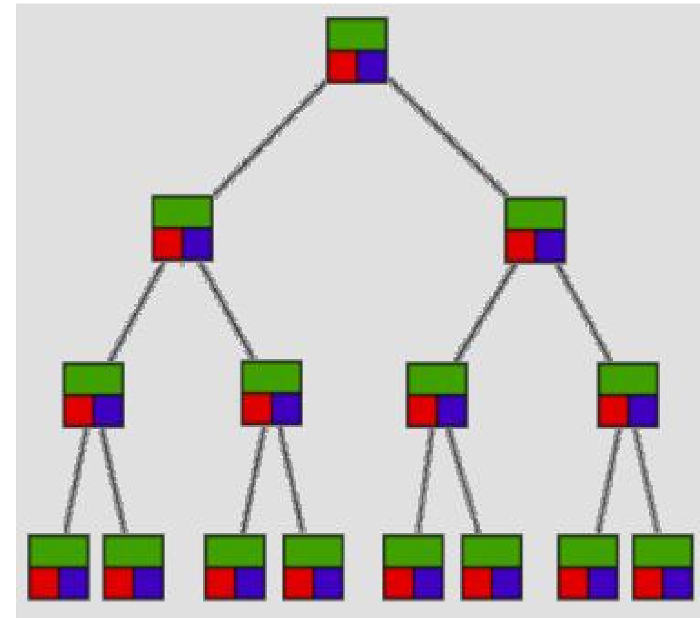
Data Structures

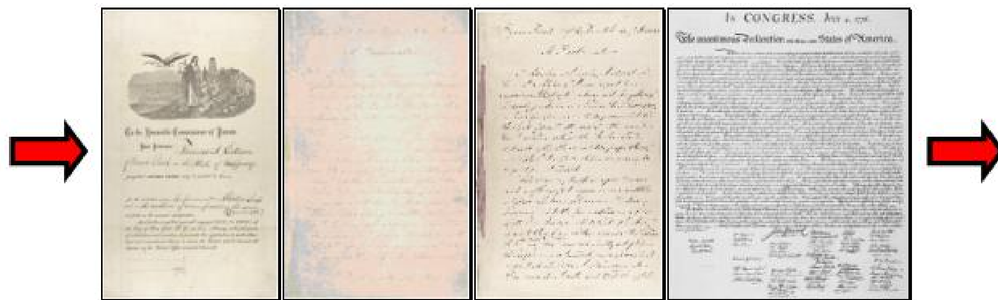
G S Lehal

Data Structures

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



Binary Tree



Queue

Data Structures

The way in which the data is organized affects the performance of a program for different tasks. The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.

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.

Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

Some Questions to Ask

- Are all data inserted into the data structure at the beginning, or are insertions interspersed with other operations?
- Can data be deleted?
- Are all data processed in some well-defined order, or is random access allowed?

Data Structure Philosophy

Each data structure has costs and benefits.

Rarely is one data structure better than another in all situations.

A data structure requires:

- space for each data item it stores,
- time to perform each basic operation,
- programming effort.

Data Structure Philosophy (cont)

Each problem has constraints on available space and time.

Only after a careful analysis of problem characteristics can we know the best data structure for the task.

Bank example:

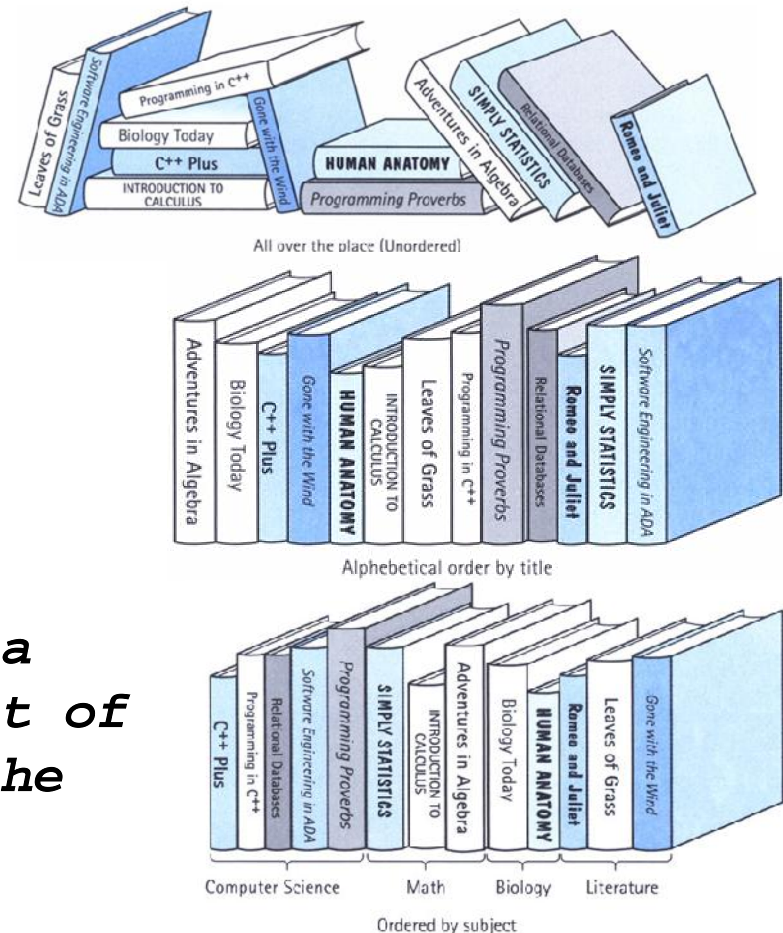
- Start account: a few minutes
- Transactions: a few seconds
- Close account: overnight

