ESC101: Introduction to Computing

Pointers

Multi-dimensional Array vs. Multi-level pointer

Are these two equivalent

```
int a[2][3];
```

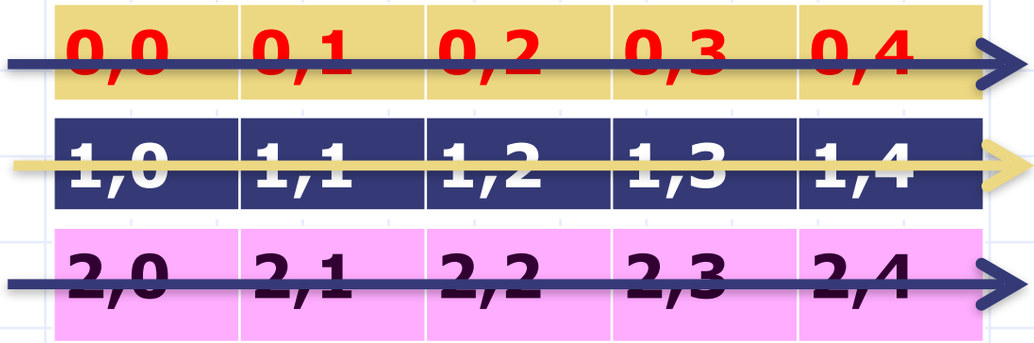
```
int **b;
b = (int**)malloc(2*sizeof(int*));
b[0] = (int*)malloc(3*sizeof(int));
b[1] = (int*)malloc(3*sizeof(int));
```

Row Major Layout

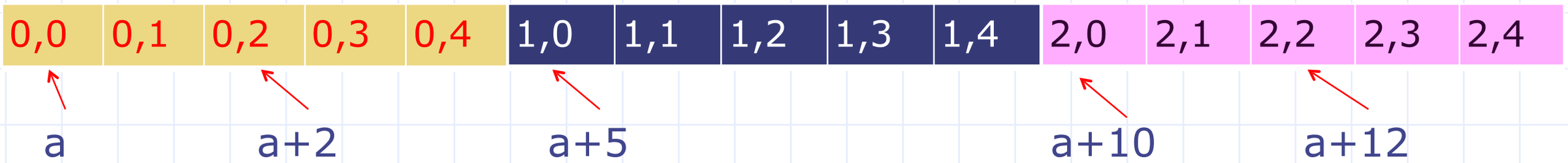
- 2D (or >2D) arrays are "flattened" into 1D to be stored in memory
- In C (and most other languages), arrays are flattened using Row-Major order
  - In case of 2D arrays, knowledge of number of columns is required to figure out where the next row starts.
  - Last n-1 dimensions required for nD arrays

Row Major Layout

mat[3][5]



Layout of mat[3][5] in memory



- **for a 2D array declared as mat[M][N], cell [i][j] is stored in memory at location i*N + j from start of mat.**

Row Major Layout

mat[3][5]

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

Layout of mat[3][5] in memory

0,0	0,1	0,2	0,3	0,4	1,0	1,1	1,2	1,3	1,4	2,0	2,1	2,2	2,3	2,4

mat[0] mat[1] mat[2]

int a[3][3]

