning Contd..

- Different ways of data representation
- Binary and Decimal **integers**
- Real Numbers (Mantissa and Exponent)
- Character strings
- Memory of computer is a group of bits
- Bits –Bytes – Words
- Bytes are stored in location (Address)
- Operations (add, move, store etc)

Data Types in C

- Basic Types
  – Int (short, int and long int), float, double, char
- Derived types
  – pointer, arrays, structures, unions
- Void type
  – Void
- Variable declaration in C specifies

  - amount of storage, how data is represented and the operations allowed on that data

Abstract Data Types

- A useful tool for specifying the logical properties of a data type is the ***Abstract Data Type***, or ***ADT.***
- Fundamentally, a data type is a collection of values and a set of operations on those values.
- The term "abstract data type" refers to basic mathematical concept that defines the data types.

Abstract Data Types Contd..

- In order to define an ADT we need to specify:
  - The components of an object of the ADT.
  - A set of procedures that provide the behavioral description of objects belonging to the ADT.
- The ADT consists of
  - **Value definition**
  - **Operator definition**

ADT for a Rational Number

- Rational number is a number that can be expressed as the quotient of two integers.
- The operations that can be defined are
  - Creation of a rational number from two integers
  - Addition
  - Multiplication
  - Testing for equality

Abstract Data Types Contd..

- Value definition
  - Defines collection of values for the ADT
    - Definition Clause
      - Example
        - » RATIONAL value consists of two integers
    - Condition Clause (optional)
      - Example
        - » The second integer should not be zero.
- Operator definition
  - Defines an operator as an abstract function with three parts
    - Header
    - Optional preconditions
    - Postconditions

ADT RATIONAL

```
/*value definition*/
abstract  typedef <integer, integer> RATIONAL;
Condition RATIONAL[1] != 0;

/*operator definition*/
abstract RATIONAL makerational (a,b)
int a,b;
precondition b!=0;
postcondition  makerational[0]==a;
               makerational[1]==b;
```

ADT RATIONAL Contd..

```
abstract  RATIONAL add(a,b)                /* written a+b */
RATIONAL a,b;
Postcondition  add[1]==a[1]*b[1];
                add[0]==a[0]*b[1] + b[0]*a[1];

abstract RATIONAL  mult(a,b)               /* written a*b */
RATIONAL a,b;
postcondition  mult[0] == a[0] * b[0];
                mult[1]==a[1] * b[1];

abstract equal(a.b)                        /*written a==b*/
RATIONAL a,b;
postcondition   equal == (a[0]*b[1] == b[0]*a[1]);
```

Structure of an ADT

- Example

  abstract RATIONAL mult(a, b)

  RATIONAL a,b;

  postcondition mult[0]==a[0]*b[0];

  mult[1]==a[1]*b[1];

Abstract defines that it is an ADT

mult[0] - numerator

mult[1] - denominator

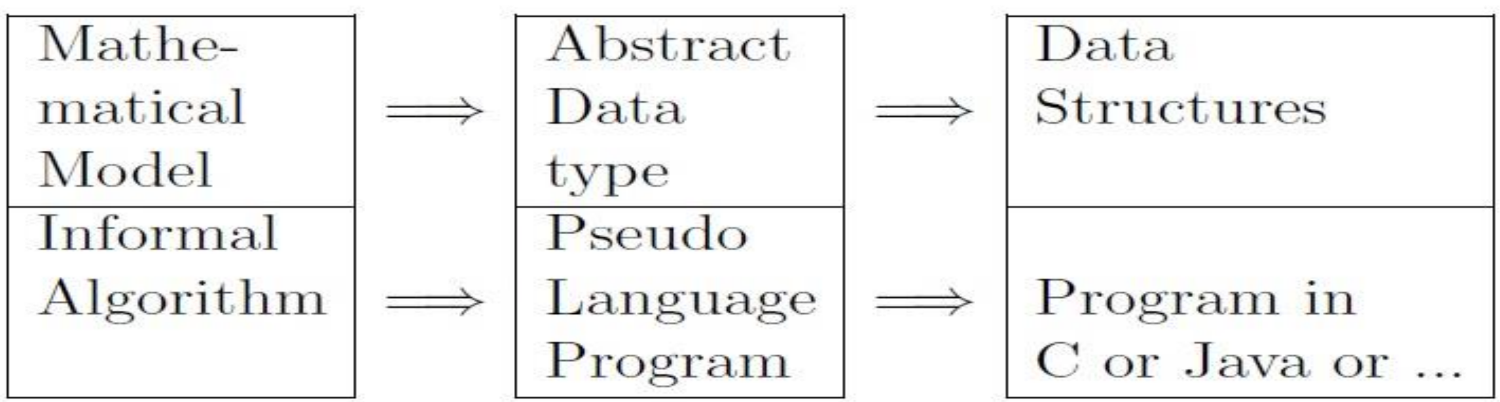
Specifies what the operation does. **The name of the function denotes the result of the operation**