Common operations Data Structures

- Add
- Change
- Delete
- Traverse
- Search

Data Structures

- Example:library
  - is composed of elements (books)
  - Accessing a particular book requires knowledge of the arrangement of the books
  - Users access books only through the librarian



The logical arrangement of data elements, combined with the set of operations we need to access the elements.

Common Data Structures

- Array
- Stack
- Queue
- Linked List
- Tree
- Heap
- Hash Table
- Priority Queue

Arrays

- Groups of nearly identical variables are very common and very tedious to program.
  - Example: make one variable to hold the total marks of each student in the class. We would have to declare 22 integer variables that all represent something similar (total marks) and all have similar names (e.g. marks1, marks2, …, marks22):

```
int marks1, marks2, marks3, ...
```

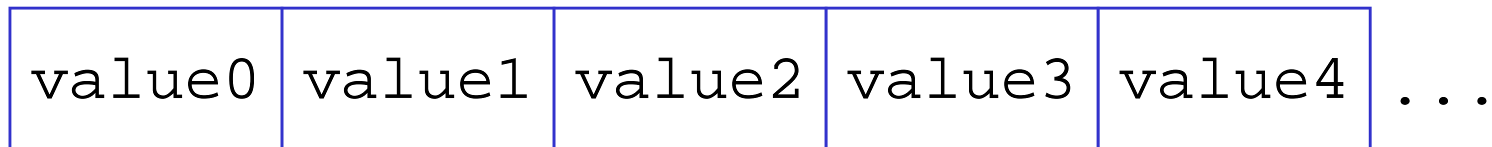
- Arrays allow us to "group variables together" under a common name.

```
int marks[22];   /* holds 22 integers */
```

Arrays

- Variable = a named memory location for just one stored value
- Array = named memory location for many values, stored sequentially.

- Think of an array as a sequence of boxes, each one holding a value:



Arrays

- Each box holds one value
- All boxes hold values of the same type
- The entire set of boxes is given one name
- We can select any box by using the common name and the index of the box
- The indices start at zero.

marks

223	256	312	299	329	. . .
0	1	2	3	4	

The Array data structure

- An array is an indexed sequence of components stored in contiguous memory locations.
  - The length of the array is determined when the array is created, and cannot be changed
  - Each component of the array has a fixed, unique index
    - Indices range from a lower bound to an upper bound
  - Any component of the array can be inspected or updated by using its index
    - This is an efficient operation: O(1) = constant time

Array variations

- The array indices may be integers (C, Java) or other discrete data types (Pascal, Ada)
- The lower bound may be zero (C, Java), one (Fortran), or chosen by the programmer (Pascal, Ada)
- In most languages, arrays are homogeneous (all components must have the same type); in some (Lisp, Prolog) the components may be heterogeneous (of varying types)

Arrays in C/C++

- Array indices are integers
- An array of length n has bounds 0 and n-1
- Arrays are homogeneous

Arrays

- To create an array, we must first declare it (just as we need to declare variables).
- In the declaration we need to specify the name of the array, the type of the elements and the number of elements:

```
int marks[5];
```

- After the declaration:

(All box contents are initially undefined)

Arrays

- Array declaration syntax:

```
element_type array_name[size];
```

- The size must be a constant expression.
  - NO variables allowed
  - You cannot change the size at runtime.
- The size is often set by a #define directive.

Arrays

- Array indexing:
  - Each element of the array is like an ordinary variable.
  - In order to access that element, you must specify the name of the array and the index of the element:

```
array_name[index]
```

- Example:

```
marks[0] = 312;
```

This assigns the value 312 to the first element of the array marks.

Arrays

- Array indexing:
  - Element indices start at 0: **`array[0]`** is the first element
  - The index of last element is n-1,where n is the size of the array. A common mistake is to access a[20], when the array contains only 20 elements.
  - CAREFUL! The compiler will not catch a wrong index.
  - For example, it will allow you to write/read in `grades[-10]` or `grades[4325]`.
  - This may corrupt your program or your data.