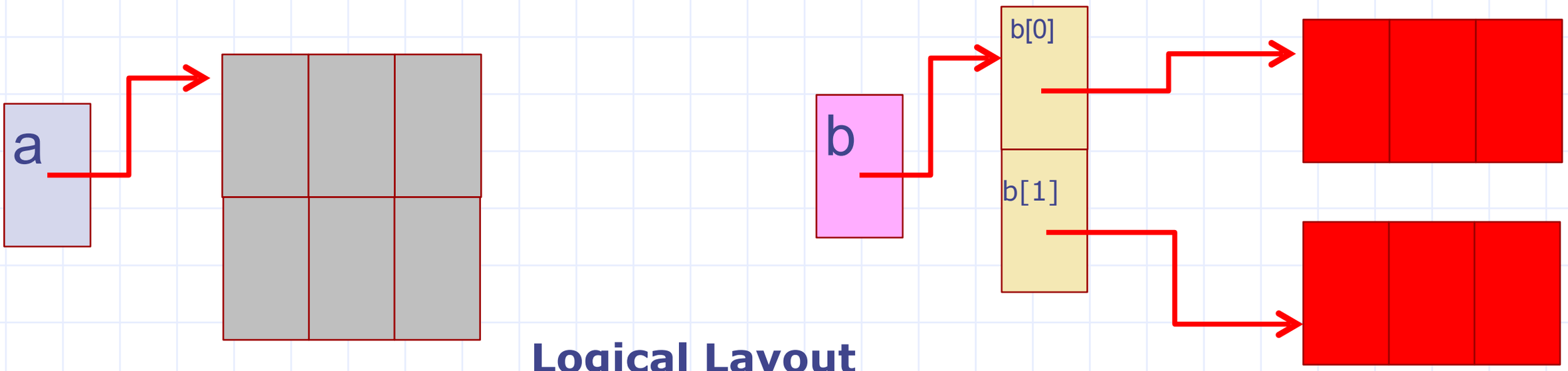
3	9	27
4	16	64
5	25	125

Expression	Value	Expression
*(*a+0)	a[0][0]=3	
*(*a+2)	a[0][2]=27	
*(*a+3)	a[1][0]=4	*(*(a+1)+0)
*(*a+7)	a[2][1]=25	*(*(a+2)+1)

Memory layout

```
int a[2][3];
```

```
int **b;
b = (int**)malloc(2*sizeof(int*));
b[0] = (int*)malloc(3*sizeof(int));
b[1] = (int*)malloc(3*sizeof(int));
```



Logical Layout

Warning:

- **(*b+3) may not point to b[1][0].**
- **(*a+3) points to a[1][0].**

int a[3][3]

3	9	27
4	16	64
5	25	125

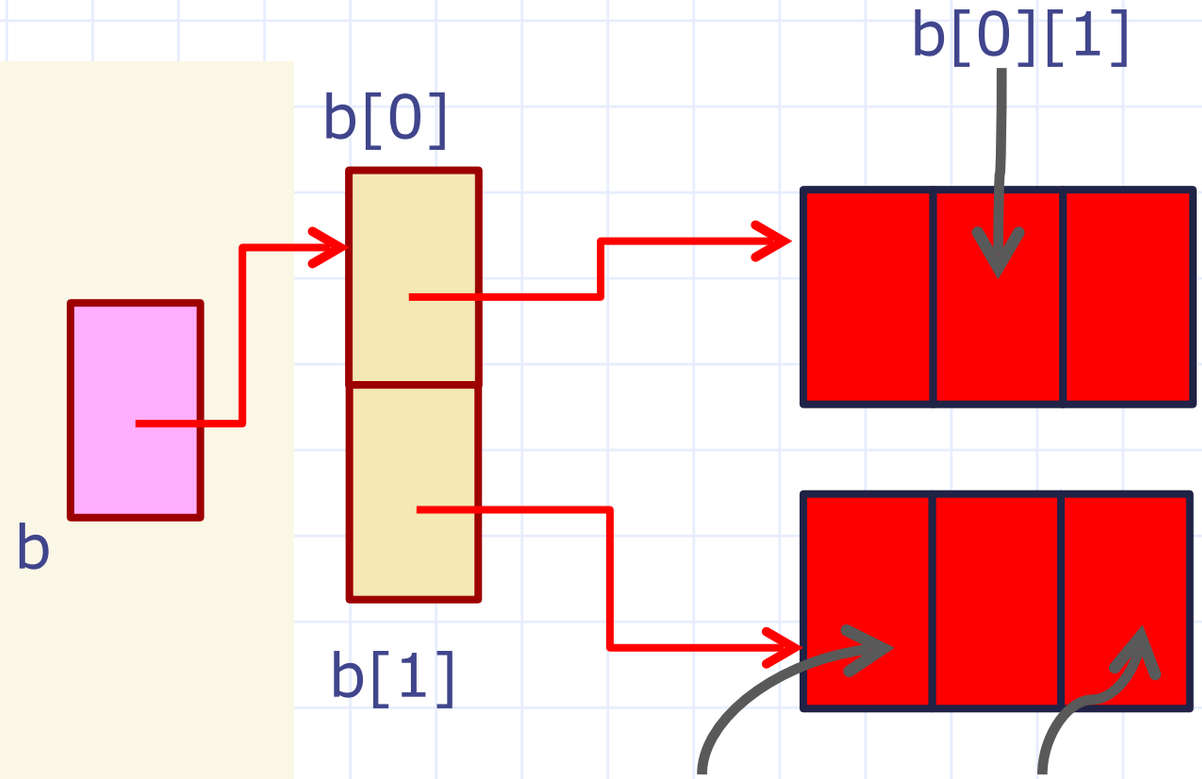
Expression	Value	Expression
*(*a+0)	a[0][0]=3	
*(*a+2)	a[0][2]=27	
*(*a+3)	a[1][0]=4	*(*(a+1)+0)
*(*a+7)	a[2][1]=25	*(*(a+2)+1)