Another way of writing ADT RATIONAL

```
/*value definition*/
abstract  typedef <int , int> RATIONAL;
condition RATIONAL[1] !=0;

/*operator definition*/
abstract equal(a,b);                 /* written  a == b */
RATIONAL a , b;
postcondition equal ==(a[0]*b[1] == b[0]*a[1]);

abstract  RATIONAL makerational(a,b)    /* written[a,b]  */
Int a,b;
precondition  b!=0;
postcondition  makerational[0]*b == a*makerational[1]

abstract  RATIONAL add(a,b)                  /*written a + b*/
RATIONAL a, b;
postcondition  add == [a[0] * b[1] + b[0] * a[1], a[1]*b[1]]

abstract  RATIONAL mult(a,b)                 /*written a * b*/
RATIONAL  a, b;
postcondition  mult == [a[0] * b[0], a[1] * b[1]]
```

ADT Contd..

- While defining the operations for ADT, we are **not specifying how they are to be computed.** It is determined by their **implementations.**
- Data Structure is an implementation of an ADT into statements of a programming language,



The Problem Solving Process in Computer Science

Sequences as Value Definitions

- Sequence is an ordered set of elements
  - $S=<s_0,s_1,s_2, \ldots.. S_{n-1}>$
- If S contains n elements, length of S is n.
- Operations
  - len(s) Gives the length of the sequence S
  - first(s) returns the value of the first element of S
  - last(s) returns the value of the last element of S
- A special sequence of length ***0*** called ***nilseq***.
  - It contains no elements.
  - ***first(nilseq)*** and ***last(nilseq)*** are undefined.

ADT Definition for sequence of elements

- The sequences are of arbitrary length and all the elements are of the same type ***tp***.

  abstract typedef <<tp>> seq1;

- The sequences are of fixed length and elements are of specific types.

  abstract typedef <tp0, tp1, tp2, …….,tpn> seq2;

- The sequences are of fixed length and elements are of the same type.

  abstract typedef <tp,n> seq3;

  - ***seq3*** represents a sequence of length ***n*** whose all elements are of the type ***tp***

ADT Definition for sequence of elements – Examples

abstract typedef <<int>> intseq; /*sequence of integer of any length*/

abstract typedef <integer, char, float> seq3; /*sequence of length 3 consisting of an integer, a character and a floating-point number*/

abstract typedef <<int,10>> intseq; /*sequence of 10 integers*/

abstract typedef <<,2>> pair; /*arbitrary sequence of length 2*/

Two sequences are equal, if each element of the first is equal to the corresponding element of the second.

ADT Definition for sequence of elements Contd..

- ***Subsequence***
  - It is the contiguous portion of a sequence.
- If ***S*** is the sequence, the function ***sub(S, i ,j)*** refers to the subsequence of S starting at position ***i*** in S and consisting of ***j*** consecutive elements.
- If ***T*** equals ***sub(S, i, k)*** and ***T*** is the sequence $<t_0,t_1,\ldots\ldots t_{k-1}>$ then $t_0 = s_i$, $t_1 = s_{i+1}$,…., $t_{k-1} = s_{i+j-1}$
- ***Concatenation: S + T*** It is the sequence of all elements of ***S*** followed by all the elements of ***T***.