Arrays

- After you declare an array, you must assign values (initialize it) before you can use it in any other way. There are several ways to do this.

1. Initialize the array during declaration:

```
int marks[5] = {219, 320, 199, 256, 311};
```

- After the initialization:

marks

219	320	199	256	311

Array initialization

- You can initialize some of the elements with non zero values, while the rest will automatically be initialized to zero.

```
int marks[5] = {219, 320};
```

Equivalent to:

```
marks[0] = 219; marks[1]=320;
marks[2] = marks[3] = marks[4] = 0;
```

- You can let the compiler figure the size for you:

```
int marks[] = {219, 320, 199, 256, 311};
```

This will set the size of array **`marks`** equal to the number of entries in the RHS bracket

Array initialization

2. Use assignment statements to initialize each element separately:

```
int marks[5];
marks[0] = 219;
marks[1] = 320;
marks[2] = 199;
marks[3] = 256;
marks[4] = 311;
```

Index
Selects box #,
or 'element'

The index of the first element is zero.
The index of the last element is size - 1

Array initialization

3. Use a for-loop to read the values from the user (or assign some default value).

```
int marks[5];
for (i=0; i<5; i++) {
      printf("Type total marks : ");
      scanf("%d", &marks[i]);
}
```

Array initialization

4. For arrays of integers only:
   You can initialize all elements to zero using the following syntax:

```
int scores[5] = {0};
```

This is equivalent to:

```
int scores[5] = {0, 0, 0, 0, 0};
```

It only works for 0, no other initial value!

Array pitfalls

- You CANNOT assign a value to the entire array as if it were one big variable:

```
int scores[5];

scores = 18;
```

ERROR!

- You CANNOT change the size of an array:

```
int size;
int scores[size];
printf("Type array size: ");
scanf("%d", &size);
```

ERROR!

Array pitfalls

- You CANNOT rely on C to enforce valid index values

```
int scores[5];

scores[-1] = 18;
```

CAREFUL!

- You CANNOT assign one array name to another:

```
int scores[30], grades[30];
scores = grades
```

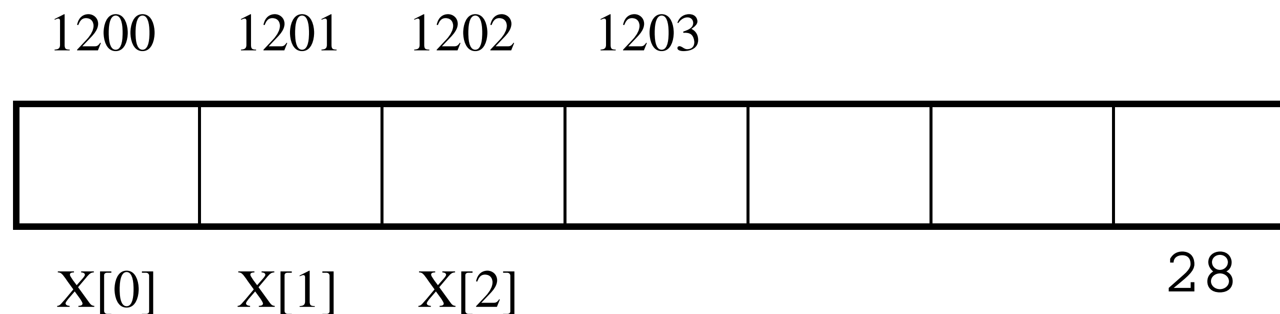
ERROR!

  - If you need to do something like this, use a for-loop to assign the elements one at a time.

Representation of Array in a Memory

- The process to determine the address in a memory:
  a) First address – base address.
  b) Relative address to base address through index function.

  Example: char X[100];
  Let *char* uses 1 location storage.
  If the base address is 1200 then the next element is in 1201.
  Index Function is written as:
  Loc (X[i]) = Loc(X[0]) + i , *i* is subscript and LB = 0



Representation of Array in a Memory

- In general, index function:
  Loc (X[i]) = Loc(X[0]) + w*(i);

  where w is length of memory location required.
  For real number: 4 byte, integer: 2 byte and character: 1 byte.