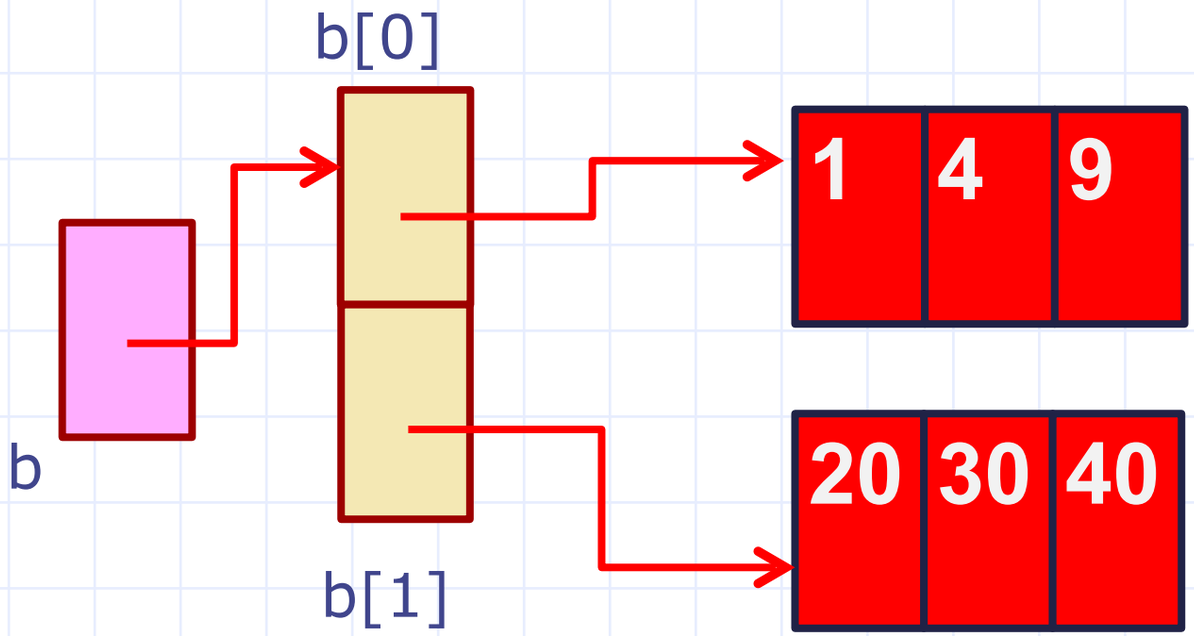
Indexing Elements

How to refer to an element of the array in the language of pointers?

