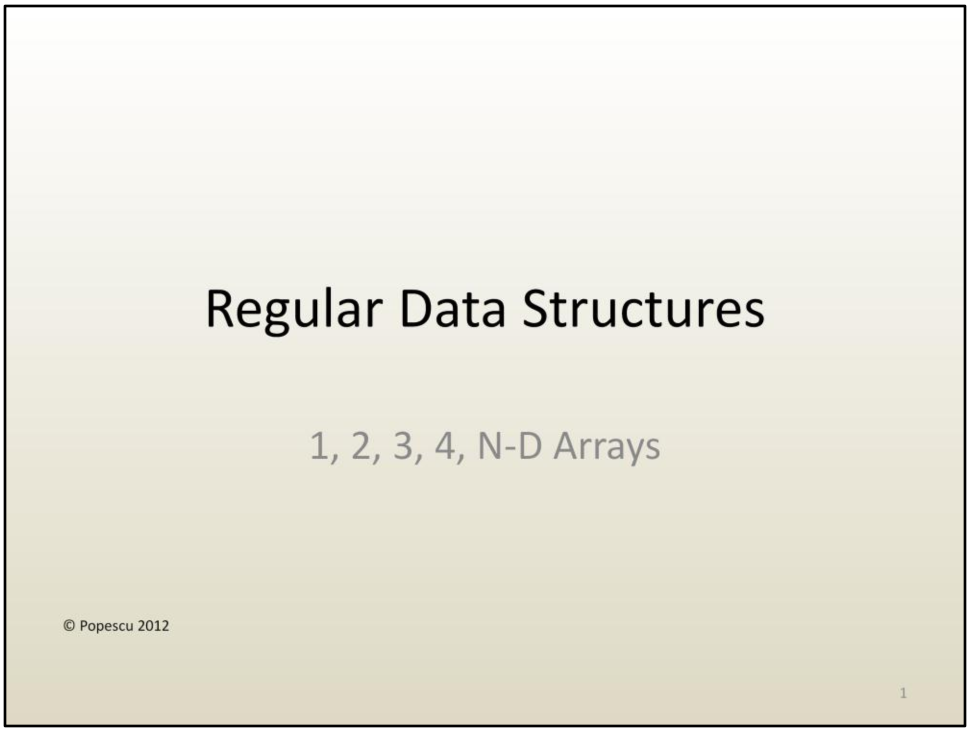
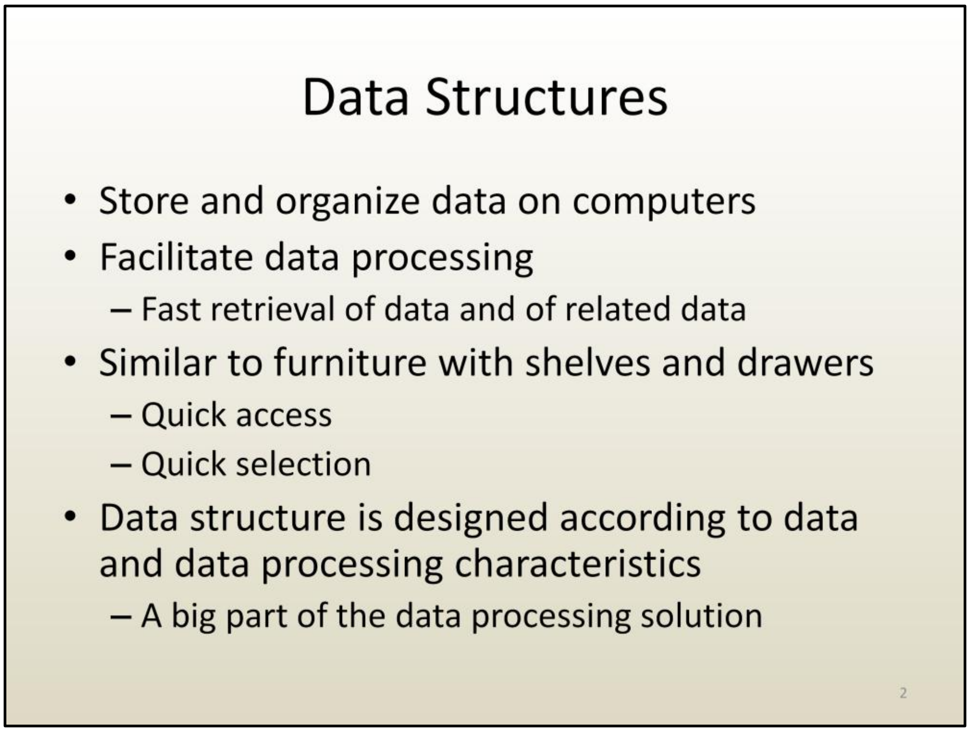


Regular Data Structures

1, 2, 3, 4, N-D Arrays

In this lecture we will begin exploring how data is actually organized (i.e. structured) to be stored, accessed, and processed on the computer.



Data Structures

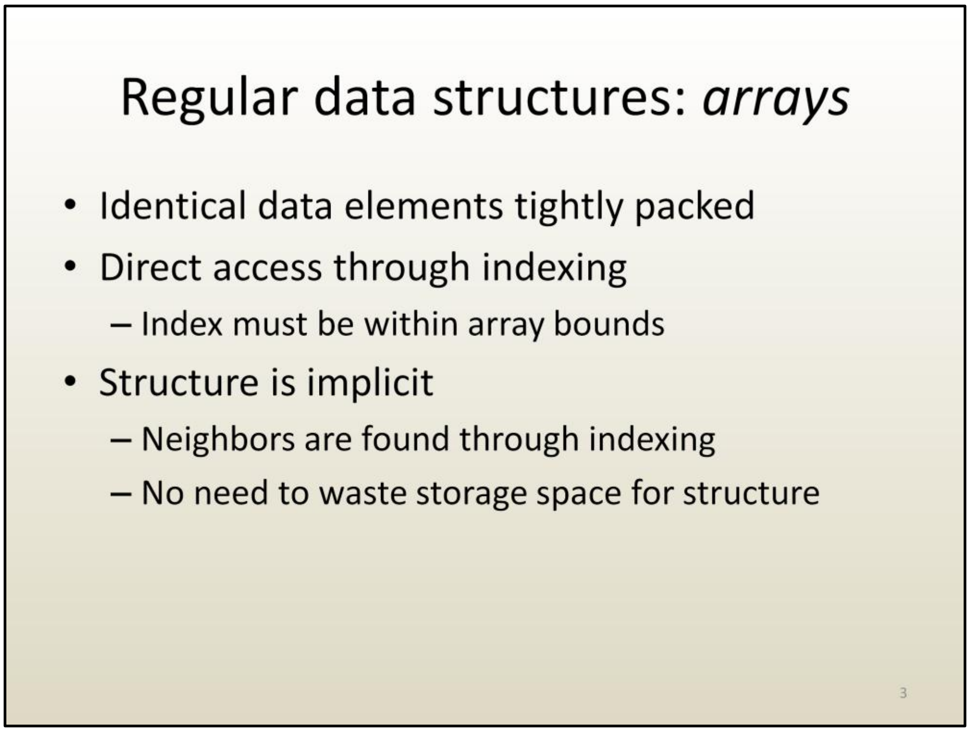
- Store and organize data on computers
- Facilitate data processing
  - Fast retrieval of data and of related data
- Similar to furniture with shelves and drawers
  - Quick access
  - Quick selection
- Data structure is designed according to data and data processing characteristics
  - A big part of the data processing solution

2

The data structure has great influence on the performance of data processing.

The data structures used for a particular data processing application have to let the application easily (i.e. quickly) retrieve the data currently being processed, as well as all the related data needed for the processing. For example in the case of blurring a digital image, the application needs to have easy access to the current pixel as well as to the neighbors of the current pixel.

The data structures to be used for a particular data processing application have to be designed in conjunction with the actual data processing step-by-step instructions, or algorithm.



Regular data structures: *arrays*

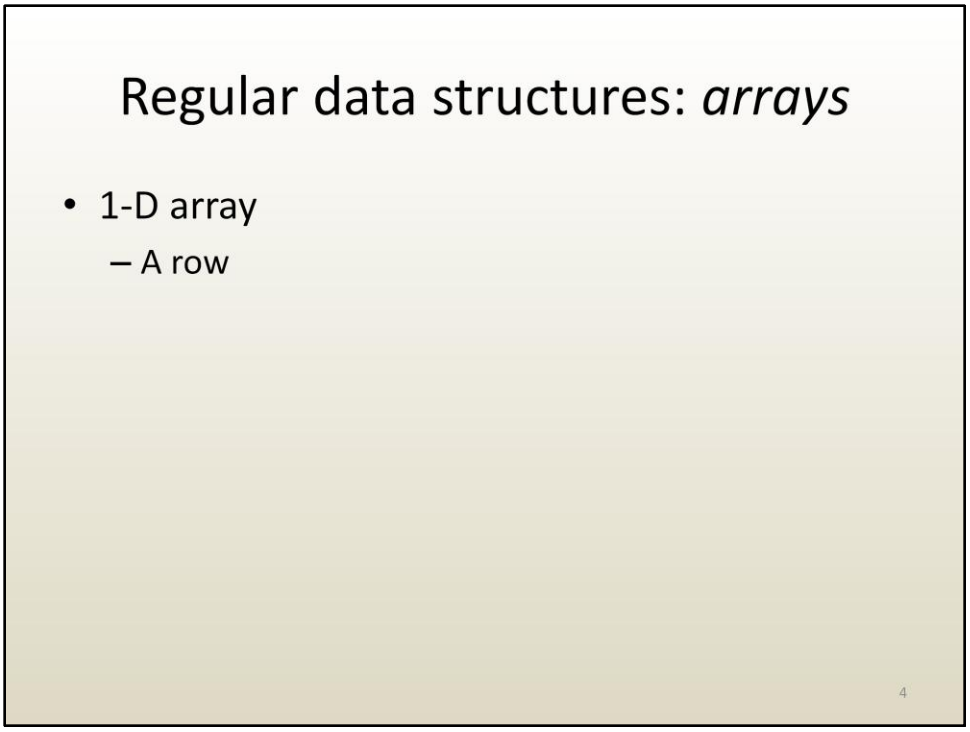
- Identical data elements tightly packed
- Direct access through indexing
  - Index must be within array bounds
- Structure is implicit
  - Neighbors are found through indexing
  - No need to waste storage space for structure

3

In this lecture we will look at regular, uniform data structures called arrays. Arrays are used very frequently in data processing.

An array is a regularly arranged collection of identical data elements. Each data element takes up the same amount of (memory) space. There is no space between adjacent elements. This allows direct access to any element in the array through indexing.

Arrays have implicit structure. In other words there is no special data to encode the structure. All memory used by the array goes towards storing the payload, i.e. the data that one wants to process.

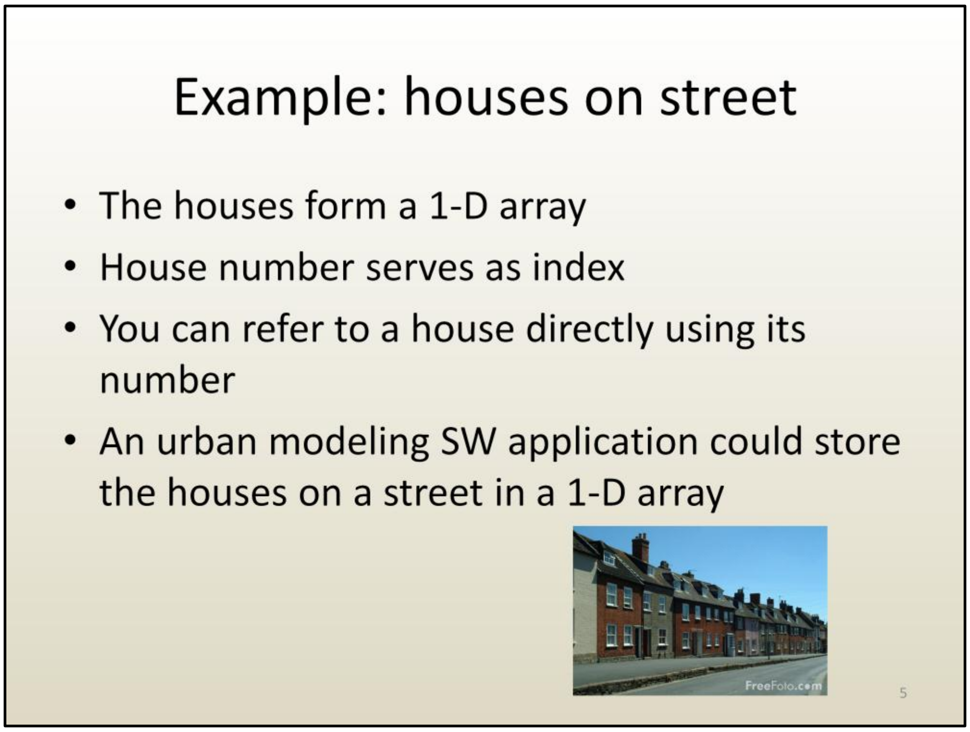


Regular data structures: *arrays*

- 1-D array
  - A row

4

The simplest array is one dimensional. Think of a 1-D array as of a row of elements.

