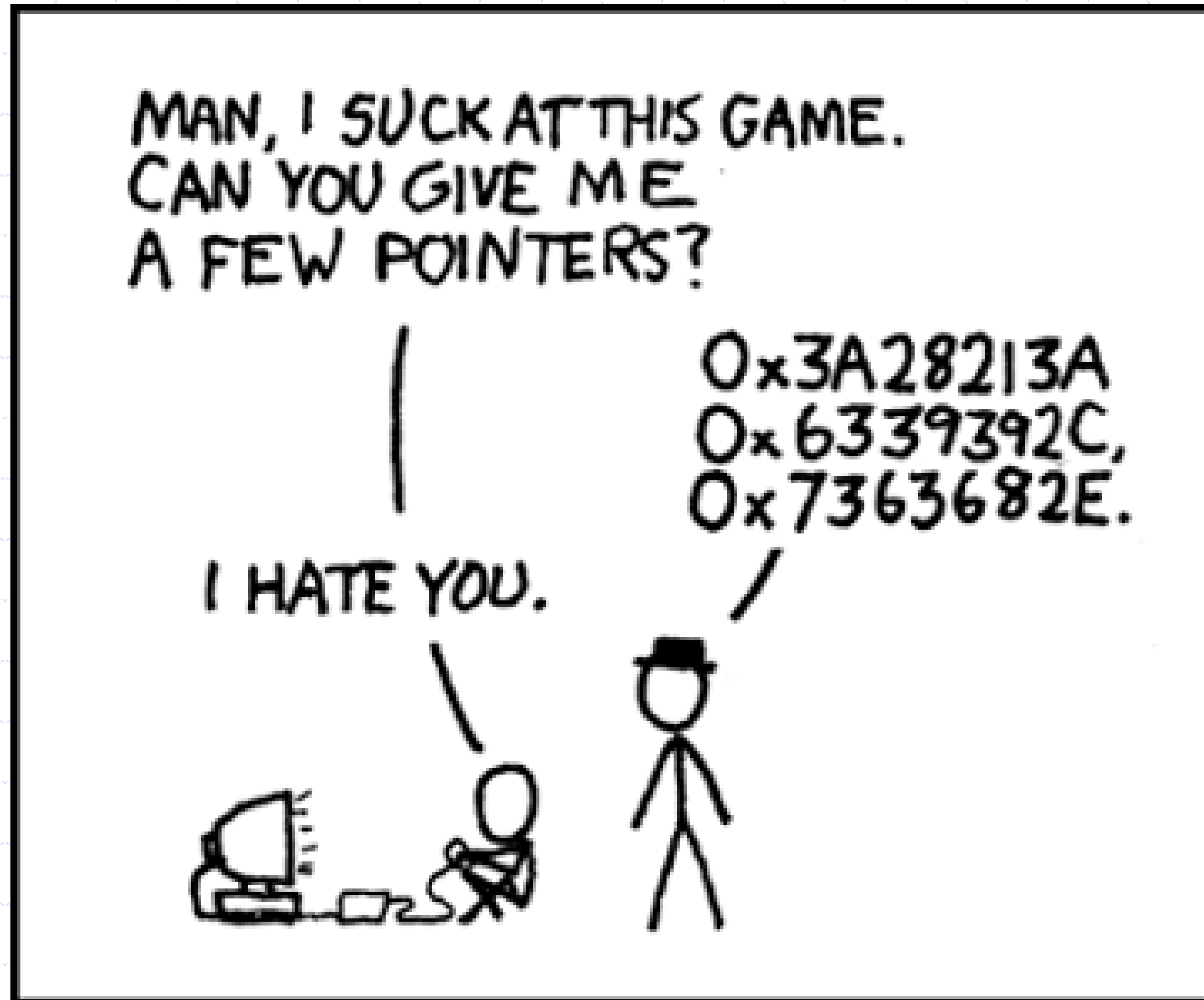
Returning Pointers



Source: http://www.xkcd.com/138

Example Function that Returns Pointer

```
char *strdup(const char *s);
```

- strdup creates a copy of the string (char array) passed as arguments
  - copy is created in dynamically allocated memory block of sufficient size
- returns a pointer to the copy created
- C does not allow returning an Array of any type from a function
  - But we can use a pointer to simulate return of an array (or multiple values of same type)