- **b[0][1] is *(*b+1)**
- **b[1][0] is **(b+1)**
- **b[1][2] is *(*(b+1)+2)**

In general, b[i][j] is *(*(b+i)+j)





Expression	Value
**(b+1)	20
*(*b+1)	4
(**b+1)	2
*(*b+2)+2	11
((b+1)+2)+2	42

Pointers vs. Arrays: Indexing

- Matrix style notation A[i][j] is easier for humans to read
- Computers understand pointer style notation *(*(p + i) + j)
  - More efficient in some cases
- Be extremely careful with brackets
  - **(p + i +j) ≠ *(*(p + i) + j) ≠ *(*p + i + j)

```
int a[3][3], i, j, *b, *c;

for (i=0; i<3; i++)
    for (j=0; j<3; j++)
        a[i][j] = pow((i+3),(j+1));

b = *a;
c = *(a+2);
for (i=0; i<3; i++)
    printf("%d ", b[i]);
printf("\n");

for (i=0; i<3; i++)
    printf("%d ", *(c+i));
```

At this point, array **a** is:

3	9	27
4	16	64
5	25	125

What do **b** and **c** point-to here?

b is a pointer to a[0][0]?

c is a pointer to a[2][0]?

OUTPUT

3 9 27

5 25 125

```
int a[3][3], i, j, *b, *c;

for (i=0; i<3; i++)
    for (j=0; j<3; j++)
        a[i][j] = pow((i+3),(j+1));

b = *a;
c = *(a+2) + 1;
for (i=0; i<3; i++)
    printf("%d ", b[i]);
printf("\n");

for (i=0; i<2; i++)
    printf("%d ", *(c+i));
```

At this point, array **a** is:

3	9	27
4	16	64
5	25	125

What do **b** and **c** point-to here?

b is a pointer to a[0][0]?
c is a pointer to a[2][1]?