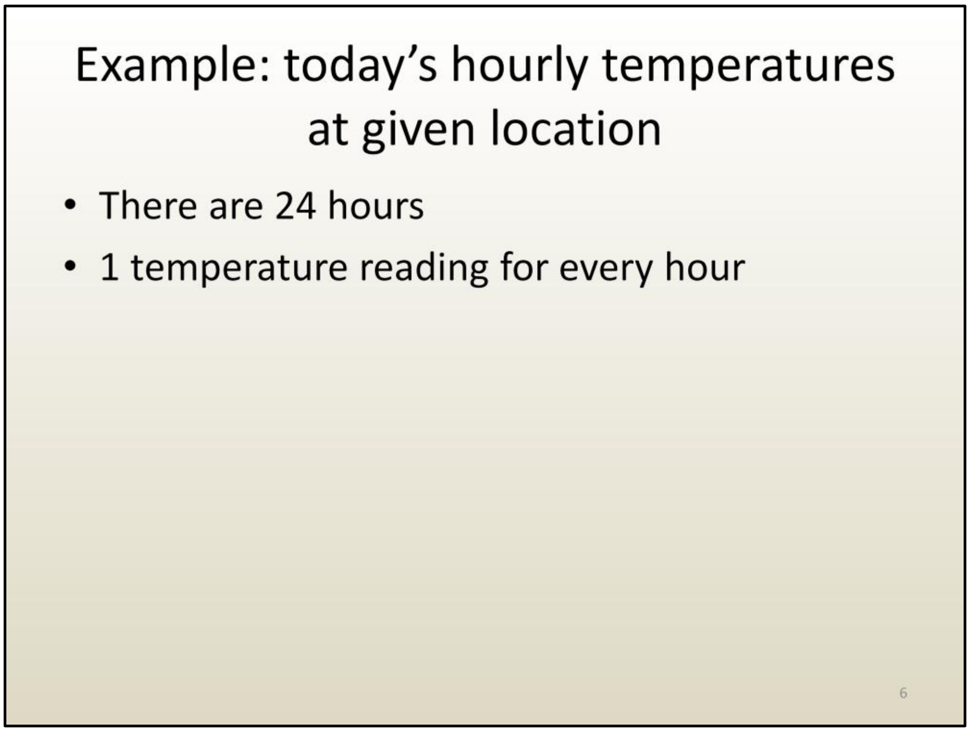


Example: houses on street

- The houses form a 1-D array
- House number serves as index
- You can refer to a house directly using its number
- An urban modeling SW application could store the houses on a street in a 1-D array

5

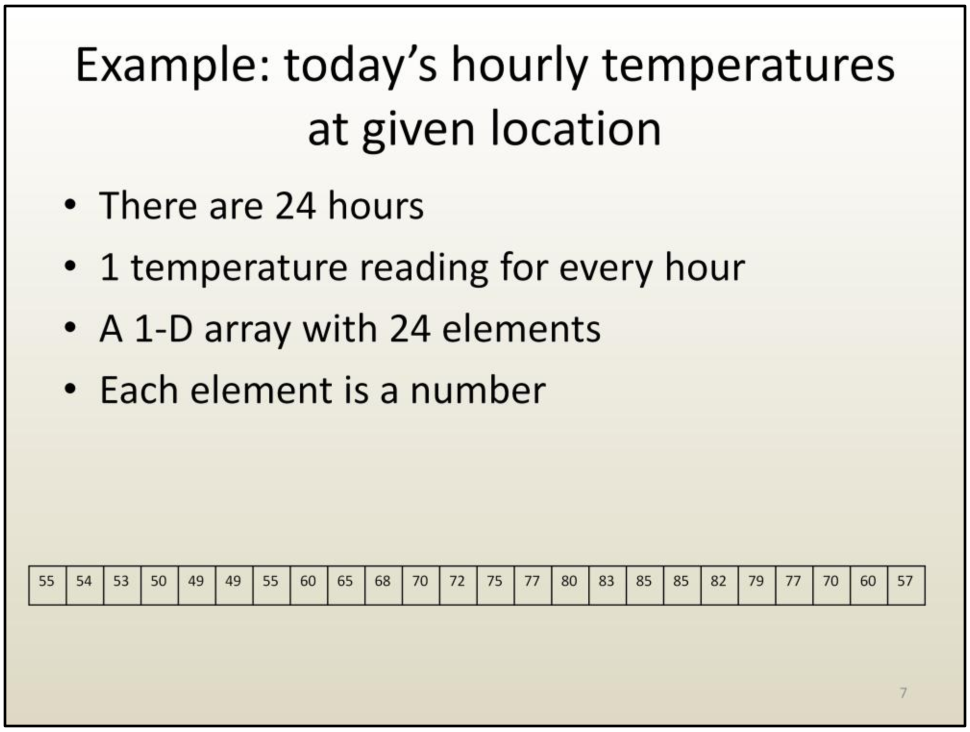
The houses on one side of a street can be thought of as an 1-D array.



Example: today's hourly temperatures at given location

- There are 24 hours
- 1 temperature reading for every hour

6



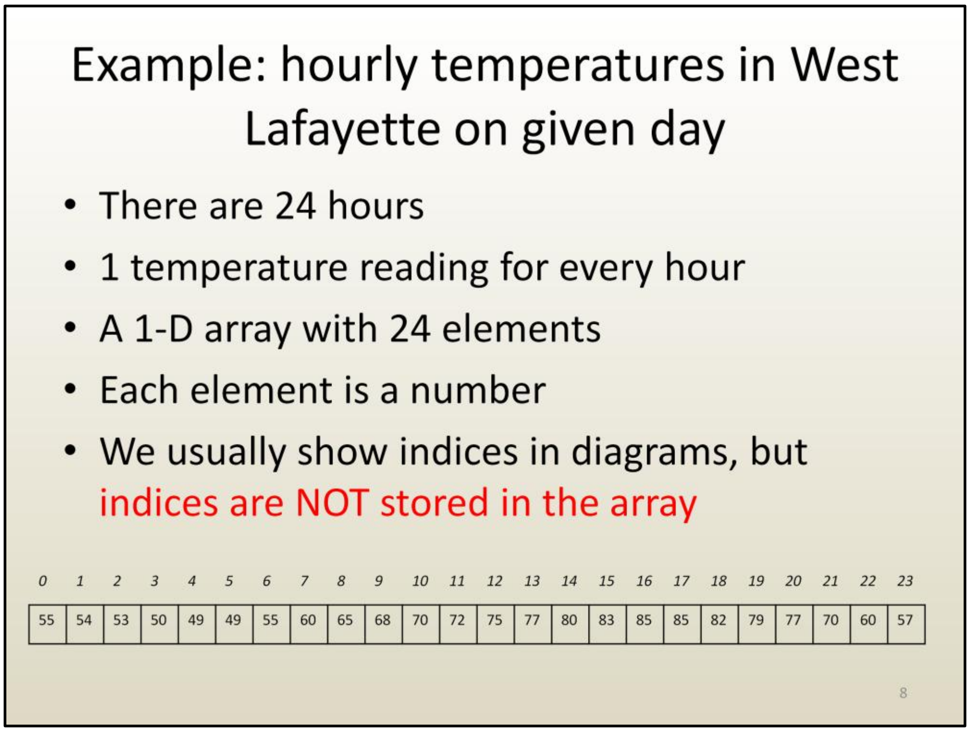
Example: today's hourly temperatures at given location

- There are 24 hours
- 1 temperature reading for every hour
- A 1-D array with 24 elements
- Each element is a number

55	54	53	50	49	49	55	60	65	68	70	72	75	77	80	83	85	85	82	79	77	70	60	57

7

Another example: the temperatures recorded every hour at a given location. An element is a number: the temperature at that hour. The index of the array corresponds to time (a 1-hour digitization of continuous time). There are 24 elements since there are 24 hours in a day.

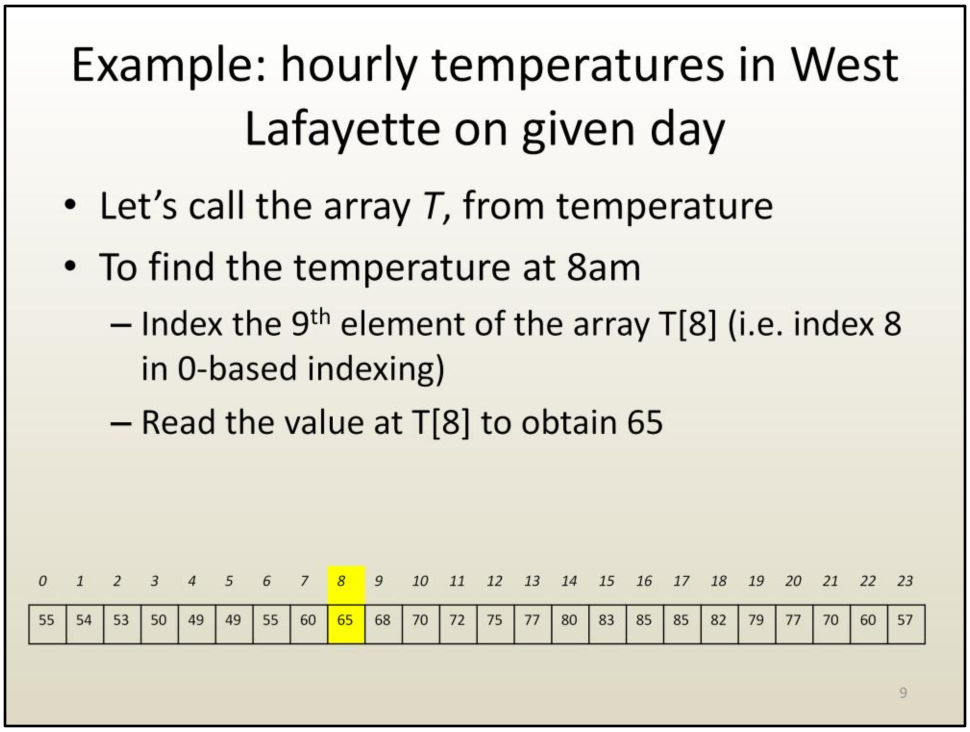


Example: hourly temperatures in West Lafayette on given day

- There are 24 hours
- 1 temperature reading for every hour
- A 1-D array with 24 elements
- Each element is a number
- We usually show indices in diagrams, but indices are NOT stored in the array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
55	54	53	50	49	49	55	60	65	68	70	72	75	77	80	83	85	85	82	79	77	70	60	57

Indices are shown in this and other diagrams for explanation purposes, but they are not actually stored in the array.



Example: hourly temperatures in West Lafayette on given day

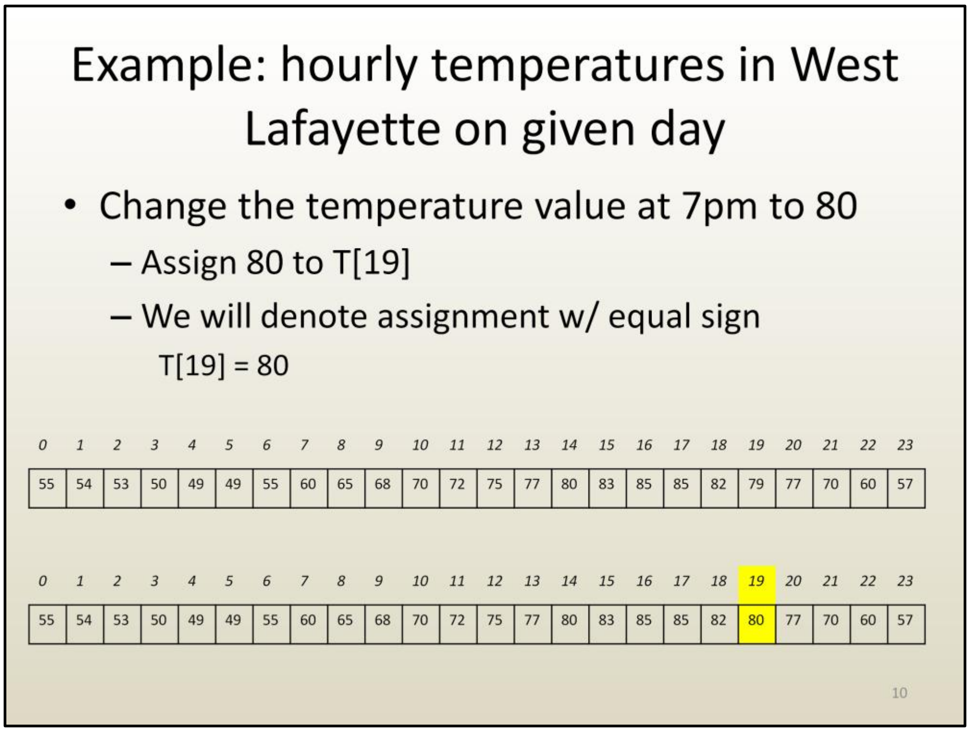
- Let's call the array *T*, from temperature
- To find the temperature at 8am
  - Index the 9th element of the array T[8] (i.e. index 8 in 0-based indexing)
  - Read the value at T[8] to obtain 65

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
55	54	53	50	49	49	55	60	65	68	70	72	75	77	80	83	85	85	82	79	77	70	60	57

9

Assuming the array is called T and that the first element corresponds to midnight, the temperature at 8am is found as T[8] and it is 60 degrees.

Note that we use 0-based indices. This means that the k-th element in the array has index k-1. The first index is 0, and the last index is n-1, if the array has n elements.