Returning Pointer: Beware

```
#include<stdio.h>
int *fun();
int main() {
  printf("%d",*fun());
}

int *fun() {
  int *p, i;
  p = &i;
  i = 10;
  return p;
}
```

OUTPUT

```
#include<stdio.h>
int *fun();
int main() {
  printf("%d",*fun());
}

int *fun() {
  int *p;
  p = (int*)malloc(sizeof(int));
  *p = 10;
  return p;
}
```

OUTPUT: 10

Returning Pointer: Beware

- The function stack (except for the return value) is gone once the function completes its execution.
  - All addresses of local variables and formal arguments become invalid
  - available for "reuse"
- But the heap memory, once allocated, remains until it is explicitly "freed"
  - even beyond the function that allocated it.
- addresses of static and global variables remain valid throughout the program.

An Intuition

- Think of executing a function as writing on a classroom blackboard.
- Once the function finishes execution (the class is over), everything on the blackboard is erased.
- What if we want to retain a message, after class is over?
- Solution could be to post essential information on a "notice board", which is globally accessible to all classrooms.
- The blackboard of a class is like the stack (possibly erased/overwritten in the next class), and the notice board is like the heap.

Class Quiz

- The following program illustrates the difference between ***int *ptr[2]*** and ***int (*ptr)[2]*** .

```
#include<stdio.h>
int main() {
        int a[] = {1,2,3};
        int (*ptr)[2] = &a;

        printf("%d\n", (*ptr)[0]);
        printf("%d\n", (*ptr)[1]);

        (*ptr)[0] = -1;
        printf("%d\n", a[0]);
        return 0;
}
```

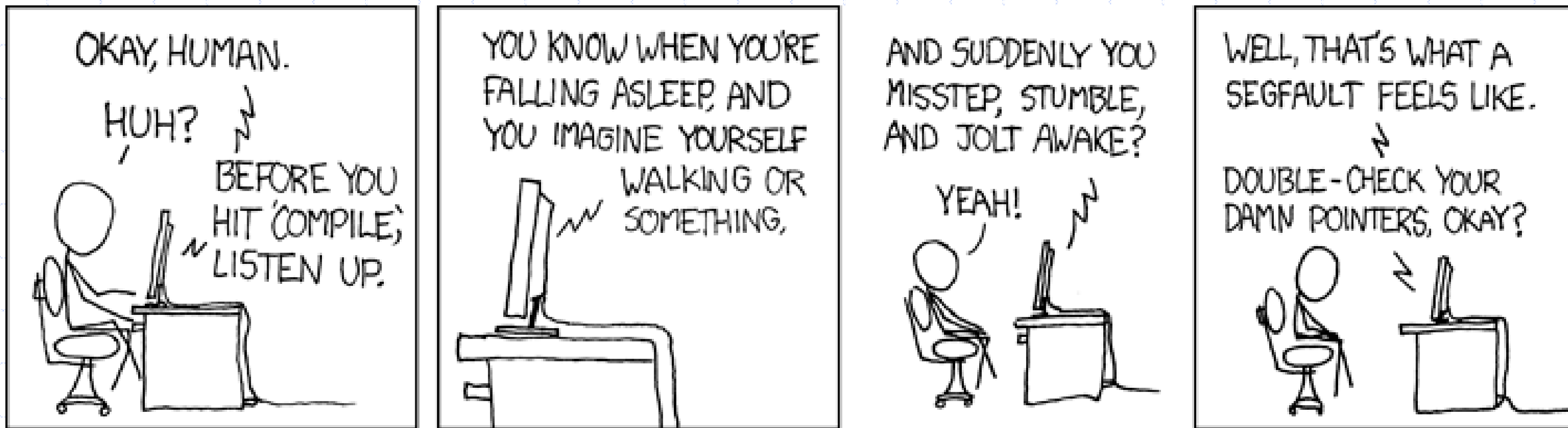
An equivalent assignment is:

```
int (*ptr)[2];
ptr = &a;
```