- <u>Example:</u>
  If Loc(X[0]) = 1200, and w = 4, find Loc(X[8]) ?
  Loc(X[8])= Loc(X[0]) + 4*(8)
  = 1232

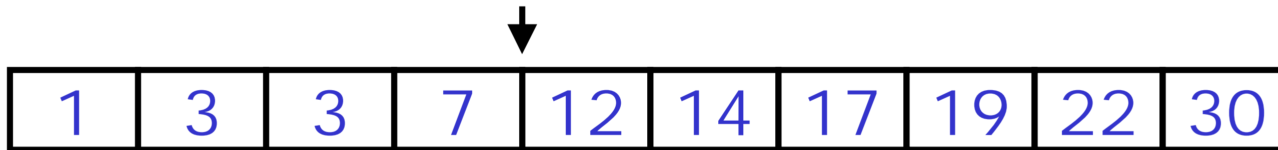
Insertion into Array

What happens if you want to insert an item at a specified position in an existing array?

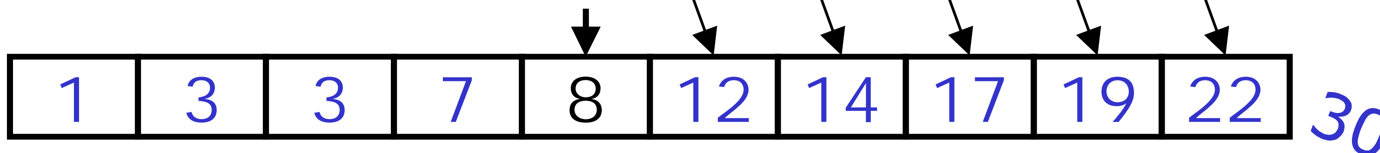
- Write over the current contents at the given index (which might not be appropriate), or
- The item originally at the given index must be moved up one position, and all the items after that index must shuffled up

Inserting an element into an array

- Suppose we want to insert the value 8 into this sorted array (while keeping the array sorted)



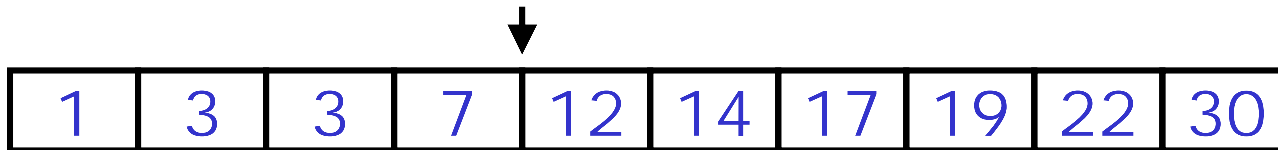
- We can do this by shifting all the elements after the mark right by one location
  - Of course, we have to discard the 30 when we do this



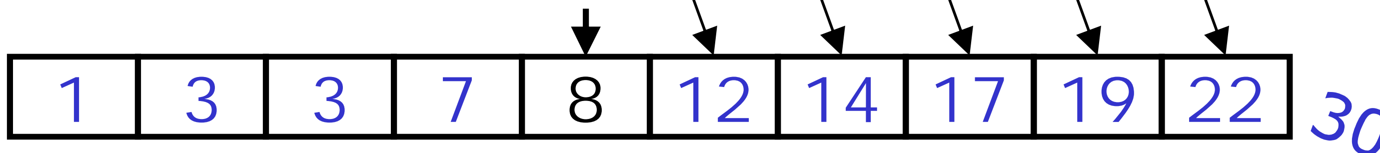
- Moving all those elements makes this a *slow* operation (linear in the size of the array)

Inserting an element into an array

- Suppose we want to insert the value 8 into this sorted array (while keeping the array sorted)



- We can do this by shifting all the elements after the mark right by one location
  - Of course, we have to discard the 30 when we do this



- Moving all those elements makes this a *slow* operation (linear in the size of the array)

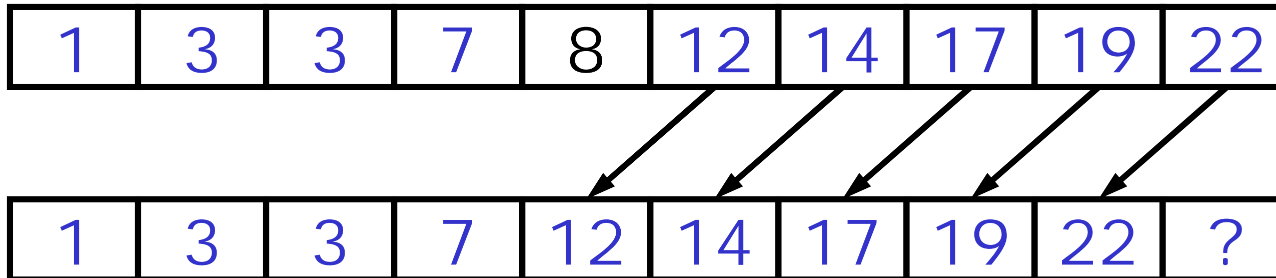
Removal from Arrays

What happens if you want to remove an item from a specified position in an existing array?