Example: hourly temperatures in West Lafayette on given day

- Change the temperature value at 7pm to 80
  - Assign 80 to T[19]
  - We will denote assignment w/ equal sign

    T[19] = 80

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*
55	54	53	50	49	49	55	60	65	68	70	72	75	77	80	83	85	85	82	79	77	70	60	57

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*
55	54	53	50	49	49	55	60	65	68	70	72	75	77	80	83	85	85	82	80	77	70	60	57

10

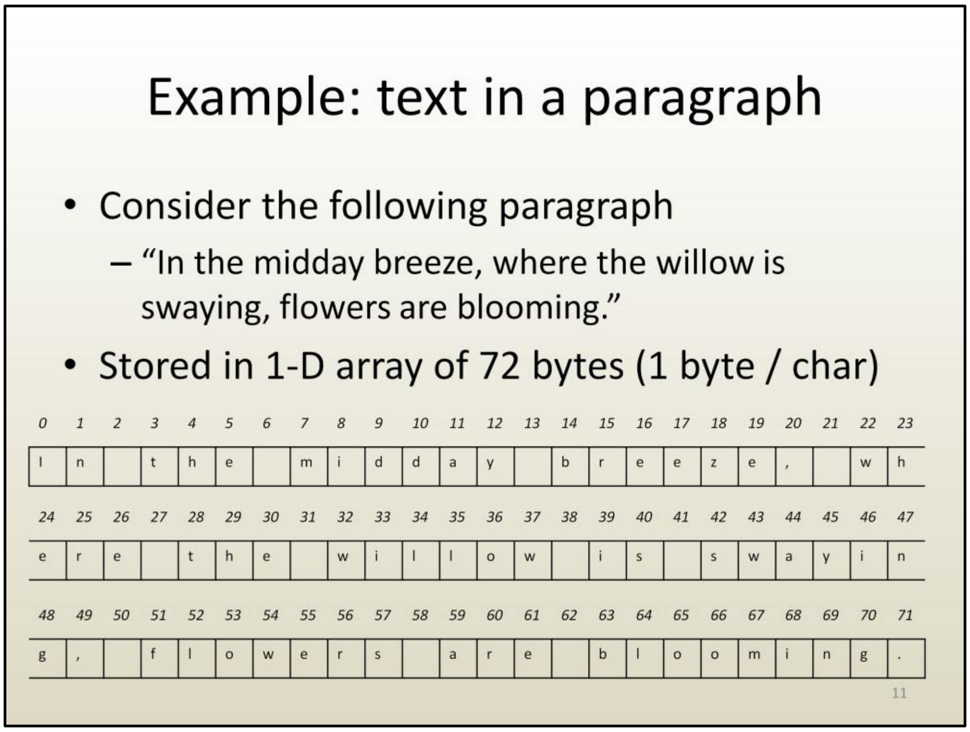
For those of you familiar with the 24-h format of time, you'll know right away that 7pm is 19:00.

Let's say we want to change the temperature for 7pm. That means we need to assign to element T[19] a different value. We write that using the equal sign.

T[19] = 80 means overwrite whatever value was stored at element 19 with the new value 80. In other words, the value 80 is assigned to T[19].

Programs have to be very careful as to not write "left" of the first or "right" of the last element of the array. This means that one cannot have indices that are negative or greater than or equal to n, where n is the size of the array (i.e. the number of elements in the array). In this example, valid indices i have to satisfy the condition 0 <= i < 24. The valid index values are 0, 1, 2, …, 23.

A large percentage of all SW crashes are due to writing outside the array limits. What happens when you write outside the array limits? You overwrite some other data that happens to be stored before or after the array. That data becomes corrupt and when the SW attempts to use it a crash occurs.



Example: text in a paragraph

- Consider the following paragraph
  - "In the midday breeze, where the willow is swaying, flowers are blooming."
- Stored in 1-D array of 72 bytes (1 byte / char)

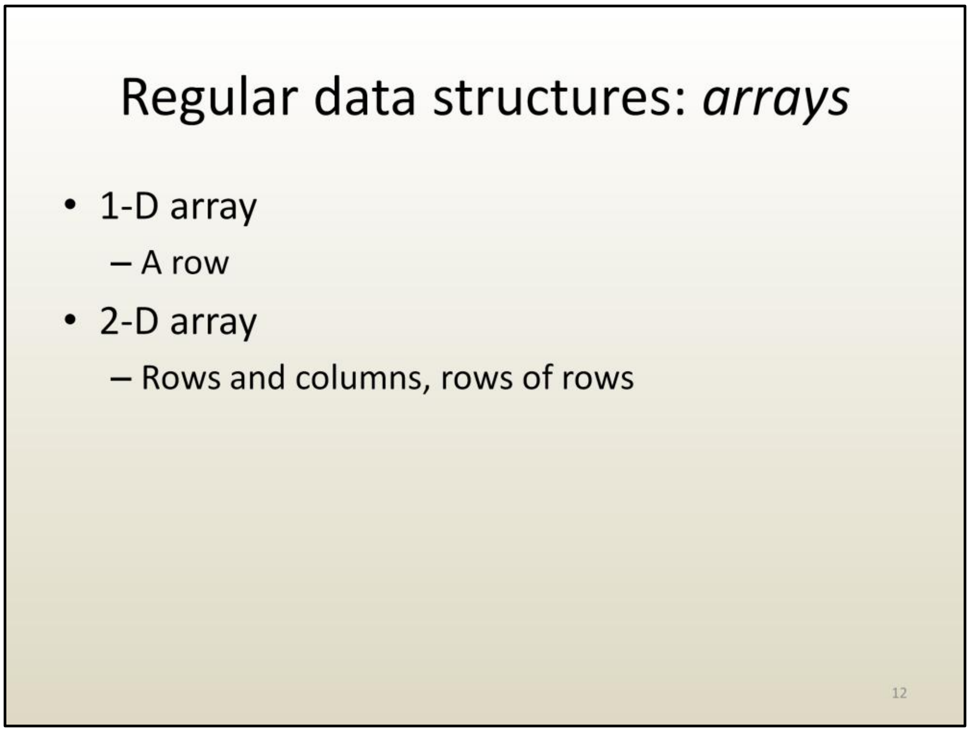
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*
I	n		t	h	e		m	i	d	d	a	y		b	r	e	e	z	e	,		w	h

24	*25*	*26*	*27*	*28*	*29*	*30*	*31*	*32*	*33*	*34*	*35*	*36*	*37*	*38*	*39*	*40*	*41*	*42*	*43*	*44*	*45*	*46*	*47*
e	r	e		t	h	e		w	i	l	l	o	w		i	s		s	w	a	y	i	n

48	*49*	*50*	*51*	*52*	*53*	*54*	*55*	*56*	*57*	*58*	*59*	*60*	*61*	*62*	*63*	*64*	*65*	*66*	*67*	*68*	*69*	*70*	*71*
g	,		f	l	o	w	e	r	s		a	r	e		b	l	o	o	m	i	n	g	.

11

Here is a 1-D array that stores text. One element takes up one byte and stores one character. Note that spaces, commas, and other special characters have their own array elements.



Regular data structures: *arrays*

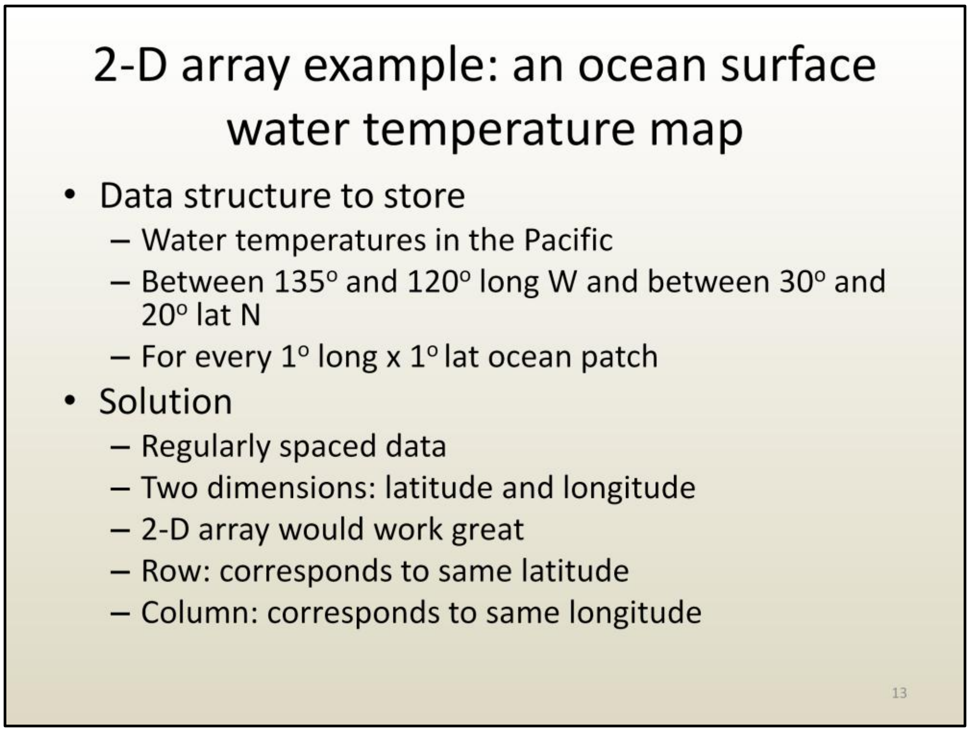
- 1-D array
  - A row
- 2-D array
  - Rows and columns, rows of rows

12

Arrays do not have to be one dimensional, they can be of any dimension.

Let’s talk about two dimensional or 2-D arrays. Whereas a 1-D array was equivalent to a single row, a 2-D array has multiple rows. One can also think of a 2-D array as multiple columns.

A 2-D array is a 1-D array of 1-D arrays.



2-D array example: an ocean surface water temperature map

- Data structure to store
  - Water temperatures in the Pacific
  - Between 135° and 120° long W and between 30° and 20° lat N
  - For every 1° long x 1° lat ocean patch
- Solution
  - Regularly spaced data
  - Two dimensions: latitude and longitude
  - 2-D array would work great
  - Row: corresponds to same latitude
  - Column: corresponds to same longitude

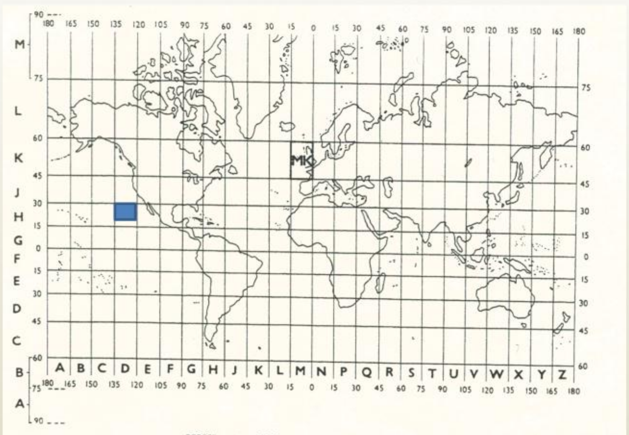
13

Here is an example. You are asked to design a data structure to store ocean surface water temperatures. You have to cover a certain longitude and latitude range. You are also told that the spatial resolution of your data has to be 1 degree of longitude by 1 degree of latitude. Spatial resolution means the number of data points per unit of surface area. (Temporal resolution means how many data points are collected per unit of time.)

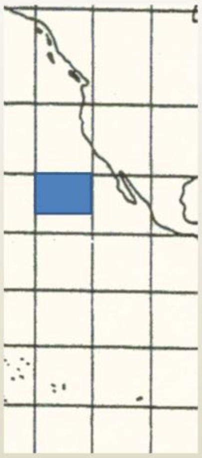
Since the pieces of data are identical, i.e. they are all one number measuring temperature, and since the data covers a 2-D domain, the obvious choice is to use a 2-D array.

A row in the 2-D array corresponds to water temperatures measured at the same latitude. A column means same longitude.

2-D array example: an ocean surface water temperature map

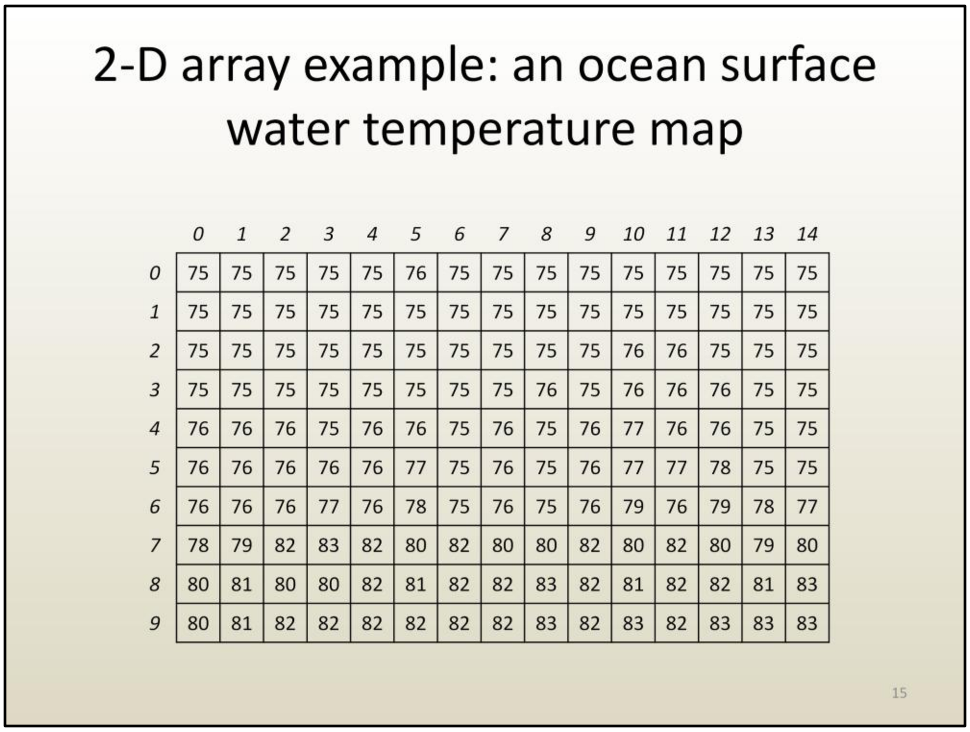


GEOREF system of 15° quadrangle identification letters.



14

Here is a visualization of the ocean patch covered by our data (blue rectangle).