OUTPUT:

```
1
2
-1
```

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 (Leaked)
  - Ideally, memory should be freed before losing last reference to it

Multi-dimensional Array vs. Multi-level pointer

- Are these two equivalent?

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

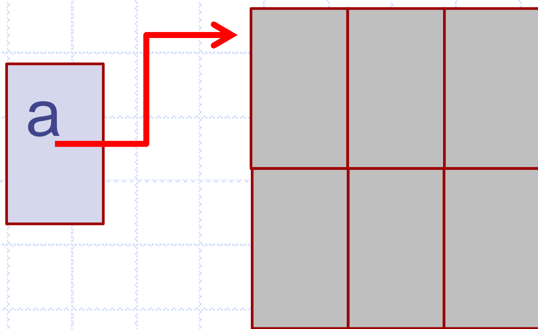
```
int **b;
b = (int**)malloc(2*sizeof(int*));
b[0] = (int*)malloc(3*sizeof(int));
b[1] = (int*)malloc(3*sizeof(int));
```

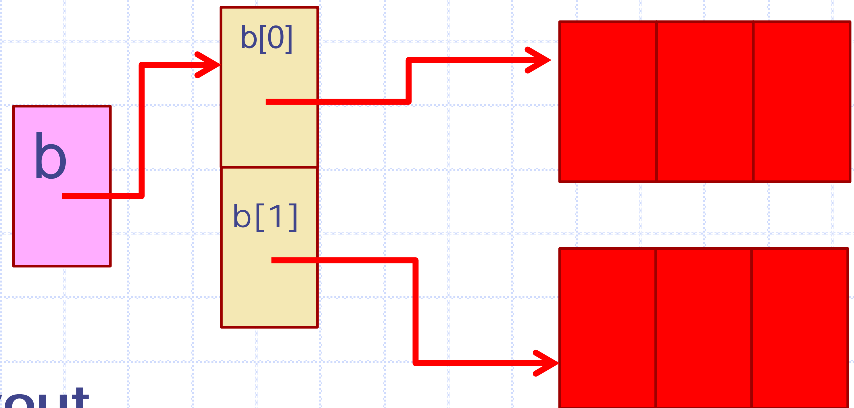
  - Both a and b can hold 6 integer in a 2x3 grid like structure.
  - In case of a all 6 cells are consecutively allocated. For b, we have 2 blocks of 3 consecutive cells each.

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:

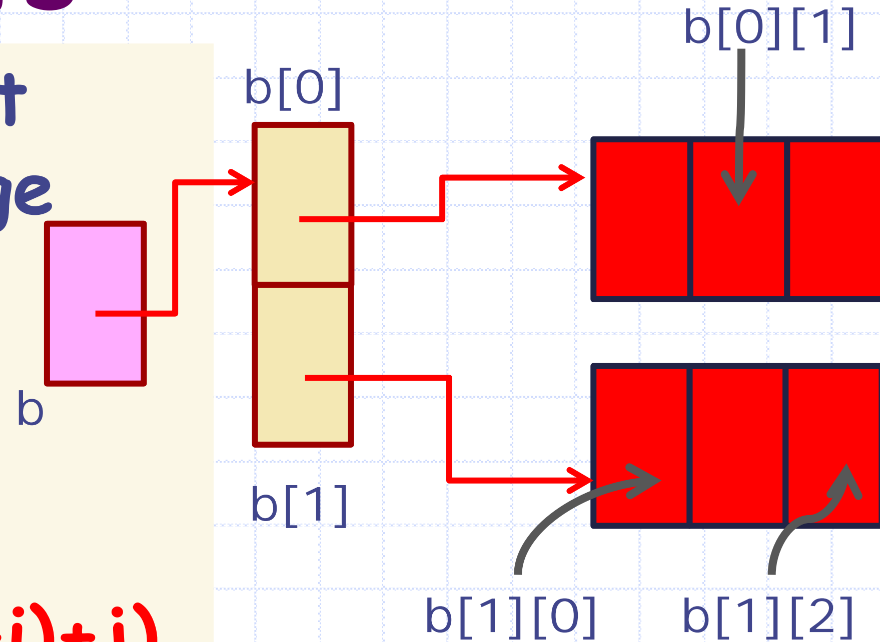
- (*b+3) **may not** point to b[1][0].
- (*a+3) points to a[1][0].