ADT Definition for sequence of elements Contd..

Insertion:

- ***place(S, i, x)*** It is the sequence ***S*** with the element ***x*** inserted immediately following position ***i***.
- All subsequent elements are shifted by one position.
- **Deletion: Two ways**
  - **I**f *x* is an element of sequence *S*, *S* - <*x*> represents the sequence without all occurrences of element *x*.
    - Sequence *delete(S,i)* is equal to *S* with the element at position *i* deleted.
  - It can also be represented as

    sub(S, 0, i) + sub(S, i+1, len(S) – i -1)

ADT for varying-length character Strings

We have identified four basic operations that support strings:

Length – a function that returns the current length of the string.

Concat – a function that returns the concatenation of its two input strings.

Substr – a function returns a substring of a given string

Pos – a function that returns the first position of one string as a substring of another.

```
abstract typedef <<char>> STRING;

abstract length(s);
STRING s;
postcondition length == len(s);

abstract STRING  concat (s1,s2)
STRING s1,s2;
Postcondition concat == s1 + s2;

abstract STRING substr(s1,i,j)
STRING s1;
Int i ,j;
precondition  0 <= i < len(s1);
                          0 <= j <= len(s1) – i;
postcondition substr == sub(s1, i, j);

abstract  pos(s1,s2)
STRING s1,s2;
postcondition         /*lastpos = len(s1) – len(s2)*/
((pos == -1) && (for(i = 0;  i <= lastpos;  i++) (s2 <> sub(s1, i, len(s2)))))
||
(pos >= 0) && (pos <= laspos) && (s2 == sub(str1, pos, len(s2)) && (for(i = 1; i<pos; i++) (s2<>
sub(s1, i, len(s2)))));
```

- Post condition explanation

  /*lastpos = len(s1) – len(s2)*/

  - Defines symbol lastpos as representing the values of ***len(s1) – len(s2)*** for use within the postcondition
  - lastpos represents the maximum possible value of the result
    - The last position of s1 where a substring whose length equals that of s2 can start.

- The post condition states that one of the two conditions must be satisfied
  - The function’s value (pos) is -1 and s2 is not a substring of s1.
  - The function’s value is between 0 and lastpos, s2 is a substring of s1 beginning at the function values’s position and not in any earlier position.
  - Usage of pseudo for loop

```
for (i=x; i <=y; i++)
    (condition(i))
```

It is true if condition(i) is true for all i from x to y inclusive. It is also true if x > y.

Otherwise the entire for condition is false.

ARRAYS IN C

- The simplest form of array is a ***one-dimensional array*** that may be defined abstractly as a **finite ordered set of homogeneous elements.**
- By "finite" we mean that there is a specific number of elements in array.
- By "ordered" we mean that the elements of array are arranged so that there is a zeroth, first, second, third, and so.
- By "homogeneous" we mean that all the elements in the elements in the array must be of the same type.

- The two basic operations that access an array are ***extraction*** and ***storing***.
- The extraction operation is a function that access an array, a, and an index i, and returns an element of the array.
- In ***C***, the result of this operation is denoted by the expression a[i].
- The storing operation accepts an array, a, an index i, and an element x.
- In C this operation is denoted by the assignment statement a[i] = x.
- The smallest element of an array's index is called its ***lower bound*** and in ***c*** is always 0, and the highest element is called its ***upper bound.***
- The **range,** is given by ***upper - lower + 1.***

Array Declaration

- Using bound as a constant identifier

```
int a[100];
for (i = 0; i < 100; a[i ++] = 0);
```

- The equivalent alternative:

```
#define NUMELTS 100
int  a[NUMELTS];
for (i = 0; i < NUMELTS; a[i ++] = 0);
```