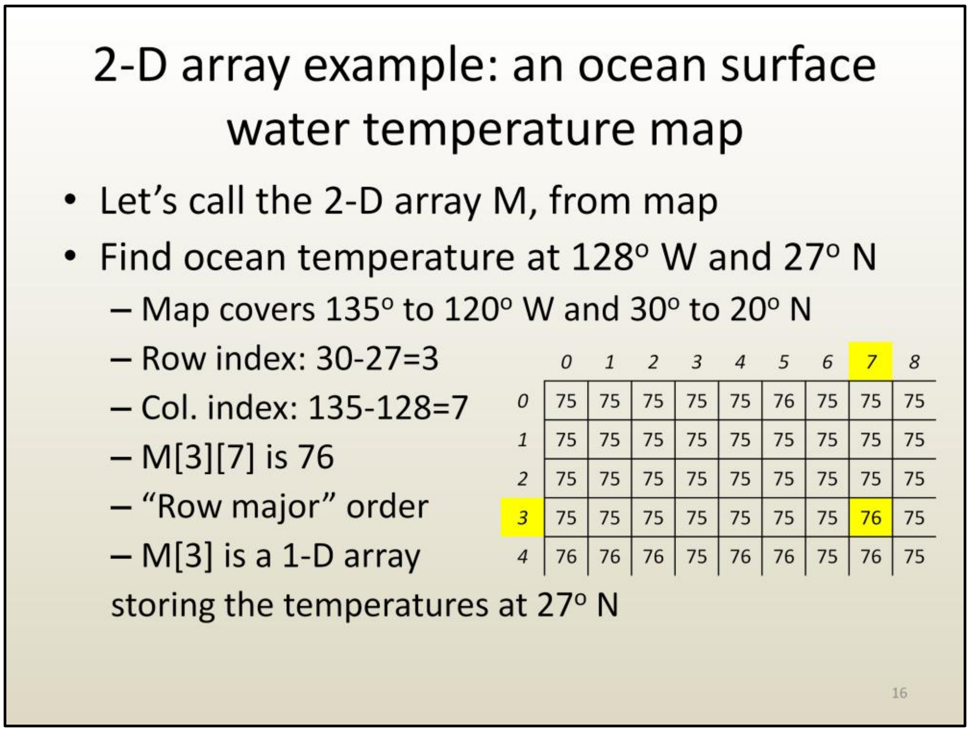


2-D array example: an ocean surface water temperature map

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	75	75	75	75	75	76	75	75	75	75	75	75	75	75	75
1	75	75	75	75	75	75	75	75	75	75	75	75	75	75	75
2	75	75	75	75	75	75	75	75	75	75	76	76	75	75	75
3	75	75	75	75	75	75	75	75	76	75	76	76	76	75	75
4	76	76	76	75	76	76	75	76	75	76	77	76	76	75	75
5	76	76	76	76	76	77	75	76	75	76	77	77	78	75	75
6	76	76	76	77	76	78	75	76	75	76	79	76	79	78	77
7	78	79	82	83	82	80	82	80	80	82	80	82	80	79	80
8	80	81	80	80	82	81	82	82	83	82	81	82	82	81	83
9	80	81	82	82	82	82	82	82	83	82	83	82	83	83	83

15

And here is the 2-D array. There are 15 columns and 10 rows. 15 columns because our ocean patch spans 15 degrees of longitude, and because we want one element per degree of longitude. 10 rows because our ocean patch spans 10 degrees of latitude, and because we want one element per degree of latitude.



2-D array example: an ocean surface water temperature map

- Let’s call the 2-D array M, from map
- Find ocean temperature at 128^o W and 27^o N
  - Map covers 135^o to 120^o W and 30^o to 20^o N
  - Row index: 30-27=3
  - Col. index: 135-128=7
  - M[3][7] is 76
  - “Row major” order
  - M[3] is a 1-D array
  storing the temperatures at 27^o N

	0	1	2	3	4	5	6	7	8
0	75	75	75	75	75	76	75	75	75
1	75	75	75	75	75	75	75	75	75
2	75	75	75	75	75	75	75	75	75
3	75	75	75	75	75	75	75	76	75
4	76	76	76	75	76	76	75	76	75

16

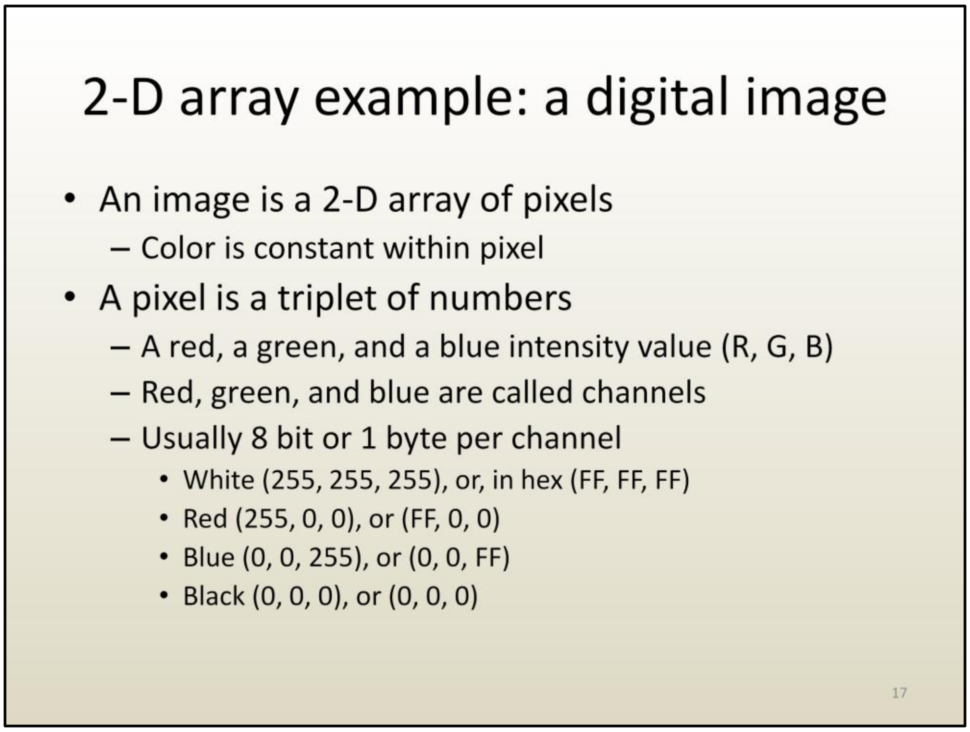
Whereas in the temperatures over 24 h 1-D array example one could quickly convert from time of day in hours to array index, in the present example conversion form longitude/latitude coordinates to array element is a bit more involved.

In order to find the ocean temperature at 128^0 W and 27^0N we need to find the row index and the column index where to read the array.

The row is found by computing how far we are from the first row. The first row corresponds to 30^0N. Consequently the row for 27^0N has index 30-27=3.

The column is found by computing how far we are from the first column: 135-128=7.

The element we want is M[3][7]. The array is stored in row major order, meaning that the first index after M selects the row, and the second index selects the column.



2-D array example: a digital image

- An image is a 2-D array of pixels
  - Color is constant within pixel
- A pixel is a triplet of numbers
  - A red, a green, and a blue intensity value (R, G, B)
  - Red, green, and blue are called channels
  - Usually 8 bit or 1 byte per channel
    - White (255, 255, 255), or, in hex (FF, FF, FF)
    - Red (255, 0, 0), or (FF, 0, 0)
    - Blue (0, 0, 255), or (0, 0, FF)
    - Black (0, 0, 0), or (0, 0, 0)

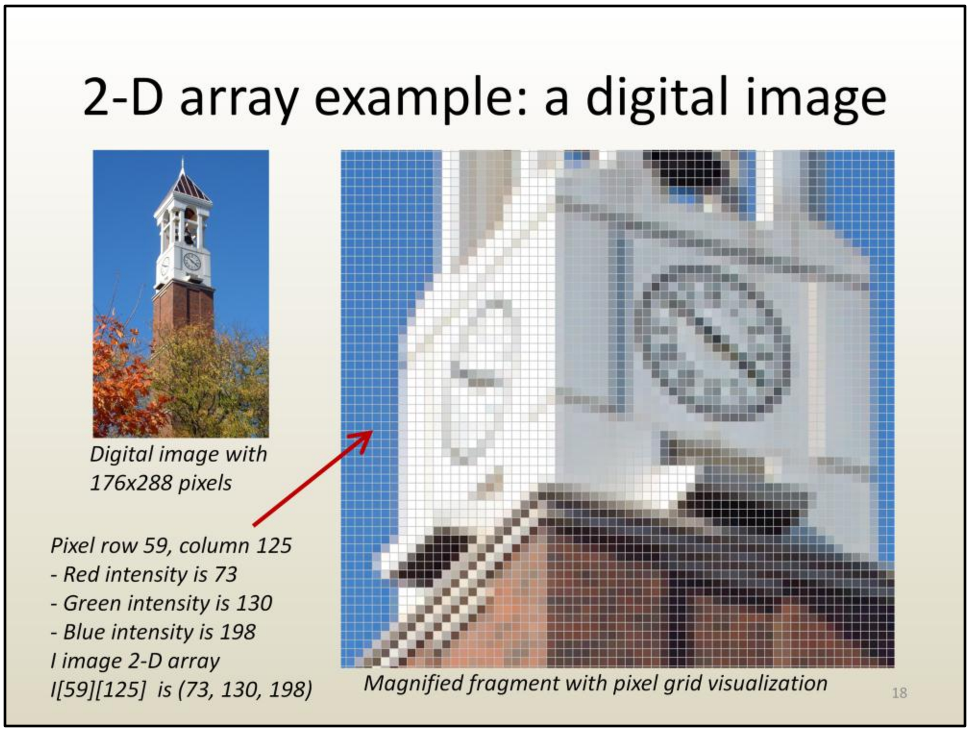
17

Another example of a 2-D array is a digital image.

The array element is a pixel. For black and white image, a pixel stores a single intensity value.

For color images, the pixel stores 3 intensity values: one for red, one for blue and one for green. Many (but not all) colors seen by the human eye can be created using these 3 RGB values.

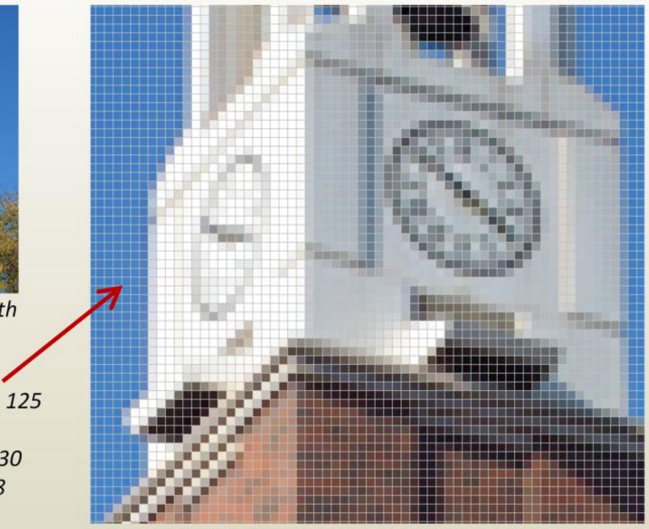
Since there is one byte or 8 bits per channel (i.e. per intensity value), a channel value can be represented with 2 hexadecimal digits (each covering 4 bits).



2-D array example: a digital image

Digital image with 176x288 pixels

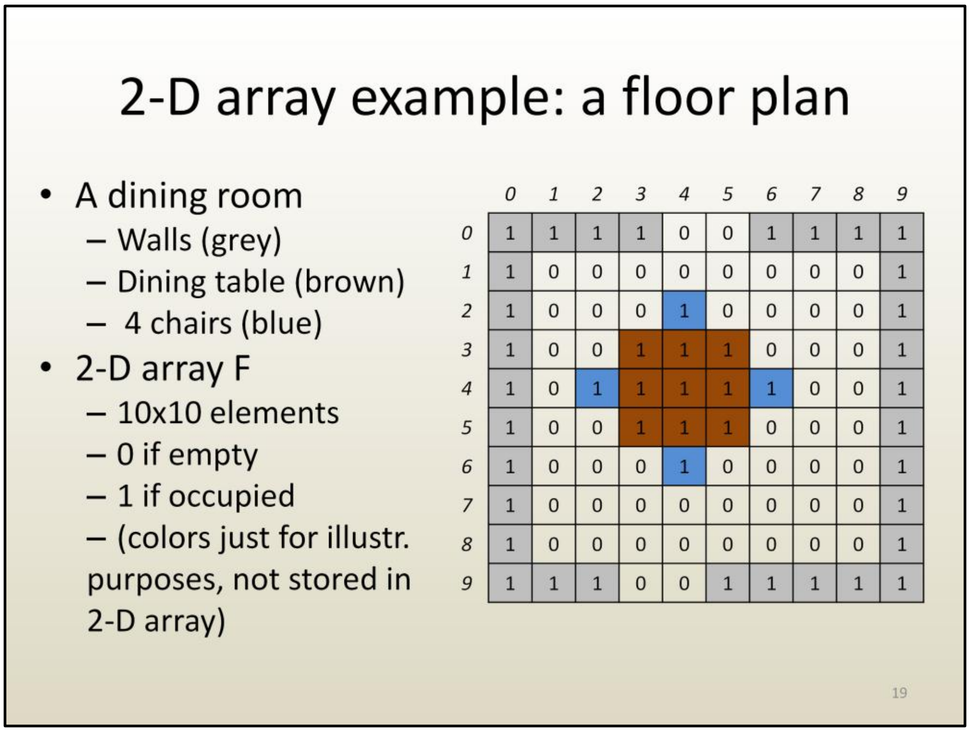
Pixel row 59, column 125
- Red intensity is 73
- Green intensity is 130
- Blue intensity is 198
I image 2-D array
I[59][125] is (73, 130, 198)



Magnified fragment with pixel grid visualization

18

The selected pixel is blue, thus the blue channel dominates. The other channels are not 0, as the pixel is not “pure” blue.