- Leave gaps in the array, i.e. indices that contain no elements, which in practice, means that the array element has to be given a special value to indicate that it is "empty", or
- All the items after the (removed item's) index must be shuffled down

Deleting an element from an array

- Deleting an element is similar--again, we have to move all the elements after it



- Deletion is a slow operation; we don't want to do it very often
- Deletion leaves a "vacant" location at the end
  - How do we mark it vacant?
    - Every bit pattern represents a valid integer
    - We must keep a count of how many valid elements are in the array

Passing Arrays as Parameters in Functions

- Arrays are always passed by reference.
- The “[ ]” in the formal parameter specification indicates that the variable is an array.
- It is a good practice to pass the dimension of the array as another parameter, as the compiler does not know the size of the array.

Smallest Value

- Problem
  - Find the smallest value in an array of integers
- Input
  - An array of integers and a value indicating the number of integers
- Output
  - Smallest value in the array
- Note
  - Array remains unchanged after finding the smallest value!

Passing An Array

Notice empty brackets

```
int ListMinimum(int Ar[], int asize) {
   int SmallestValueSoFar = Ar[0];
   for (int i = 1; i < asize; ++i) {
      if (Ar[i] < SmallestValueSoFar ) {
         SmallestValueSoFar = Ar[i];
      }
   }
   return SmallestValueSoFar ;
}
```

Could we just assign a 0 and have it work?

Reverse function

```
void reverse(int list[], int newList[], int size)
{
   for (int i = 0, j = size - 1; i < size; i++, j--)
  {
     newList[j] = list[i];
   }
}
int main(){
int list1[5] = {1, 2 , 4, 5, 6};
int list2[5];
reverse(list1, list2 ,5);
return 0;
}
```

list	1	2	3	4	5	6
newList	6	5	4	3	2	1

A few exercises on arrays

- Write down the function
  - int del_array(int a[], int size, int pos) to delete the value stored at position pos in the array a.
- Write down the function
  - void insert(int a[], int size, int pos, int val) to insert val at position pos in the array a.
- Write down the function
  - void reverse(int a[], int size) to reverse the contents of array a.

2D Arrays

- Arrays are not limited to type int; we can have arrays of… any other type including float, char , double and even arrays !!!
- An array of arrays is called a two-dimensional array and can be regarded as a table with a number of rows and columns:

2D Arrays

- Each row corresponds to marks of a student. Each column corresponds to marks in a subject.

is a 5 Î 4 array:
5 rows, 4 columns

<table>
<tr><td>50</td><td>50</td><td>50</td><td>60</td></tr>
<tr><td>80</td><td>80</td><td>88</td><td>72</td></tr>
<tr><td>49</td><td>50</td><td>40</td><td>60</td></tr>
<tr><td>60</td><td>60</td><td>60</td><td>76</td></tr>
<tr><td>75</td><td>80</td><td>75</td><td>81</td></tr>
</table>

Declaring 2D Arrays

50	50	50	60
80	80	88	72
49	50	40	60
60	60	60	76
75	80	75	81

- This is
  an array of size 5 marks[5]
  whose elements are arrays of size 4 [4]
  whose elements are integer int
- Declare it like this: **`int marks[5][4];`**

```
int marks[5][4];
```

type of element in each slot → `int`; name of array → `marks`; number of rows → `[5]`; number of columns → `[4]`

Initializing 2D arrays

- An array may be initialized at the time of declaration:

```
int marks[5][4] = {
                   { 50, 50, 50, 60} ,
                   { 80, 80, 88, 72} ,
                   { 49, 50, 40, 60} ,
                   { 60, 60, 60, 76} ,
                   { 75, 80, 75, 81}
                  };
```

Initializing 2D arrays

- An array may be initialized by using a loop

```
#define ROWS 5
#define COLS 4

int main () {
        int i, j;
        int marks[5][4];

        /* initialize all elements to zeroes: */
        for (i=0; i<ROWS; i++)
                for (j=0; j<COLS; j++)
                        marks[i][j] = 0;
}
```

Initializing 2D arrays

- An array may be initialized by the user

```
#define ROWS 5
#define COLS 4

int main () {
      int i, j;
      int marks[5][4];
      for (i=0; i<ROWS; i++)
            for (j=0; j<COLS; j++)
                  scanf("%d", &marks[i][j]);
}
```

Initializing 2D arrays

- An integer array may be initialized to all zeros as follows:

```
int nums[5][4] = {0};
```

- This only works for zero.