The Array as an ADT

```
abstract typedef <<eltype, ub>> ARRTYPE(ub, eltype); /* parameterized ADT */
Condition type(ub) == int;                  /* eltype is the type indicator not value*/

abstract eltype extract(a , i);                        /*written a[ i ]*/
ARRTYPE(ub, eltype) a;
int i;
precondition 0  <= i  <  ub;
postcondition extract  ==  a;

abstract store(a, i, elt)                              /*written a[ i ] = elt */
ARRTYPE (ub, eltype)a;
int i;
eltype elt;
precondition 0 <= i < ub;
postcondition a[ i ] == elt;
```

Using One-Dimensional Arrays

```
#define NUMELTS 100
int main(void)
{
 int num [NUMLETS];                    /*array of numbers*/
 int i;
 int total;                            /*sum of the numbers*/
 float avg;                            /*average of the numbers*/
 float diff;                           /*difference between each number and the average*/

 total = 0;
 for (i = 0; i < NUMELTS; i++)         /*read the numbers into the  array and add  them*/
 scanf ("%d", &num [i]);
 total += num [i];
}                                      /* end for */
avg = (float) total/NUMLETS;            /*compute the average*/
printf("\nnumber difference") ;        /*print heading*/ /*print each number and its difference*/
for (i = 0; i < NUMELTS; i ++)
{
 diff  = num [i] – avg;
 printf ("\n %d %f", num [i], diff);
}                                      /*end for */
 printf ("/n average is : %f", avg);
}                                      /*end main*/
```

Arrays as Parameters

```
Ffloat avg (float a [], int size) /*no range is specified for the array  a*/
{
   int i;
   float sum ;
  sum = 0;
  for (i = 0; i < size; i++)
  sum += a[i];
  return (sum / size);
}/*end avg*/
```

In the main program,

```
#define ARANGE 100
float a [ARANGE];
avg(a, ARANGE);
```

Note: Since array variable in C is a pointer the array parameters have to be passed by ***reference***.

The array's contents are not copied when it is passed as a parameter in C. Only the ***base address*** of the array is passed.

Passing by reference is more efficient in both time and space.

Character String Operations

```
#define STRSIZE 80
char string[STRSIZE];
```

The first function finds the current length of a string.

```
strlen (string)
char string [];
{
  int i;
  for (i = 0; string[i] != '\0'; i++)
 ;
 return (i);
}/*end strlen*/
```

```
int strpos(char s1[], char s2[])
{
  int len1, len2;
  int i, j1, j2;
  len1 = strlen(s1);
  len2 = strlen(s2);
  for (i = 0; i + len2 <= len1; i++)
        for (j1 = i; j2 = 0; j2 <= len2 && s1[j1] == s2[j2];
                                                        j1++, j2++)
                if ( j2 == len2 )
                 return (i);
return (-1);
}/*end strpos*/
```

```
void strcat (char s1[], char s2[])
{
  int i, j;
  for (i = 0; s1[i] != ‘\0’; i++)
  ;
  for (j = 0; s2[j] != ‘\0’; s1[i++] = s2[j ++])
 ;
}/*end strcat*/
```

```
void substr (char s1 [], int i, int j, char s2[])
{
  int k, m;
  for(k = i, m = 0; m <j; s2[m++] = s1[k++])
  ;
 s2[m] = '\0';
}/* end substr */
```

Two Dimensional Array

- It is an array which has both rows and columns.
- The number of rows or columns is called the ***range*** of the dimension.
- Methods of representing two-dimensional array
  - ***Row-major***
  - ***Column-major***