2-D array example: a floor plan

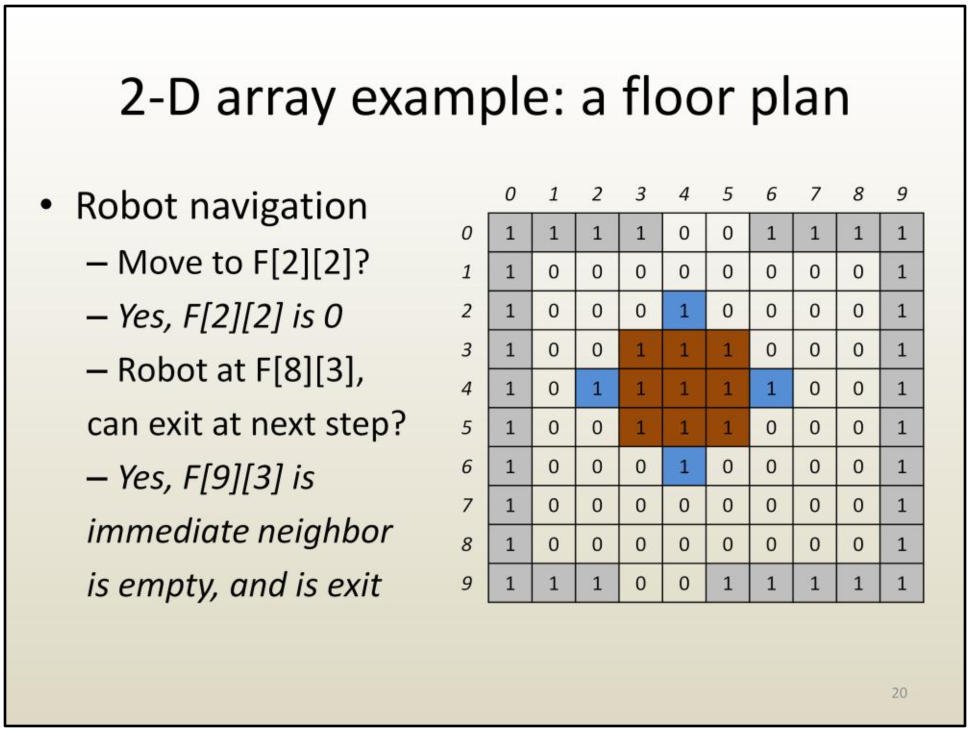
- A dining room
  - Walls (grey)
  - Dining table (brown)
  - 4 chairs (blue)
- 2-D array F
  - 10x10 elements
  - 0 if empty
  - 1 if occupied
  - (colors just for illustr. purposes, not stored in 2-D array)

	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1
2	1	0	0	0	1	0	0	0	0	1
3	1	0	0	1	1	1	0	0	0	1
4	1	0	1	1	1	1	1	0	0	1
5	1	0	0	1	1	1	0	0	0	1
6	1	0	0	0	1	0	0	0	0	1
7	1	0	0	0	0	0	0	0	0	1
8	1	0	0	0	0	0	0	0	0	1
9	1	1	1	0	0	1	1	1	1	1

19

Here is another example where a 2-D array is used to model the floor plan of a room.

Here there is only 1 bit per array element.



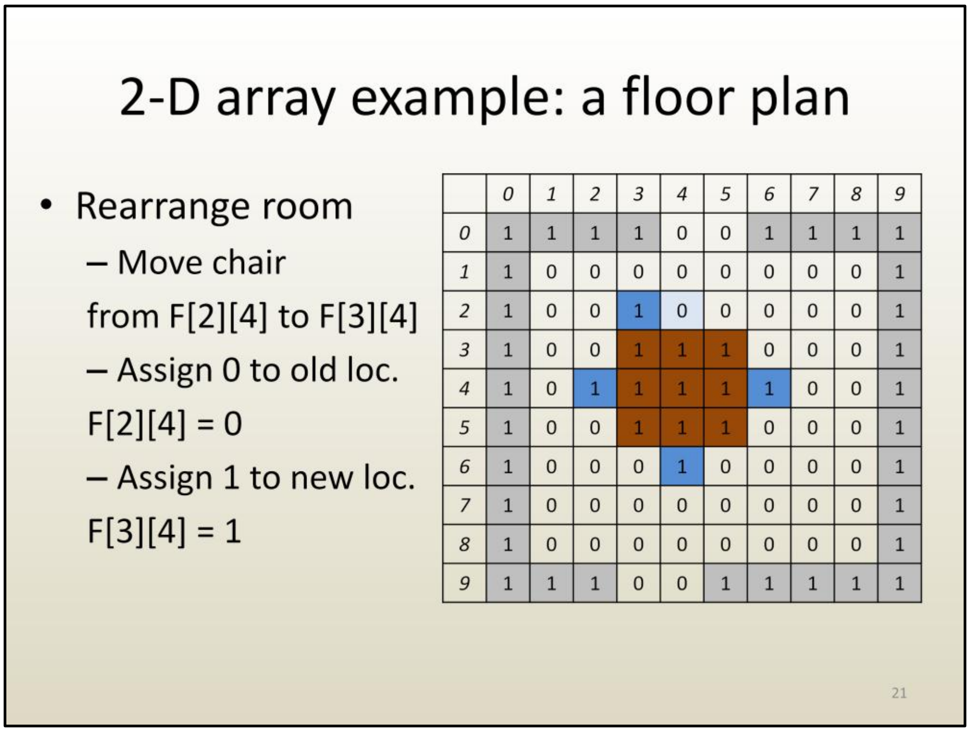
2-D array example: a floor plan

- Robot navigation
  - Move to F[2][2]?
  - *Yes, F[2][2] is 0*
  - Robot at F[8][3], can exit at next step?
  - *Yes, F[9][3] is immediate neighbor is empty, and is exit*

	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1
2	1	0	0	0	1	0	0	0	0	1
3	1	0	0	1	1	1	0	0	0	1
4	1	0	1	1	1	1	1	0	0	1
5	1	0	0	1	1	1	0	0	0	1
6	1	0	0	0	1	0	0	0	0	1
7	1	0	0	0	0	0	0	0	0	1
8	1	0	0	0	0	0	0	0	0	1
9	1	1	1	0	0	1	1	1	1	1

20

A robot navigation application might use the 2-D array to check for collisions, to find exists, etc.



2-D array example: a floor plan

- Rearrange room
  - Move chair

    from F[2][4] to F[3][4]
  - Assign 0 to old loc.

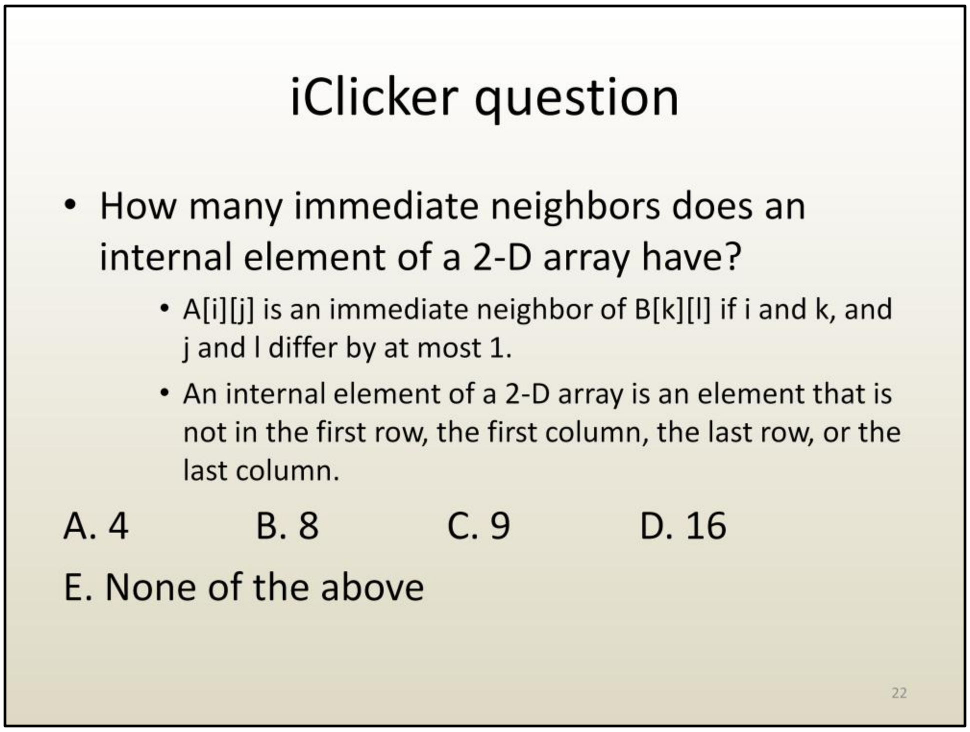
    F[2][4] = 0
  - Assign 1 to new loc.

    F[3][4] = 1

	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1
2	1	0	0	1	0	0	0	0	0	1
3	1	0	0	1	1	1	0	0	0	1
4	1	0	1	1	1	1	1	0	0	1
5	1	0	0	1	1	1	0	0	0	1
6	1	0	0	0	1	0	0	0	0	1
7	1	0	0	0	0	0	0	0	0	1
8	1	0	0	0	0	0	0	0	0	1
9	1	1	1	0	0	1	1	1	1	1

21

When the chair moves, it needs to be assigned to the new location and erased from the old location.



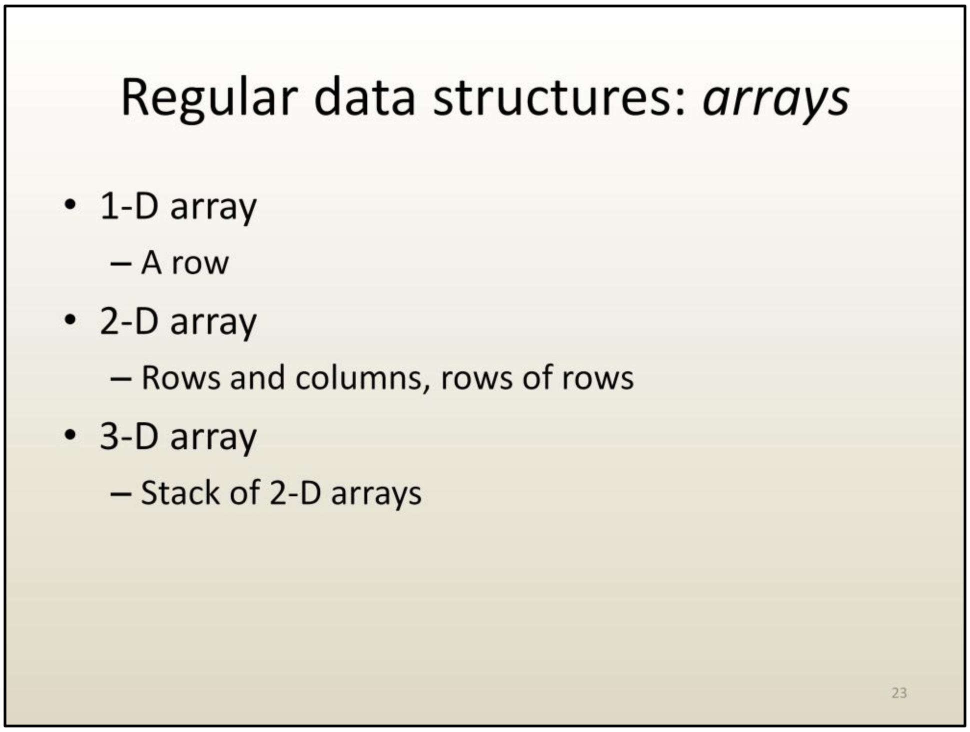
iClicker question

- How many immediate neighbors does an internal element of a 2-D array have?
  - A[i][j] is an immediate neighbor of B[k][l] if i and k, and j and l differ by at most 1.
  - An internal element of a 2-D array is an element that is not in the first row, the first column, the last row, or the last column.

A. 4 B. 8 C. 9 D. 16

E. None of the above

22

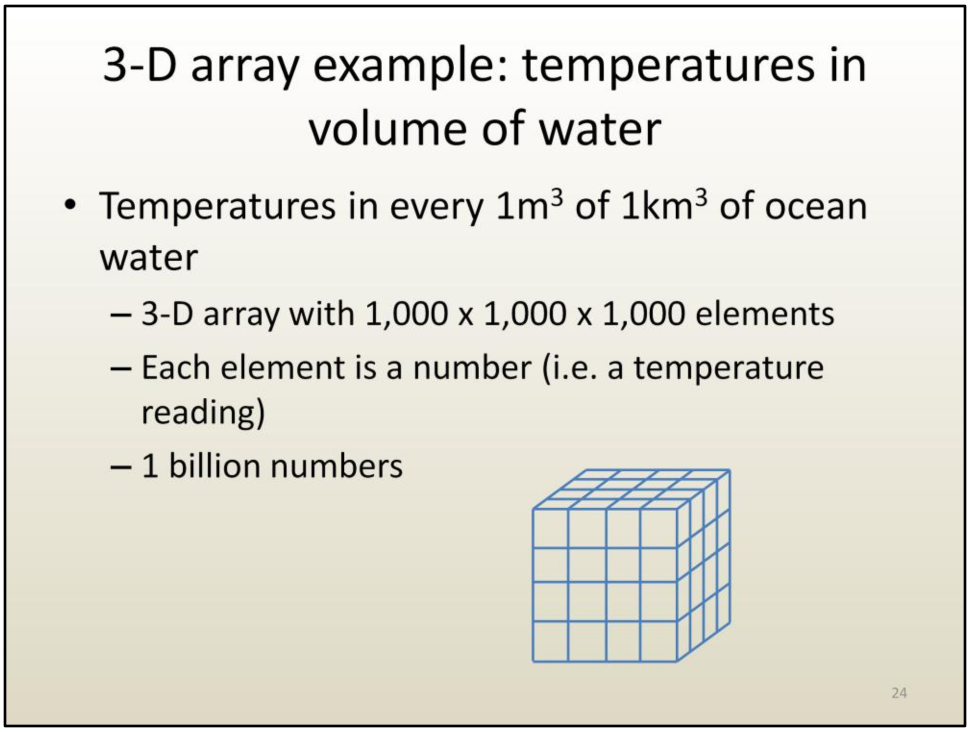


Regular data structures: *arrays*

- 1-D array
  - A row
- 2-D array
  - Rows and columns, rows of rows
- 3-D array
  - Stack of 2-D arrays

23

Let’s now look at 3-D arrays. A 3-D array can be seen as a 1-D array of 2-D arrays.