Using 2D arrays

- To access an element of a 2D array, you need to specify both the row and the column:

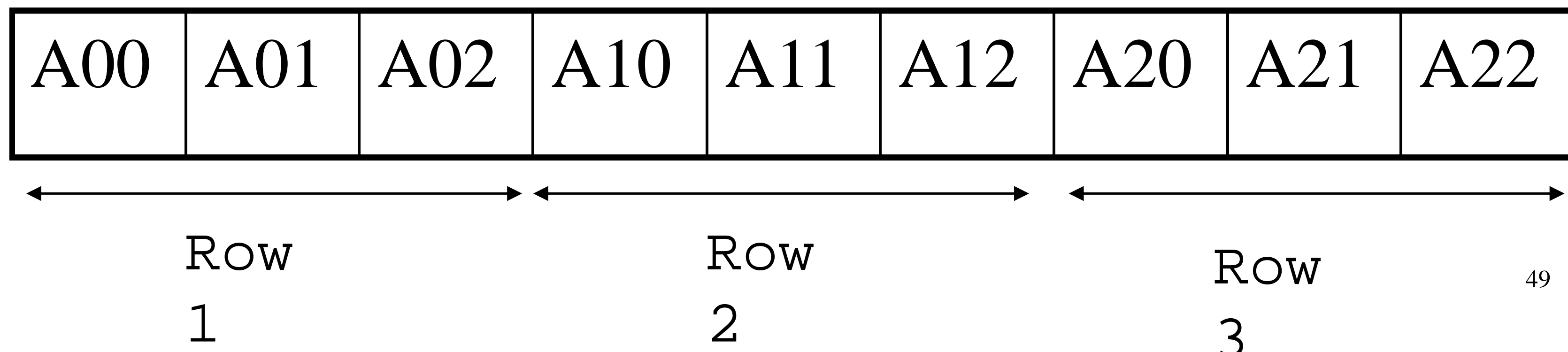
```
nums[0][0] = 16;

printf("%d", nums[1][2]);
```

Storing Two-Dimensional Array in memory

- A two-dimensional array can be stored in the memory in two ways:
  - Row-major implementation
  - Column-major implementation

Row-major Implementation

- Row-major implementation is a linearization technique in which elements of array are stored row-wise that means the complete first row is stored then the complete second row is stored and so on.
- For example an array [3][3] is stored in the memory as show bellow:



Row-major Implementation

- The storage can be clearly understood by arranging array as matrix as show bellow:

a=

A00	A01	A02	Row 1
A10	A11	A12	Row 2
A20	A21	A22	Row 3

Row-major Implementation

- **Address of elements in row major implementation:**
- the computer does not keep the track of all elements of the array, rather it keeps a base address and calculates the address of required element when needed.
- It calculates by the following relation:

  address of element $a[i][j]=B+W(n(i)+(j))$

  - Where, B=Base address
  - W=size of each element array element
  - n=the number of columns

Row-major Implementation

- **A two-dimensional array defined as a[4][5] requires 4 bytes of storage space for each element. If the array is stored in row-major form , then calculate the address of element at location a[2][3] given base address is 100.**
- **Sol:** B=100, i=2, j=3, n=5, W=4(element size)
- Address of a[i][j]= B+W(n*(i)+(j))
- Address of a[2][3]=100+4(5*2 + (3))

  =100+4(13)

  =100+52

  =152

Column-major Implementation

- In column major implementation memory allocation is done column by column that means first the elements of the complete first column is stored then elements of complete second column is stored and so on.
- For example an array [3][3] is stored in the memory as show bellow:

A00	A10	A20	A01	A11	A21	A02	A12	A22
Col 1			Col 2			Col 3		

Column-major Implementation

- The storage can be clearly understood by arranging array as matrix as show bellow:

a=

A00	A01	A02
A10	A11	A12
A20	A21	A22
Col 1	Col 2	Col 3

Column-major Implementation

- **Address of elements in column major implementation:**
- the computer does not keep the track of all elements of the array, rather it keeps a base address and calculates the address of required element when needed.
- It calculates by the following relation:

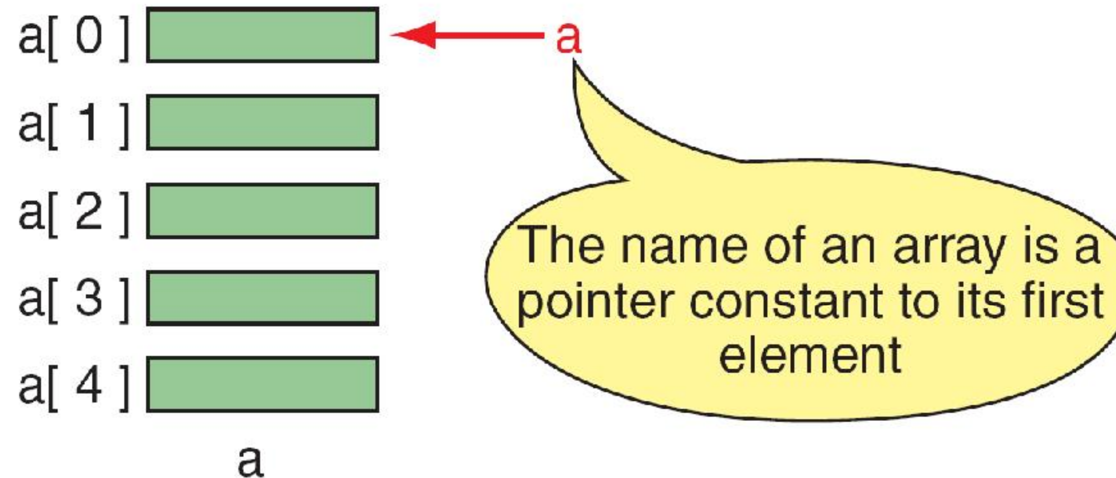
  address of element a[i][j]=B+W(i+m*(j))