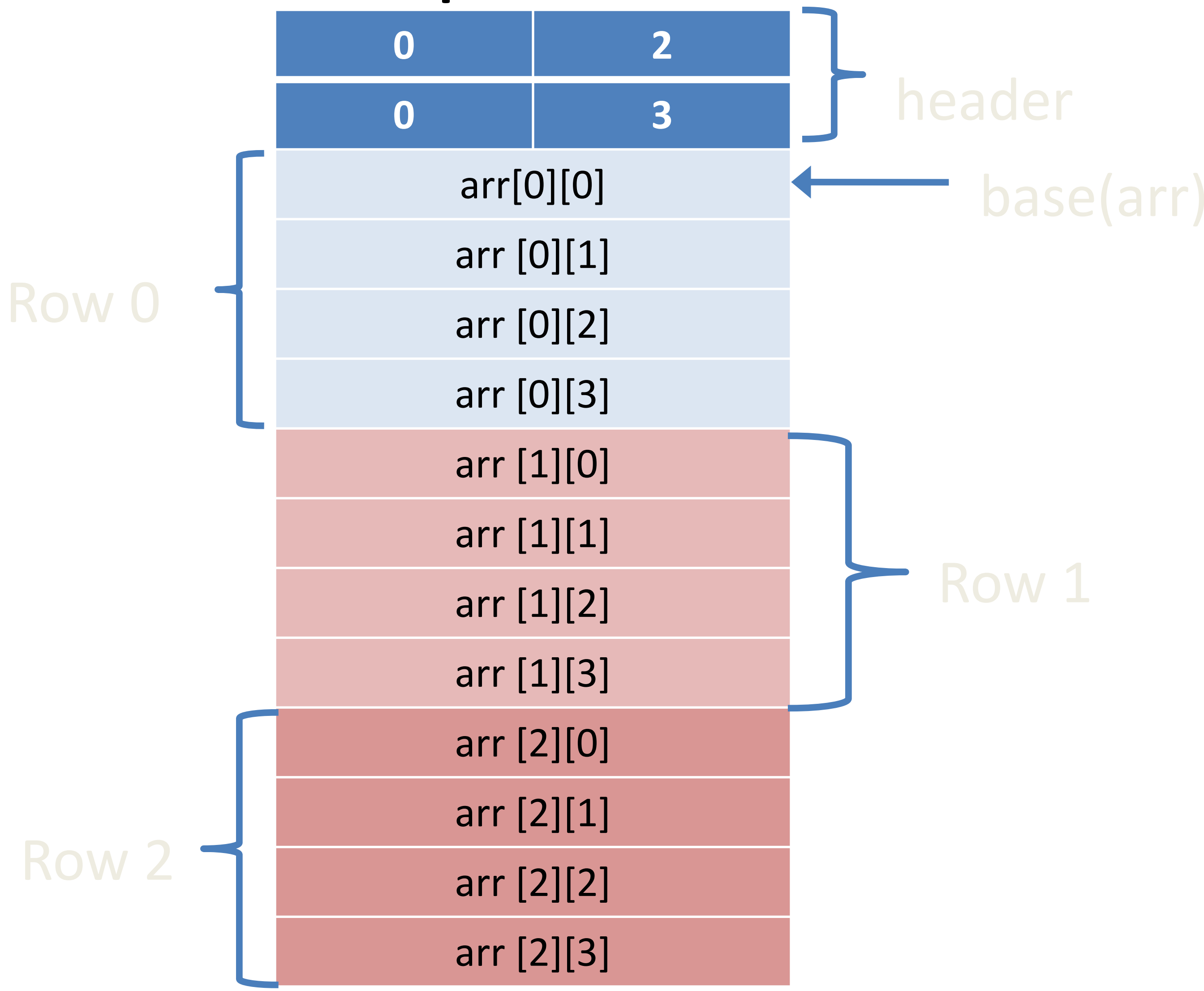
2-Dimensional Arrays – Row Major

- The first row of the array occupies the first set of memory locations reserved for the array.
- The second row occupies the next set of memory and so on.
- Header contain the upper and lower bounds of the two dimensions.

2-Dimensional Arrays – Row Major – Contd..