OUTPUT
3 9 27
25 125

note the change

Array of Pointers vs. Pointer to an Array

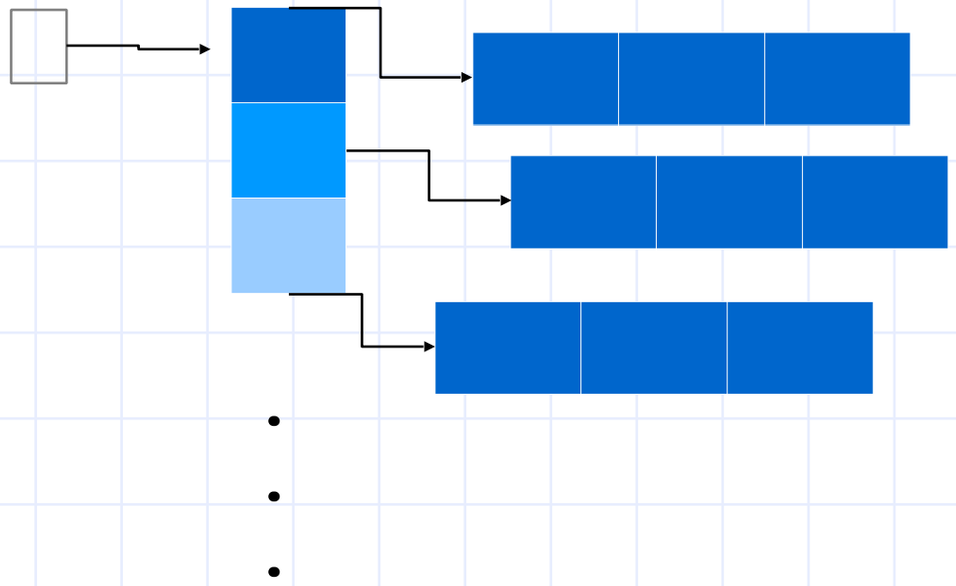
```
int arr[2][3];
```

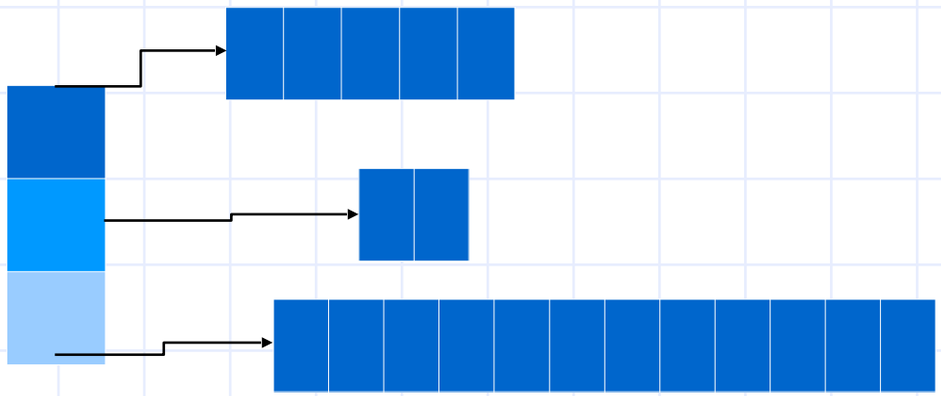
(number of rows fixed,

number of columns fixed)

```
int (*arr)[3];
```

(only the number of columns fixed)





Array of arrays

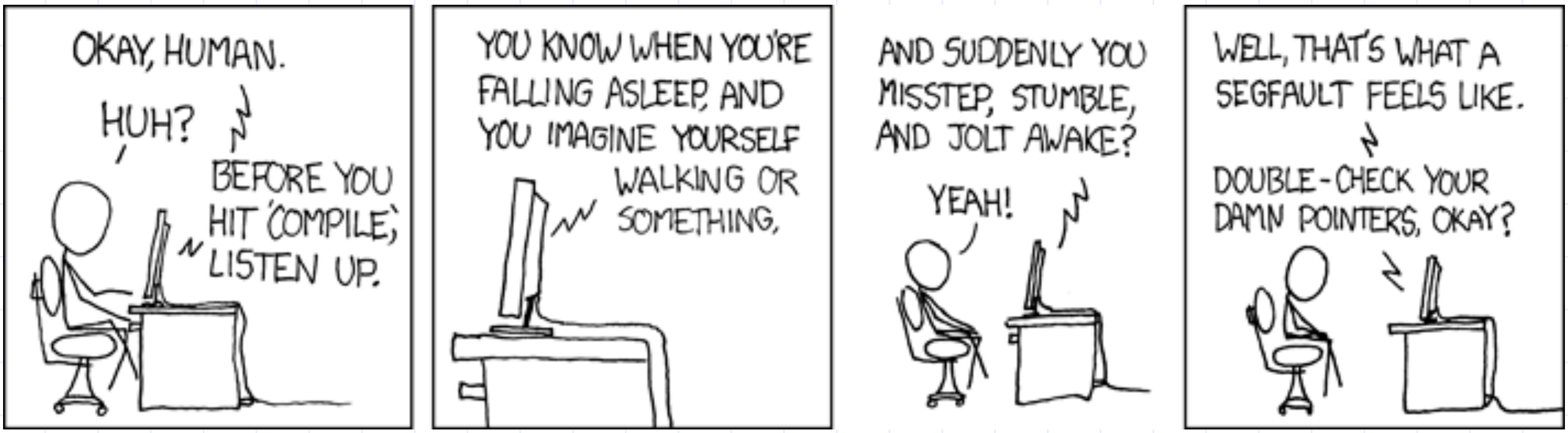
```
int* arr[3];
```

(only the number of rows fixed)

```
int **arr;
```

(general case)

Common Issues and Errors



Source: http://www.xkcd.com/371

Common Issues and Errors

- Forgetting to malloc, forgetting to initialize allocated memory
- Not allocating enough space in malloc (e.g. Allocating 4 characters instead of 5 to store the string “IITK”.)
- Returning pointers to temporaries (called dangling pointers)
- Forgetting to free memory after use (called a memory leak.)
- Freeing the same memory more than once (runtime error), using free-d memory

Memory Leaks

- Consider code:

```
1. int *a;
2. a = (int *)malloc(5*sizeof(int));
3. a = NULL;
```