  - Where, B=Base address
  - W=size of each element array element
  - m=the number of rows

Arrays and Pointers

- In C, there is a strong relationship between pointers and arrays. The name of an array is a pointer to the first element. However, the array name is not a pointer *variable*. It is a ***constant pointer*** that always points to the first element of the array and its value can not be changed.
- Also when an array is declared, space for all elements in the array is allocated, however when a pointer variable is declared only space sufficient to hold an address is allocated.
  - The declaration int a[10]; defines an array of 10 integers and reserves space for 10 integers.
  - The declaration int *p; defines p as a "pointer to an int" and reserves space for 1 pointer variable.

Arrays and Pointers



`a` is a pointer only to the first element—not the whole array

Arrays and Pointers

- If a is the name of an array, the expression

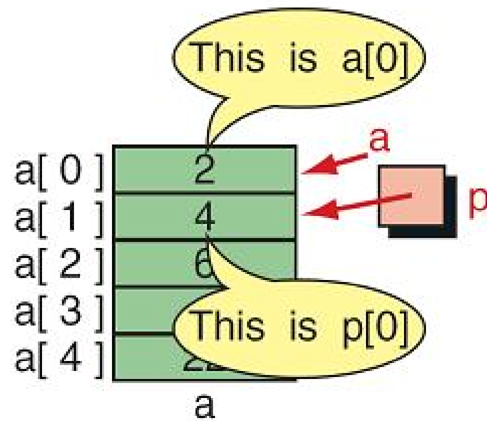
  a is equivalent to &a[0].

  a[0] is equivalent to *a.

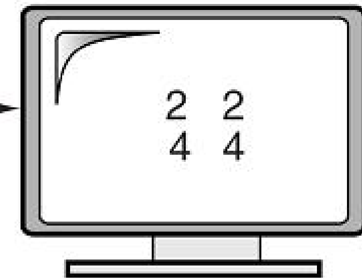
  a[i] is equivalent to *(a + i).

- It follows then that &a[i] and (a + i) are also equivalent. Both represent the address of the ith element beyond a.
- On the other hand, if p is a pointer, then it may be used with a subscript as if it were the name of an array. p[i] is identical to *(p + i)

Arrays and Pointers




```c
#include <stdio.h>
int main (void)
{
  int  a[5] = {2, 4, 6, 8, 22};
  int* p;
  …
  p = &a[1];
  printf("%d %d", a[0], p[-1]);
  printf("\n");
  printf("%d %d", a[1], p[0]);
  …
} // main
```



To access an array, any pointer to the first element can be used instead of the name of the array.

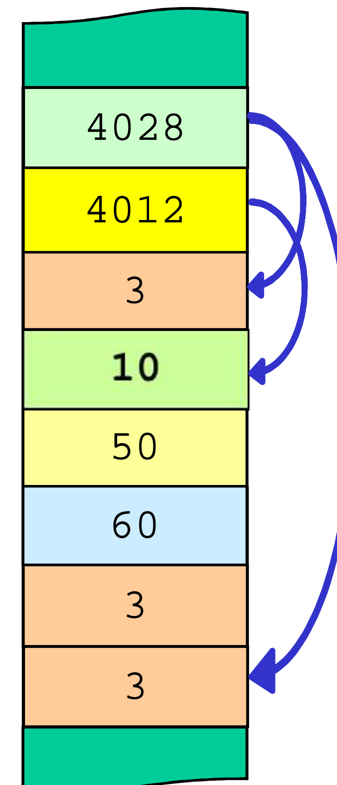
Array Example Using a Pointer

```
int x[4] = {12, 20, 39, 43}, *y;
y = &x[0];                      // y points to the beginning of the array
printf("%d\n", x[0]);           // outputs 12
printf("%d\n", *y);                     // also outputs 12
printf("%d\n", *y+1);           // outputs 13 (12 + 1)
printf("%d\n", (*y)+1);         // also outputs 13
printf("%d\n", *(y+1));         // outputs x[1] or 20
y+=2;                           // y now points to x[2]
printf("%d\n", *y);                     // prints out 39
*y = 38;                        // changes x[2] to 38
printf("%d\n", *y-1);           // prints out x[2] - 1 or 37
*y++;                           // sets y to point at the next array element
printf("%d\n", *y);                     // outputs x[3] (43)
(*y)++;                         // sets what y points to to be 1 greater
printf("%d\n", *y);                     // outputs the new value of x[3] (44)
```