	Column 0	Column 1	Column 2	Column 3
Row 0				
Row 1				
Row 2				

2-Dimensional Arrays – Row Major – Representation



2-Dimensional Arrays – Row Major – Representation

- If we have declared array as

  int ar[r1][r2];

Then base(ar) is the address of the first element of the array, r1 and r2 are the ranges of first and second dimensions

Let esize is the size of the element in thw array (eg: int 4 bytes), then the address of ar[i1][i2] is calculated as **base(ar) + (i1 * r2 + i2) * esize**

i.e., base(ar) + offset * esize

2-Dimensional Arrays – Row Major – Offset Calculation

- Offset (Position) = (Row number * Number of Columns) + Column Number
- Example
  - The row and col number   : 4 3
  - Total number of rows and columns   : 5 5
  - Offset = 4 * 5 + 3 => 23

2-Dimensional Arrays – Address Calculation

- Address = offset * size of data type of array + base address
- Example
  - Base address of integer array : 1020
  - Offset calculated using row major : 23
  - Address = 23 * 4 + 1020 = 1112

2-Dimensional Arrays – Column Major

- All the elements of the first column are stored first, then the next column elements and so on.

2-Dimensional Arrays – Column Major – Contd..

- Offset (Position) = (Column number * Number of Rows) + Row Number
- Example
  - The row and col number : 4 3
  - Total number of rows and columns : 5 5
  - Offset = 3 * 5 + 4 => 19

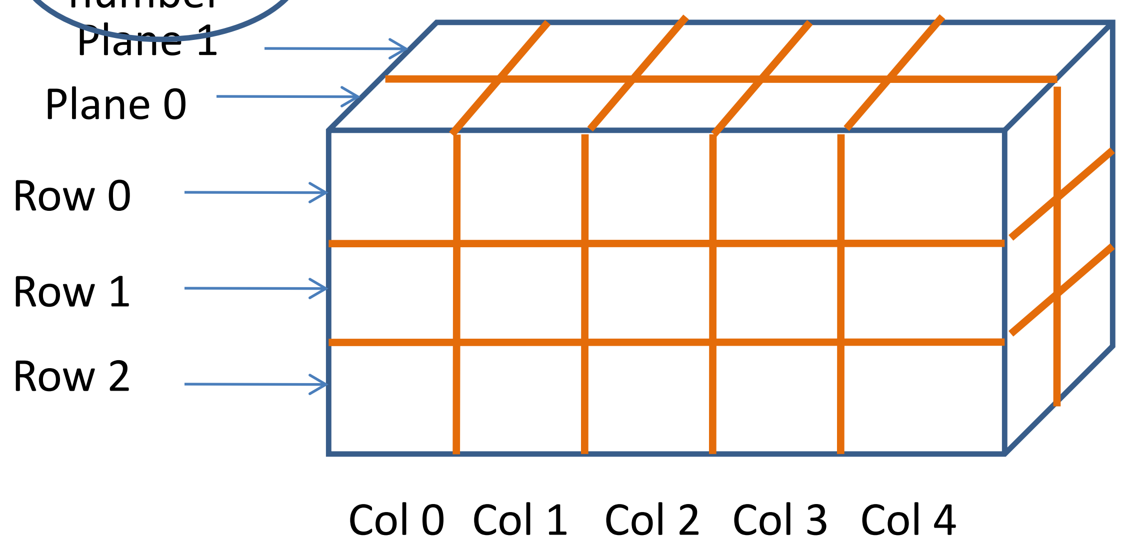
Multi-Dimensional Arrays

- C allows arrays with more than two dimensions.
- Example
  - int arr[2][3][5];