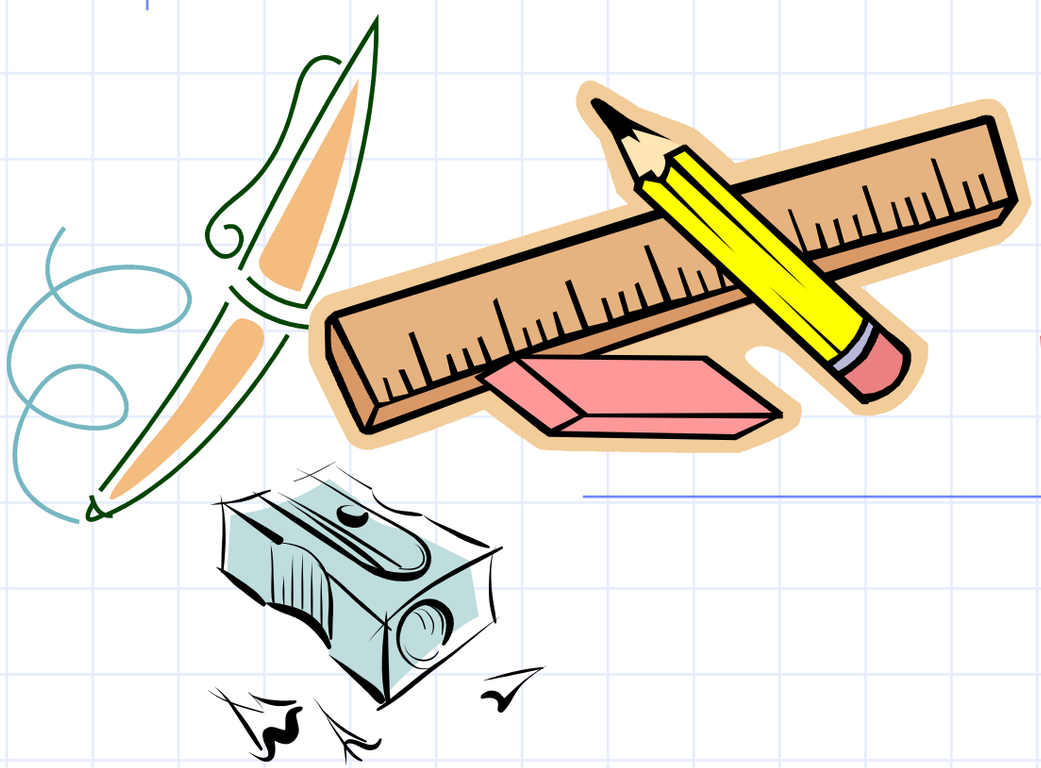
- Memory is allocated to a at line 2.
- However, at line 3, a is reassigned NULL
- No way to refer to allocated memory!
  - We can not even free it, as free-ing requires passing address of allocated block
- This memory is practically lost for the program
  - Ideally, memory should be freed before losing last reference to it

```c
int main(){

void **arr = malloc(3 * sizeof(void *));
arr[0] = strdup("Some string");
arr[1] = (int *) malloc(sizeof(int));
*((int *)(arr[1])) = 10;
arr[2] = malloc(sizeof(double));
*((double *)(arr[2])) = 10.5;

printf( "String: %s\n", (char *)(arr[0]) );
printf( "Integer: %d\n", *((int*)(arr[1])) );
printf( "Double: %f\n", *((double *)(arr[2])) );

return 0;
}
```

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main(){
void **arr = malloc(3 * sizeof(void *));
arr[0] = strdup("Some string"); /* strdup returns a pointer */
arr[1] = (int *) malloc(sizeof(int));
*((int *)(arr[1])) = 10;
arr[2] = malloc(sizeof(double));
*((double *)(arr[2])) = 10.5;
/* print the values */
printf( "String: %s\n", (char *)(arr[0]) );
printf( "Integer: %d\n", *((int*)(arr[1])) );
printf( "Double: %f\n", *((double *)(arr[2])) );