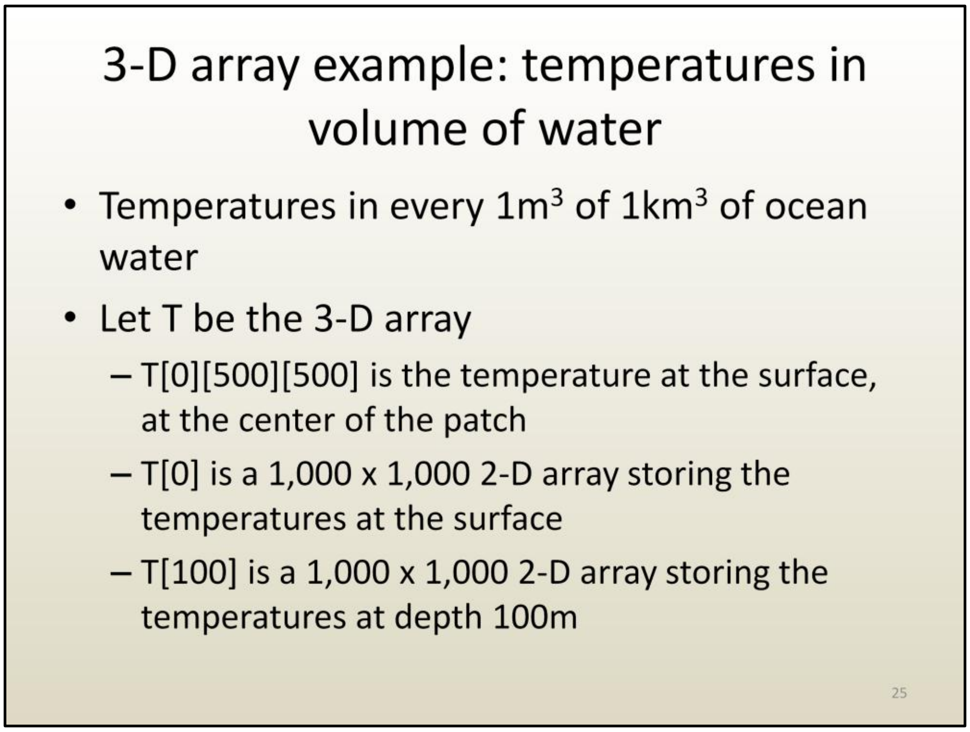


3-D array example: temperatures in volume of water

- Temperatures in every $1m^3$ of $1km^3$ of ocean water
  - 3-D array with 1,000 x 1,000 x 1,000 elements
  - Each element is a number (i.e. a temperature reading)
  - 1 billion numbers

24

The previous example can be extended to store water temperatures at multiple depths for every surface location.

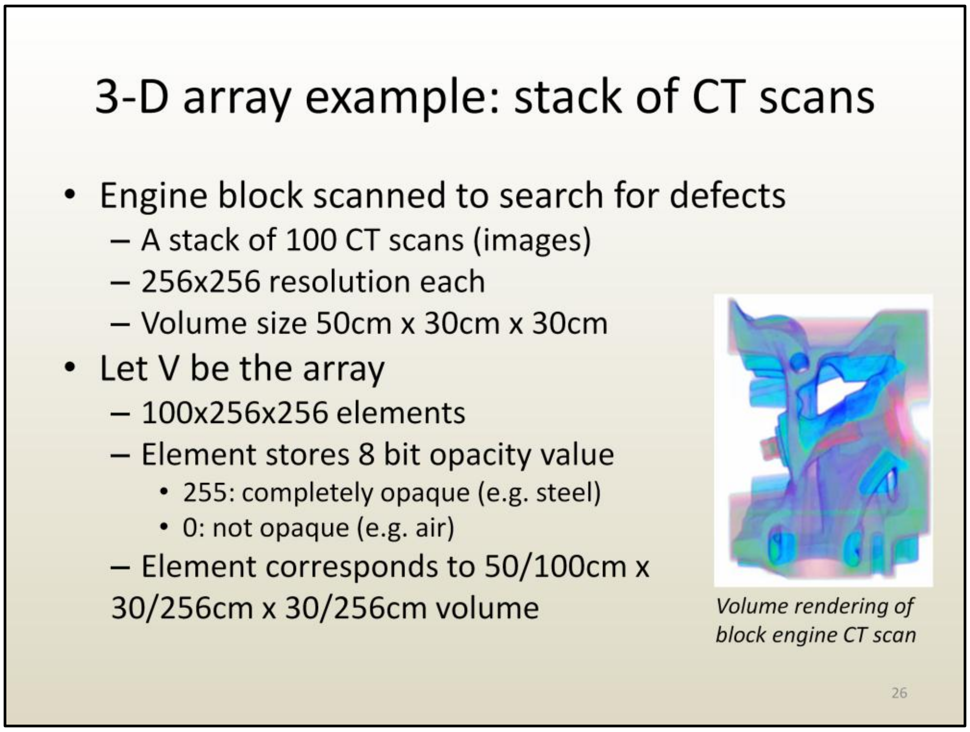


3-D array example: temperatures in volume of water

- Temperatures in every $1m^3$ of $1km^3$ of ocean water
- Let T be the 3-D array
  - T[0][500][500] is the temperature at the surface, at the center of the patch
  - T[0] is a 1,000 x 1,000 2-D array storing the temperatures at the surface
  - T[100] is a 1,000 x 1,000 2-D array storing the temperatures at depth 100m

25

The 3-D array is accessed using 3 indices. The first index tells you the depth. The second one tells you the latitude, the third one the longitude.



3-D array example: stack of CT scans

- Engine block scanned to search for defects
  - A stack of 100 CT scans (images)
  - 256x256 resolution each
  - Volume size 50cm x 30cm x 30cm
- Let V be the array
  - 100x256x256 elements
  - Element stores 8 bit opacity value
    - 255: completely opaque (e.g. steel)
    - 0: not opaque (e.g. air)
  - Element corresponds to 50/100cm x 30/256cm x 30/256cm volume

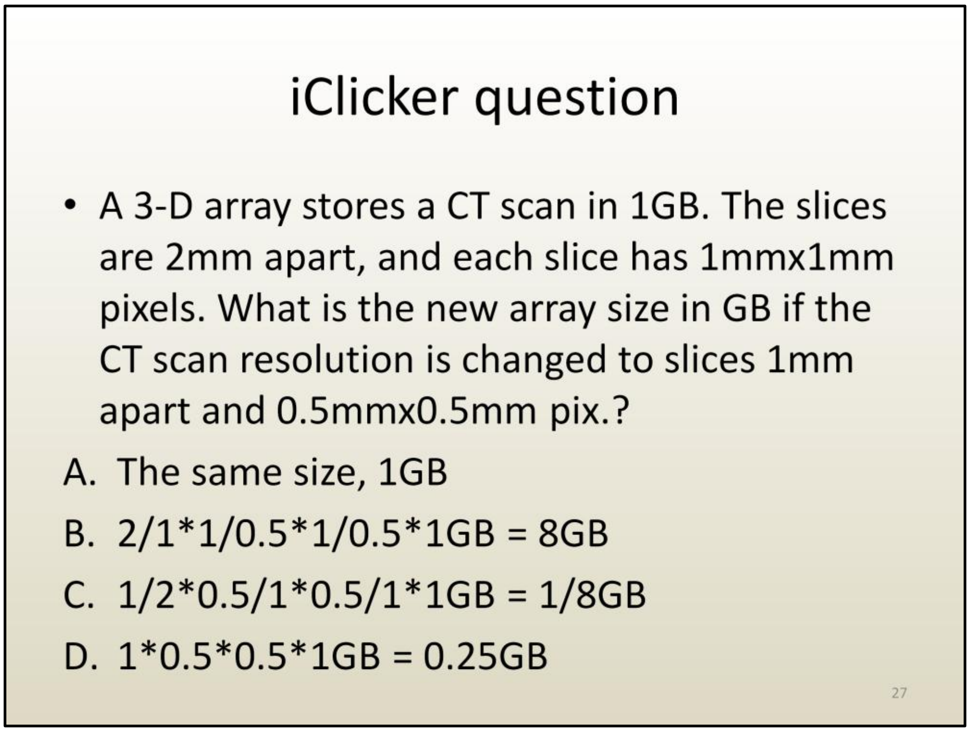
Volume rendering of block engine CT scan

26

Here is another example of a 3-D array: a stack of CT scans.

One element is stored with one byte. The element corresponds to a box of size 50/100cm x 30/256cm x 30/256cm. Whereas an image element is called a pixel, a volume element is called a voxel.

Specialized algorithms called volume rendering algorithms can visualize the 3-D array of opacities, see image.



iClicker question

- A 3-D array stores a CT scan in 1GB. The slices are 2mm apart, and each slice has 1mmx1mm pixels. What is the new array size in GB if the CT scan resolution is changed to slices 1mm apart and 0.5mmx0.5mm pix.?

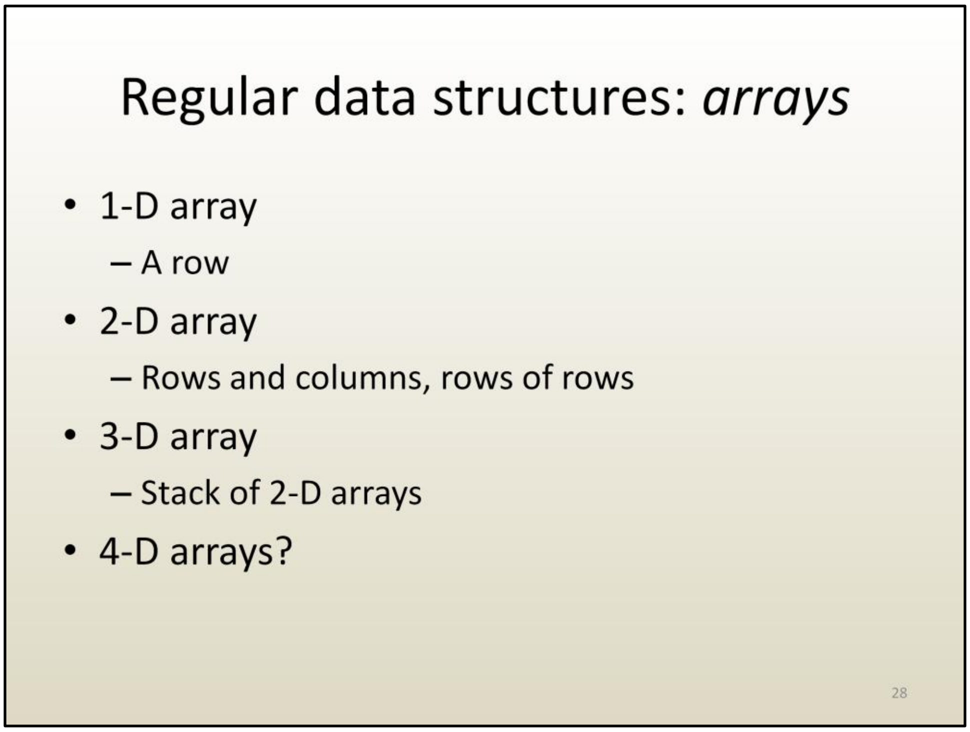
A. The same size, 1GB

B. 2/1*1/0.5*1/0.5*1GB = 8GB

C. 1/2*0.5/1*0.5/1*1GB = 1/8GB

D. 1*0.5*0.5*1GB = 0.25GB

27

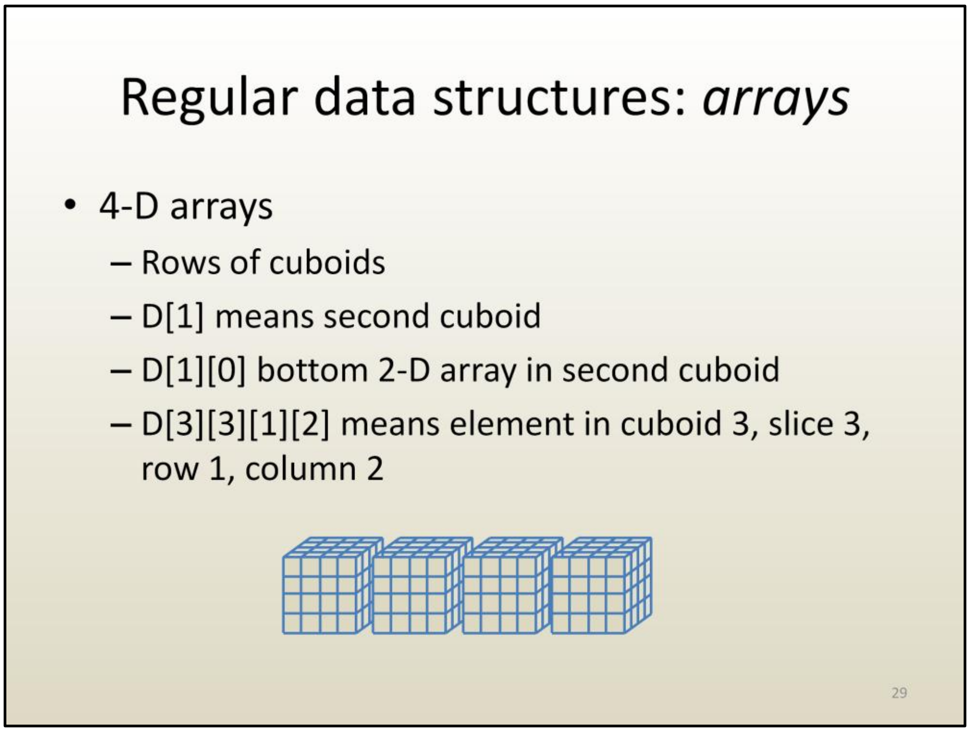


Regular data structures: *arrays*

- 1-D array
  - A row
- 2-D array
  - Rows and columns, rows of rows
- 3-D array
  - Stack of 2-D arrays
- 4-D arrays?