Row number

Column number

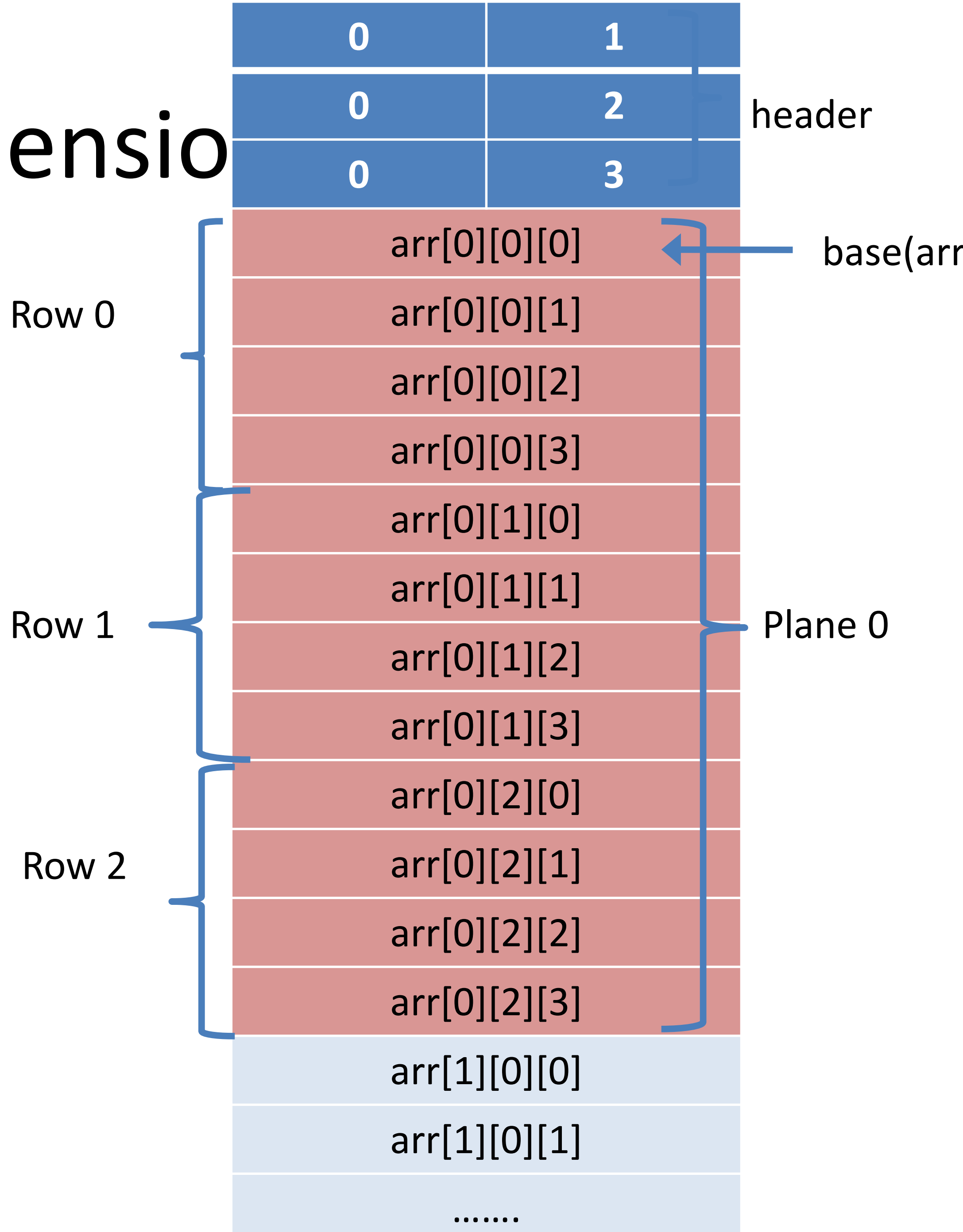
Plane number



Multi-Dimensio

- Inr arr[2][3][4]

This array contains
2 * 3 * 4
= 24 elements.



Multi-Dimensional Arrays – Contd..

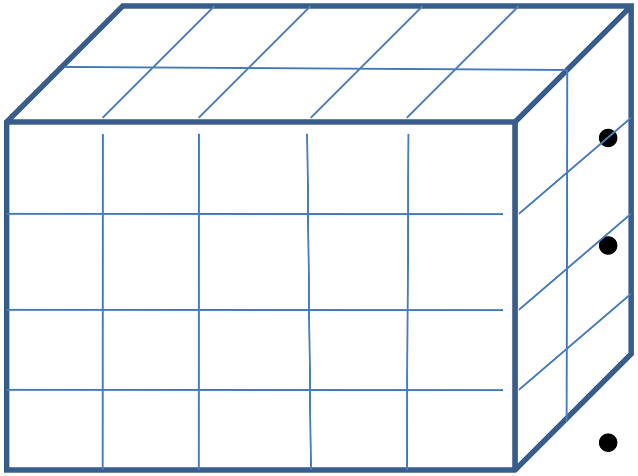
- C allows arbitrary number of dimensions.
- We can have a five–dimensional array.
  - int arr5 [7][6][5][4][7];
- Five subscripts are needed to reference any element of this array.
- This array contains 7*6*5*4*7 = 5880 elements.