/* We have to free ALL the values */
for (int i = 0; i < 3; i++)
  free(arr[i]);
/* free the array */
free(arr);
return 0;
}
```

ESC101: Introduction to Computing

Structures



Motivation

- Till now, we have used data types int, float, char, arrays (1D, 2D,...) and pointers.
- What if we want to define our own data types based on these?
- A geometry package – we want to define a point as having an x coordinate, and a y coordinate.
- Student data – Name and Roll Number
  - array of size 2?
  - two variables:
    - int point_x , point_y;
    - char *name; int roll_num;

Motivation

- A geometry package – we want to define a point as having an x coordinate, and a y coordinate.
- Student data – Name and Roll Number
  - array of size 2? (Can not mix TYPES)
  - two variables:
    - int point_x , point_y;
    - char *name; int roll_num;
  - There is no way to indicate that they are part of the same point!
  - requires a disciplined use of variable names
- Is there any better way ?

Motivation: Practical Example

- Write a program to manage customer accounts for a large bank.
- Customer information as well as account information, for e.g.:
  - Account Number
  - Account Type
  - Customer Name
  - Customer Address
  - Signature scan

int

int (enum – not covered)

char*/char[]

char*/char[]

bitmap image
(2-D array of bits)

Example: Enumerated types

- Account type via Enumerated Types.
- Enumerated type allows us to create our own symbolic name for a list of related ideas.
  - The key word for an enumerated type is **enum**.
- We could create an enumerated type to represent various "account types", by using the following C statement:

```
enum act_Type { savings, current, fixDeposit, minor  };
```

Example: Enumerated types

- Account type via Enumerated Types.

```
enum act_Type { savings, current, fixDeposit, minor  };

enum act_Type a;

a = current;

if (a==savings)
     printf("Savings account\n");

if (a==current)
     printf("Current account\n");
```

Enumerated types provide a symbol to represent one state out of several constant states.

Structures

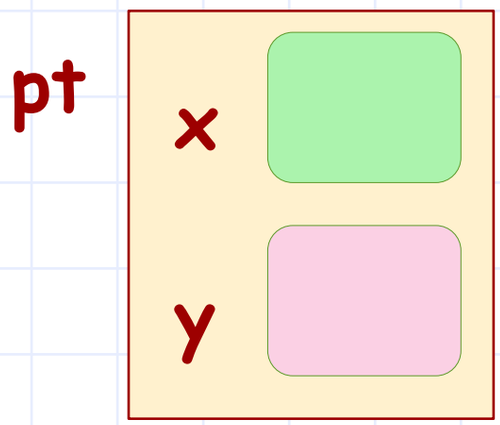
- A structure is a collection, of **variables**, under a common name.
- The variables can be of **different** types (including arrays, pointers or structures themselves!).
- Structure variables are called fields.

```
struct point {
        int x;
        int y;
};
```

```
struct point pt;
```

This defines a structure called point containing two integer variables (fields), called x and y.

struct point pt defines a variable pt to be of type **struct point**.



memory depiction of pt

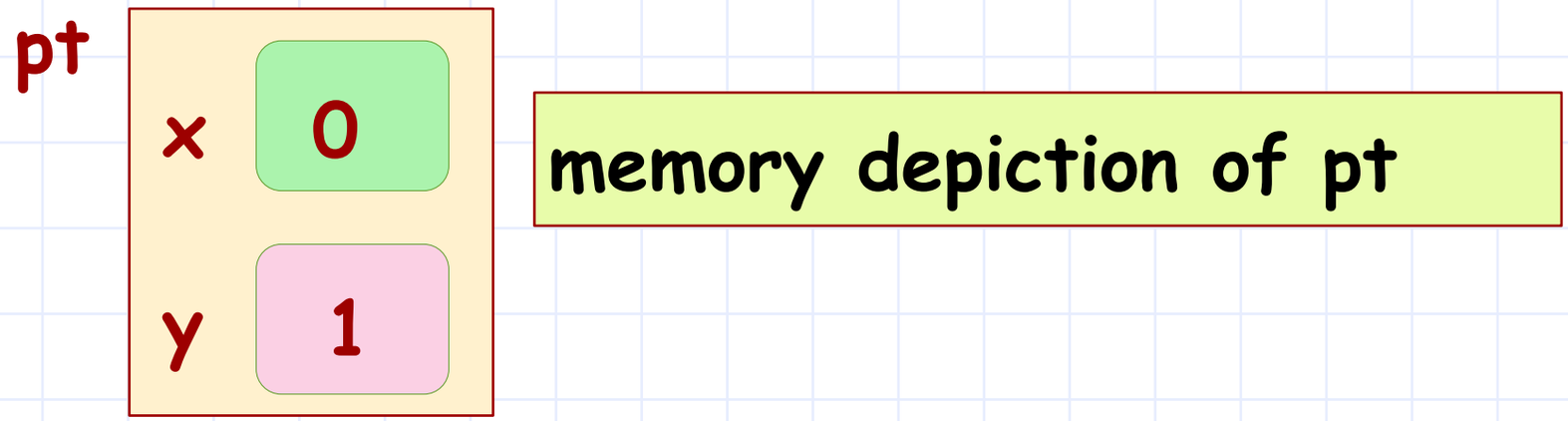
Structures

- The **x** field of **pt** is accessed as **pt.x**.
- Field **pt.x** is an **int** and can be used as any other **int**.
- Similarly the **y** field of **pt** is accessed as **pt.y**