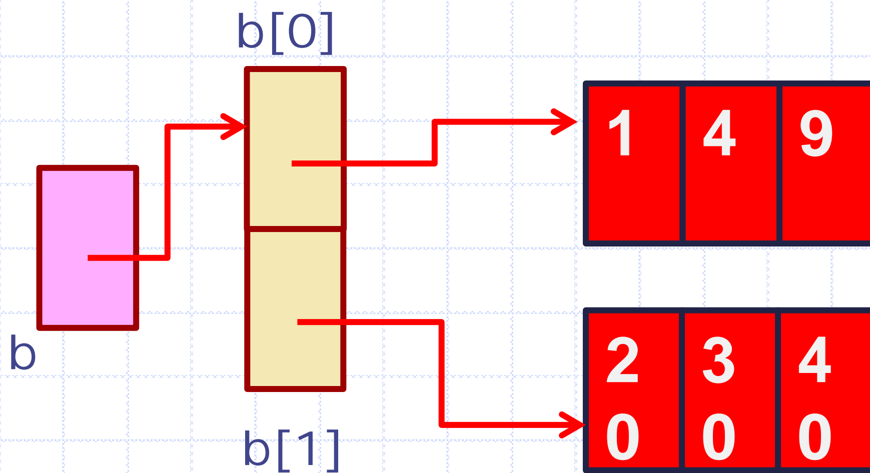
Indexing Elements

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

```c
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

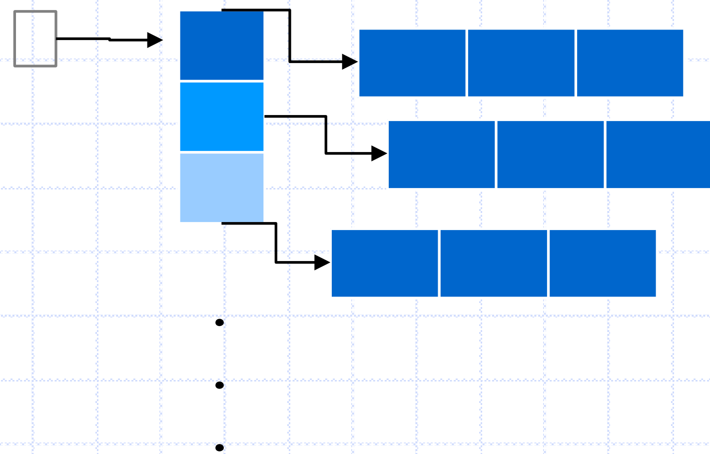
Array of Pointers vs. Pointer to an Array

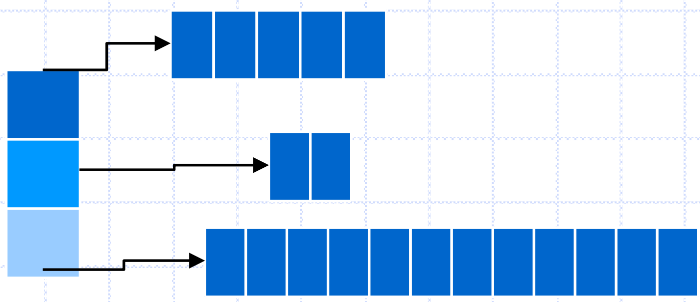
```
int arr[2][3];
```

(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)

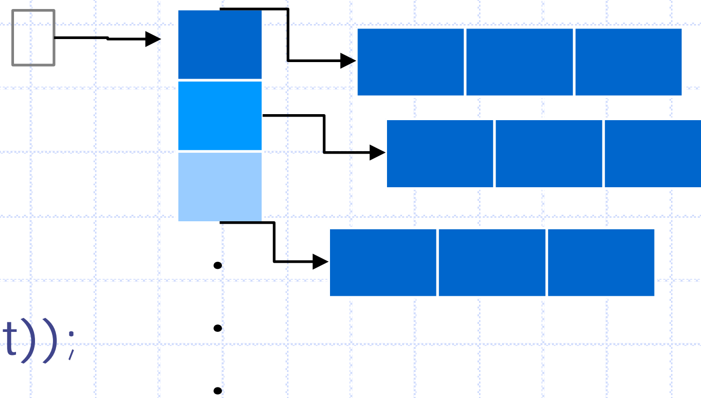
```
int **arr;
```

(general case)

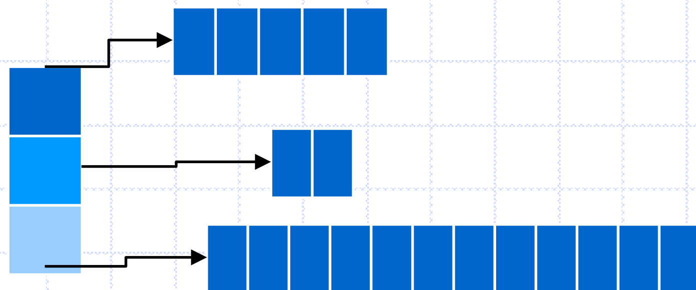
Variants of malloc – Advanced Types

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

(only the number of columns fixed)



```
arr = (int (*)[3]) malloc(n*3*sizeof(int));
                    or
arr = (int (*)[3]) malloc(n*sizeof(int[3]));
```



Array of arrays

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

(only the number of rows fixed)

```
arr = (int *[3]) malloc(n*sizeof(int[3]));
```

Only when you want to use n columns in each row.

```
int **a, *b[2], (*c)[3], d[2][3];

//c = d;  /* Fine, matches the column size */
//c = b;  /* Warning: incompatible pointer type; different sizes! */

printf("sizeof(a) = %3d, sizeof(*a) = %3d, sizeof(**a) = %3d\n",
        sizeof(a), sizeof(*a), sizeof(**a) );
```

```
sizeof(a) =   8, sizeof(*a) =   8, sizeof(**a) =   4
```

```
printf("sizeof(b) = %3d, sizeof(*b) = %3d, sizeof(**b) = %3d\n",
        sizeof(b), sizeof(*b), sizeof(**b) );
```

```
sizeof(b) =  16, sizeof(*b) =   8, sizeof(**b) =   4
```

```
printf("sizeof(c) = %3d, sizeof(*c) = %3d, sizeof(**c) = %3d\n",
        sizeof(c), sizeof(*c), sizeof(**c) );
```

```
sizeof(c) =   8, sizeof(*c) =  12, sizeof(**c) =   4
```

```
printf("sizeof(d) = %3d, sizeof(*d) = %3d, sizeof(**d) = %3d\n",
        sizeof(d), sizeof(*d), sizeof(**d) );
```

```
sizeof(d) =  24, sizeof(*d) =  12, sizeof(**d) =   4
```