Arrays and Pointers

What will be the value of elements of array a ?

```
main()
{
    int *p, *p2, x=50;
    int a[5]={1,2,3,4,5};
    p=&x;
    p2=a;
   *(p2+1)=*p;
   *p= *p2 + *(p2+2);
   *p2 = 10;
   p=p2+4;
   p[-2] = 60;
   p2[3] = x;
   p2[4] = p[-1];
}
```



Arrays and Pointers

- Consider the declaration:
  - int b[10], *bPtr=b;
- Element b[n] same as
  - *(b+n)
  - *(bPtr + n)
  - bPtr[n]
  - n[b] (Why?)
- b[n] = *(b+n) = *(n+b) = n[b]

Arrays and Pointers

```
int  i;
int  array[10];

for (i = 0; i < 10; i++)
{
  array[i] = 0;
}
```

```
int *p;
int  array[10];

for (p = array; p < &array[10]; p++)
{
  *p = 0;
}
```

These two blocks of code are functionally equivalent

Arrays and Pointers

What will be the output?

```
void main()
{
        int b[10], *bPtr = b, i;
        for(i=0; i<10; i++)b[i]=i;
        printf("%d\n", *(bPtr+5));          5
        printf("%d\n", bPtr[5]);            5
        printf("%d\n", 5[bPtr]);            5
        printf("%d\n", 5[b]);               5
}
```

Arrays and Pointers

What will be the value of elements of array a ?

```
main()
{
        int *p, *p2, x=50, i=2;
        int a[5]={1,2,3,4,5};
        p=&x;
        p2=a;
        *(p2+1)=*p;
        *p= *p2 + *(p2+2);
        *p2 = 10;
        p=p2+4;
        p[-2] = 60;
        p2[3] = *p2*i;
        p2[4] = p[-1] + *(p-1);
}
```

Arrays and Pointers

What will be the value of elements of array a ?

```
main()
{
        int *p, *p2, x=50, i=2;
        int a[5]={1,2,3,4,5};
        p=&x;
        p2=a;
        *(p2+1)=*p;
        *p= *p2 + *(p2+2);
        *p2 = 10;
        p=p2+4;
        p[-2] = 60;
        p2[3] = *p2*i;
        p2[4] = p[-1] + *(p-1);
} 10, 50, 60, 20, 40
```

Arrays and Pointers

What will be the value of elements of array x ?

```
main()
{
        int i, x[10],  *p, *q;
        for(i=0; i<10; i++)
                x[i]=i+1;
        p = x+9;
        q = p-8;
        for(i=3; i>0; i--, p--)
                *p += 10;
        for(i=1; i<5; i++, q++)
                *q += i;
}
```

Arrays and Pointers

What will be the value of elements of array x ?

```
main()
{
        int i, n=8, x[10],  *p, *q, j=10;
        for(i=0; i<10; i++)
                x[i]=i+1;
        p = x+9;
        q = p-8;
        for(i=3; i>0; i--, p--)
                *p += 10;
        for(i=1; i<5; i++, q++)
                *q += i;
} 1, 3, 5, 7, 9, 6, 7, 18, 19, 20
```

Passing Two-Dimensional Arrays to Functions

You can pass a two-dimensional array to a function; however, C++ requires that the column size to be specified in the function declaration. In the next example, we have a function that sums up two `two-dimensional array` into a third one.

Passing Two-Dimensional Arrays to Functions

```
// Sum up two 2-dimensional arrays into a third one
#include <iostream>
using namespace std;
const int max_cols = 5;
// c[i][j] = a[i][j] + b[i][j]
void  add_array(double a[][max_cols],
                double b[][max_cols],
                double c[][max_cols],
                int rows)
{     int i, j;
      for (i=0; i < rows; i++)
         for (j=0; j < max_cols; j++)
              c[i][j] = a[i][j] + b[i][j];
}
```

Summary of Arrays

- Good things:
  - Fast, random access of elements
  - Very memory efficient, very little memory is required other than that needed to store the contents (but see bellow)
- Bad things:
  - Slow deletion and insertion of elements
  - Size must be known when the array is created and is fixed (static)