Multi-Dimensional Arrays – Contd..

- If ***arr*** is an ***n***-dimensional array it is declared as
  - ***int arr[r1][r2]…..[rn];***
- Each element of ***arr*** is assumed to occupy ***esize*** of storage locations.
- ***base(arr)*** is defined as the address of the first element of the array.

Multi-Dimensional Arrays – Contd..

- Let us assume the address has to be calculated for a 3 dimensional array.



- *This array consists of 2 planes, 4 rows and 5 columns.*
- *It is an integer array.* **int a[2][4][5]**
- *The base address of the array is assumed to be 6700.*
- *Lets us calculate the address of a[1][2][3].*

Multi-Dimensional Arrays – Contd..

- a[1][2][3]
  - It is nothing but the element that is in plane 2 in row 3 and column 4.
  - Let us now start from the first element of the array and reach a[1][2][3].
  - The ***esize*** is 4 since it is an integer array.

Multi-Dimensional Arrays – Contd..

- Address of element at Plane 0 Row 0 Column 0 = 6700.
- To reach the second plane we have to pass thru all the elements of plane 0.
-

6700	6704	6708	6712	6716
6720	6724	6728	6732	6736
6740	6744	6748	6752	6756
6760	6764	6768	6772	6776

Multi-Dimensional Arrays – Contd..

- Address of last element of Plane 0 is 6776.
- The address of first element of plane 1 is 6780.
- The element is Plane 1 Row 0 Column 0.
- To reach that element 20 elements are passes through.
  - 1 * 4 * 5 = 20
    - It can be written as i1 * r2 * r3.
- Reach the row 3 in plane 1.

Multi-Dimensional Arrays – Contd..

- Address of element at Plane 1 Row 0 Column 0 = 6780.
- To reach the Row 2 we have to pass thru all the elements of row 0 and 1.

Multi-Dimensional Arrays – Contd..

•

6780	6784	6788	6792	6796
6800	6804	6808	6812	6816
6820	6824	6828	6832	

The address of a[1][2][3] is 6832

The address of element at plane 1 row 2 column 0 is 6820.

To reach the element in column 3 in the same row we have to pass through column 0, column 1, column2 elements.

Multi-Dimensional Arrays – Contd..

- To access the element ***arr[i_1][i_2][i_3]…….[i_n]***
  - Pass thru i_1 complete ***hyper-planes,*** each containing $r_2*r_3*…*r_n$ elements to reach the first element of ***arr*** whose first subscript is i_1.
  - *Next we have to pass thru* i_2 groups of $r_3*r_4*…*r_n$ elements to reach the first element of ***arr*** whose first subscript is i_1 and second subscript is i_2.
  - This continues until you reach the element desired.

Multi-Dimensional Arrays – Contd..

- The address of ***arr[i_1][i_2]…[i_n]*** is written as
  - ***base(arr) + esize * [$i_1 * r_2 * \ldots * r_n + i_2 * r_3 * \ldots * r_n + \ldots\ldots i_{(n-1)} * r_n + i_n$]***
  - ***base(arr) + esize * [$i_n + r_n * (i_{(n-1)} + r_{(n-1)*} (\ldots\ldots + r_3 * (i_2 + r_2 * i_1)))$]***

Multi-Dimensional Arrays – Contd..

- The algorithm to implement the address is

offset = 0;

for (j=0; j<n; j++)

    offset = r[j] * offest + i[j];

addr = base(arr) + esize * offset;