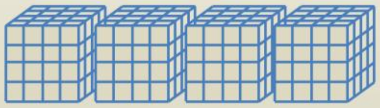
28

How about arrays with dimensionality higher than 3? Such arrays exist, space has 3 dimensions, but arrays can have any number of dimensions.



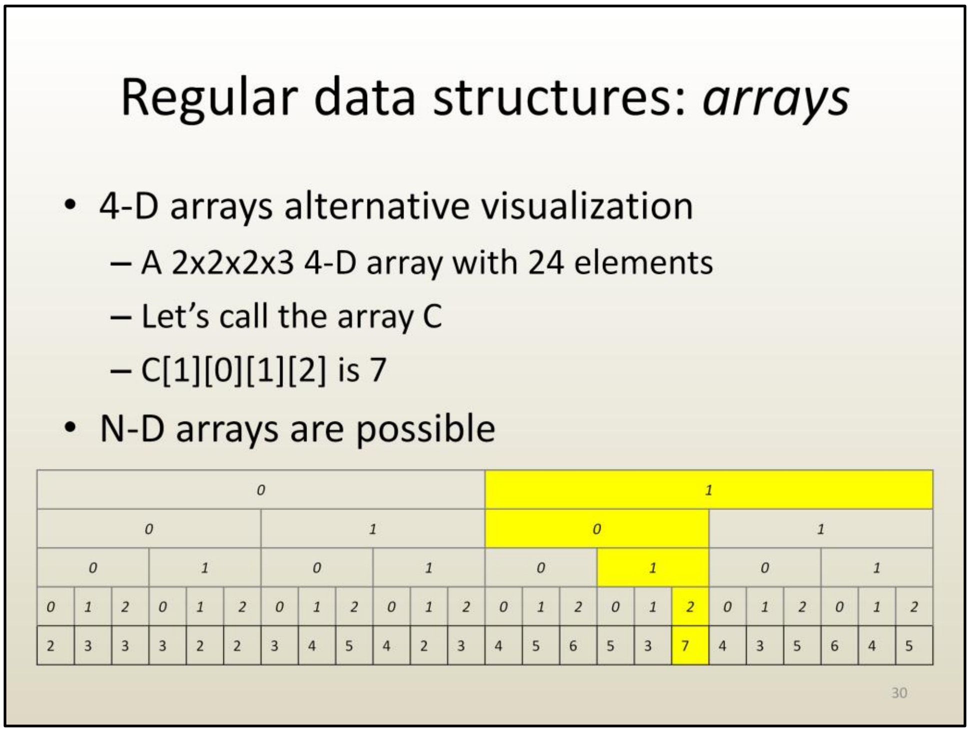
Regular data structures: *arrays*

- 4-D arrays
  - Rows of cuboids
  - D[1] means second cuboid
  - D[1][0] bottom 2-D array in second cuboid
  - D[3][3][1][2] means element in cuboid 3, slice 3, row 1, column 2



29

Here is a 4-D array: a 1-D array of cuboids.



Regular data structures: *arrays*

- 4-D arrays alternative visualization
  - A 2x2x2x3 4-D array with 24 elements
  - Let's call the array C
  - C[1][0][1][2] is 7
- N-D arrays are possible

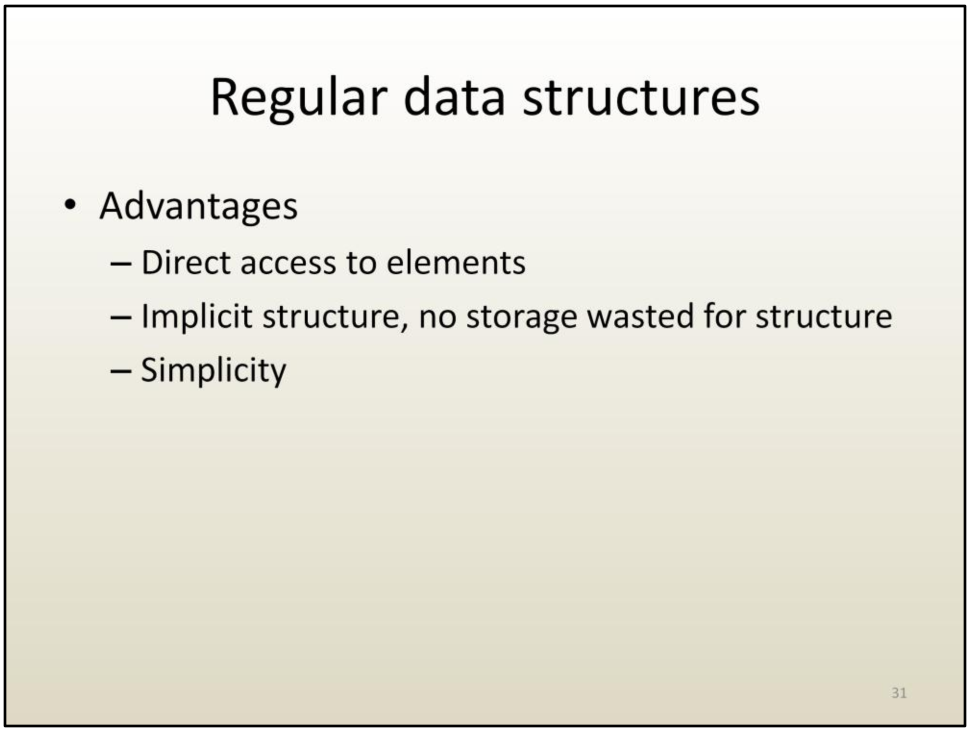
0												*1*											
0						*1*						*0*						*1*					
0			*1*			*0*			*1*			*0*			*1*			*0*			*1*		
0	*1*	*2*	*0*	*1*	*2*	*0*	*1*	*2*	*0*	*1*	*2*	*0*	*1*	*2*	*0*	*1*	*2*	*0*	*1*	*2*	*0*	*1*	*2*
2	3	3	3	2	2	3	4	5	4	2	3	4	5	6	5	3	7	4	3	5	6	4	5

30

Here is another way of thinking about 4-D arrays. Each index adds another subdivision.

For example one can store the number of injured players in a football league with a 4-D array. The league has two conferences A and B. Each conference has two divisions, north and south. Each division has two teams. Each team has 3 sub-teams, offense, defense, and special.

C[1][0][1][2] gives the number of injured players on the special team of the second team in the northern division of the B conference of the league.

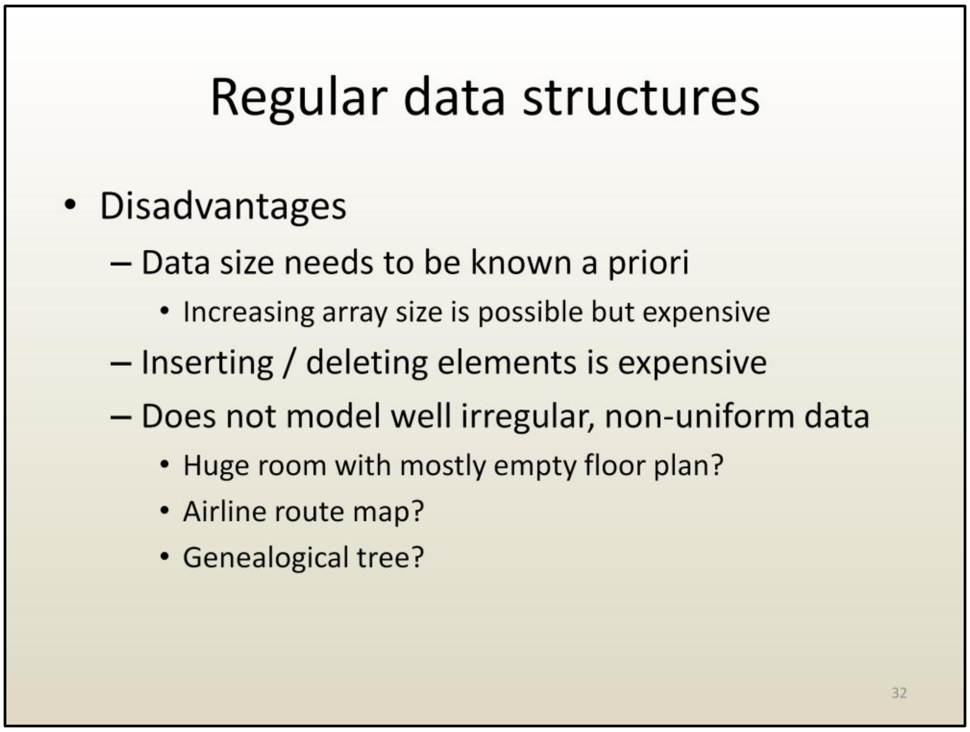


Regular data structures

- Advantages
  - Direct access to elements
  - Implicit structure, no storage wasted for structure
  - Simplicity

31

As mentioned before, arrays are used very frequently. They are simple, do not waste memory, and provide direct access to any element.



Regular data structures

- Disadvantages
  - Data size needs to be known a priori
    - Increasing array size is possible but expensive
  - Inserting / deleting elements is expensive
  - Does not model well irregular, non-uniform data
    - Huge room with mostly empty floor plan?
    - Airline route map?
    - Genealogical tree?

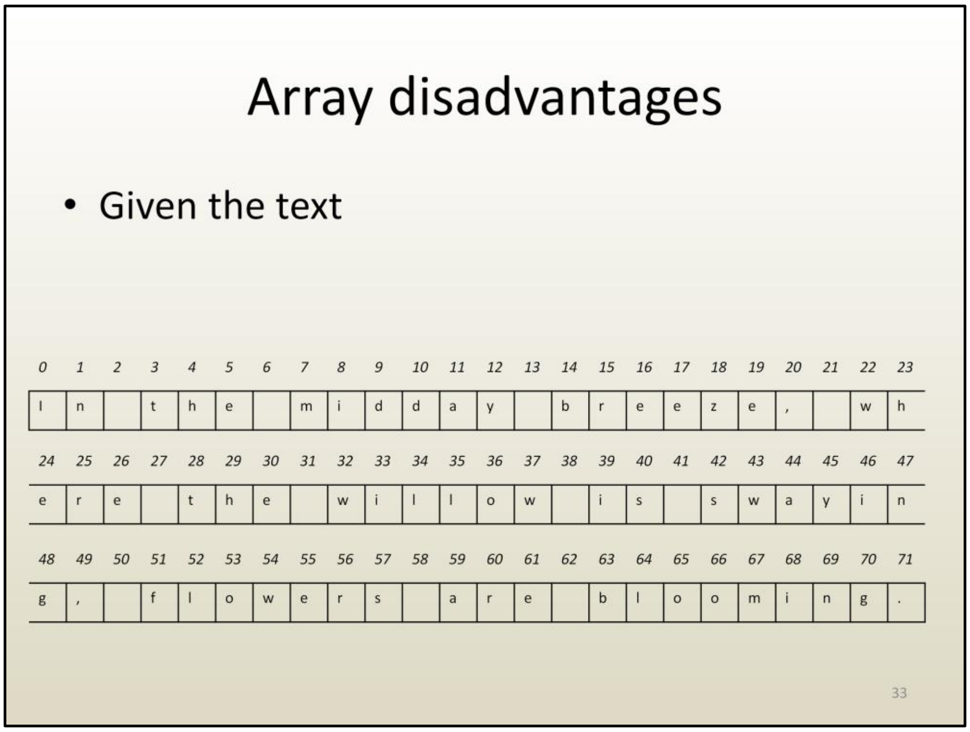
32

However, arrays also have limitations.

One limitation is that you need to know how much data you have as you design the application. For some applications the amount of data changes dynamically and the high water mark cannot be predicted.

Another limitation is that changes like inserting or removing an element are expensive.

A third limitation is that not all data is regular and using arrays to store it is wasteful. For example let’s say we have a room with a 10m x 10m floor plan. Let’s say we want to be able to position a robot with a 1mm accuracy. That mean we need a 2-D array with 10^4x10^4 elements, most of which are empty.