```
struct point {
        int x;
        int y;
};
```

```
struct point pt;
```

```
pt.x = 0;
pt.y = 1;
```



Structures

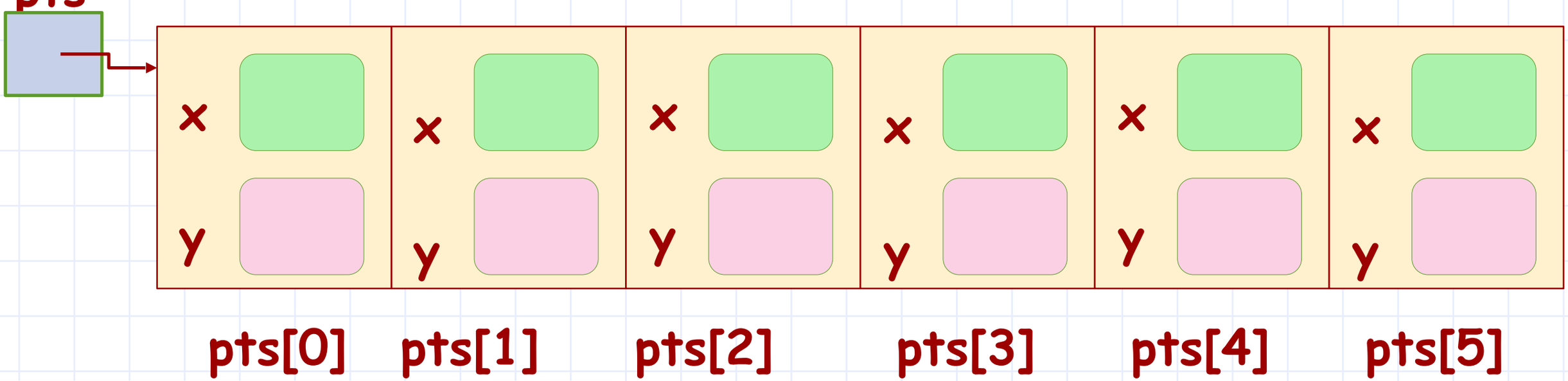
```
struct point {
        int x; int y;
}
```

struct point is a type. It can be used just like int, char etc..

For now, define structs in the beginning of the file, after #include.

```
struct point pt1,pt2;
struct point pts[6];
```

We can define array of struct point also.




```
int i;
for (i=0; i < 6; i=i+1) {
        pts[i].x = i;
        pts[i].y = i;
}
```

Read pts[i].x as (pts[i]).x
The . and [] operators have same precedence. Associativity: left-right.

Structures

```
struct point {
        int x; int y;
};
struct point pts[6];
int i;
for (i=0; i < 6; i=i+1) {
        pts[i].x = i;
        pts[i].y = i;
}
```

State of memory after the code executes.