Array disadvantages

- Given the text

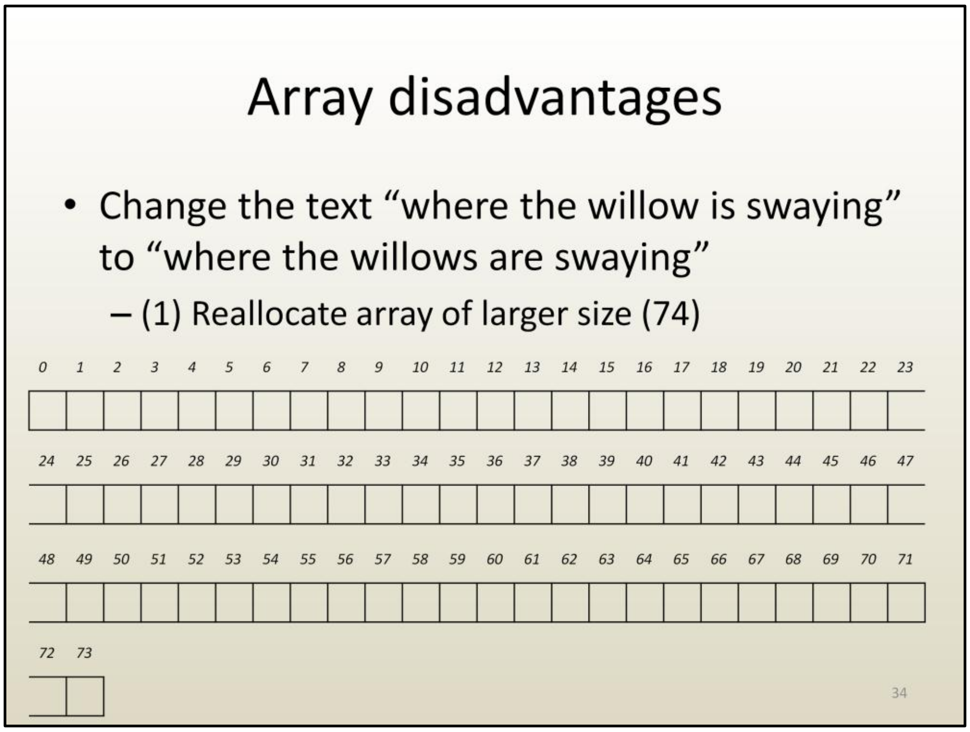
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
I	n		t	h	e		m	i	d	d	a	y		b	r	e	e	z	e	,		w	h

24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
e	r	e		t	h	e		w	i	l	l	o	w		i	s		s	w	a	y	i	n

48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71
g	,		f	l	o	w	e	r	s		a	r	e		b	l	o	o	m	i	n	g	.

33

Here is an illustration of how some simple operations are expensive with arrays.

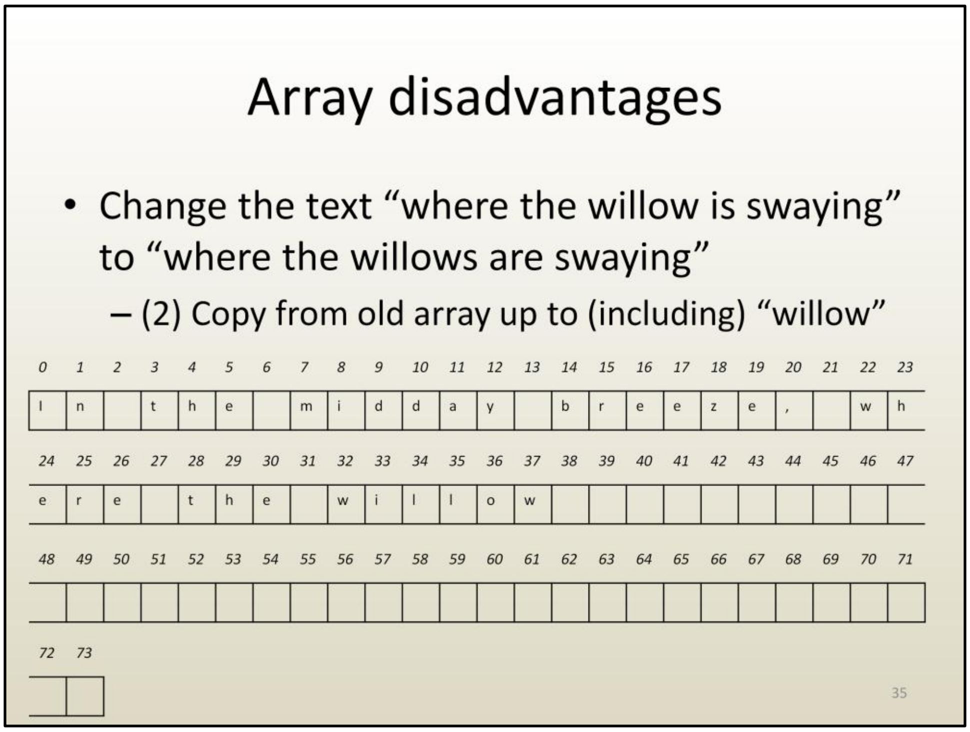


Array disadvantages

- Change the text “where the willow is swaying” to “where the willows are swaying”
  - (1) Reallocate array of larger size (74)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
72 73
34

The new array has to store more elements than the previous array, consequently we need to allocate a new array of larger size. Note that it is not possible to request that the old array be extended, as the memory following the array might be used for something else. It is also the case that the array needs to occupy a contiguous region of memory, for the indexing to work, consequently we can’t find some other space somewhere else where to store the two extra elements.



Array disadvantages

- Change the text “where the willow is swaying” to “where the willows are swaying”
  - (2) Copy from old array up to (including) “willow”

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
I	n		t	h	e		m	i	d	d	a	y		b	r	e	e	z	e	,		w	h

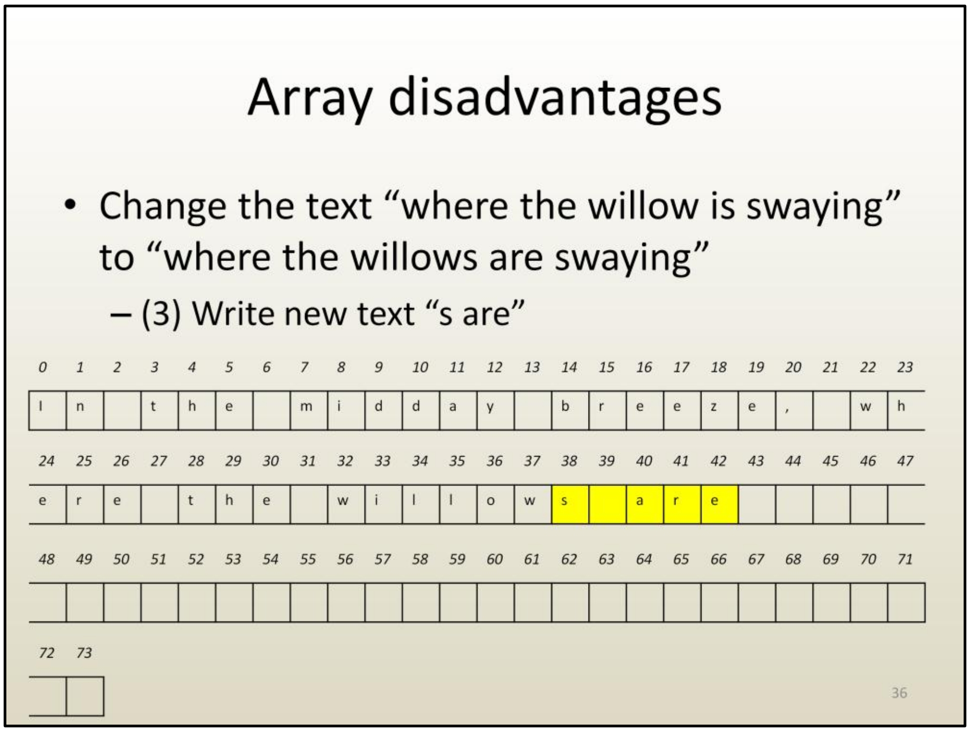
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
e	r	e		t	h	e		w	i	l	l	o	w										

48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71

72	73

35

Then we need to copy all the text before the change, from the old array to the new array.



Array disadvantages

- Change the text “where the willow is swaying” to “where the willows are swaying”
  - (3) Write new text “s are”

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
I	n		t	h	e		m	i	d	d	a	y		b	r	e	e	z	e	,		w	h

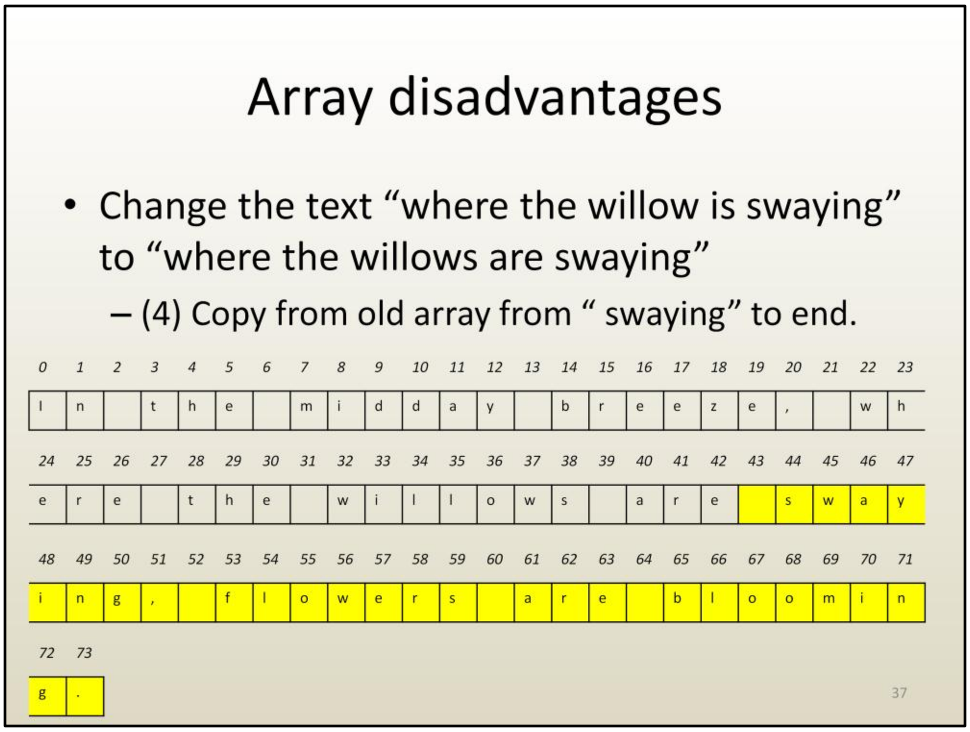
24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
e	r	e		t	h	e		w	i	l	l	o	w	s		a	r	e					

48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71

72	73

36

Then we need to put in the new text.



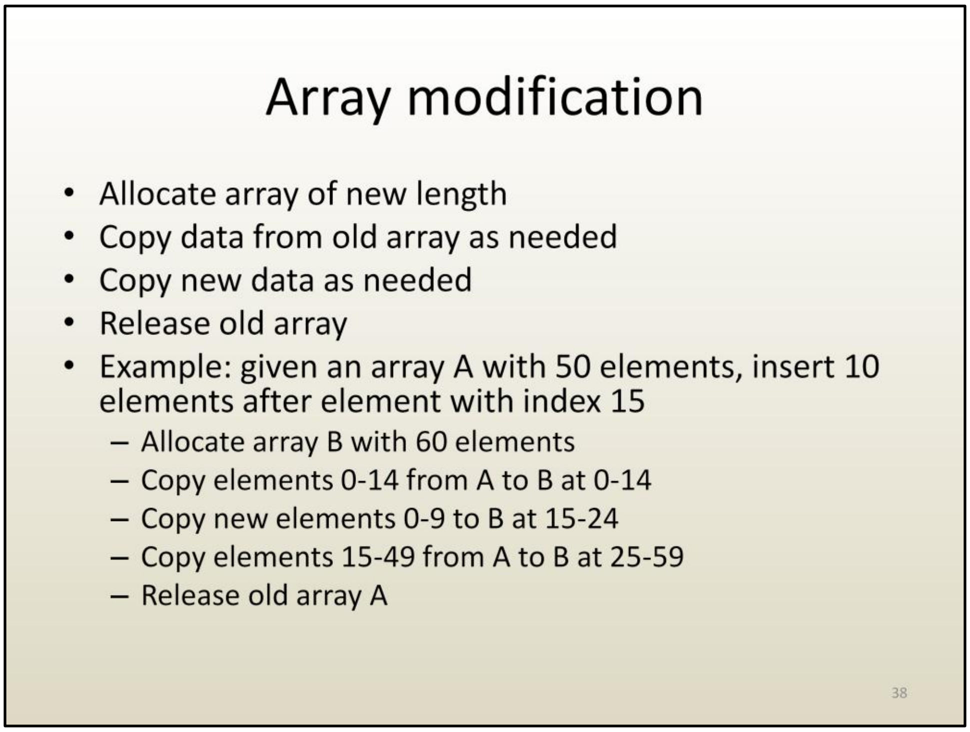
Array disadvantages

- Change the text “where the willow is swaying” to “where the willows are swaying”
  - (4) Copy from old array from “ swaying” to end.

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*
I	n		t	h	e		m	i	d	d	a	y		b	r	e	e	z	e	,		w	h
24	*25*	*26*	*27*	*28*	*29*	*30*	*31*	*32*	*33*	*34*	*35*	*36*	*37*	*38*	*39*	*40*	*41*	*42*	*43*	*44*	*45*	*46*	*47*
e	r	e		t	h	e		w	i	l	l	o	w	s		a	r	e		s	w	a	y
48	*49*	*50*	*51*	*52*	*53*	*54*	*55*	*56*	*57*	*58*	*59*	*60*	*61*	*62*	*63*	*64*	*65*	*66*	*67*	*68*	*69*	*70*	*71*
i	n	g	,		f	l	o	w	e	r	s		a	r	e		b	l	o	o	m	i	n
72	*73*																						
g	.																						

37

And finally we need to copy the old text following the change from the old array to the new array.



Array modification

- Allocate array of new length
- Copy data from old array as needed
- Copy new data as needed
- Release old array
- Example: given an array A with 50 elements, insert 10 elements after element with index 15
  - Allocate array B with 60 elements
  - Copy elements 0-14 from A to B at 0-14
  - Copy new elements 0-9 to B at 15-24
  - Copy elements 15-49 from A to B at 25-59
  - Release old array A

38

Here is a summary of the steps that need to be taken to perform an array modification that implies changing the array size.

Note the last step of releasing the old array—this ensures that that memory goes back to the pool of unused memory and can be used in the future. Not doing that implies a memory leak. A memory leak is the software error when new memory blocks are allocated repeatedly without releasing the old blocks that are not needed anymore. Memory leaks are a frequent and serious problem that causes SW to crash.