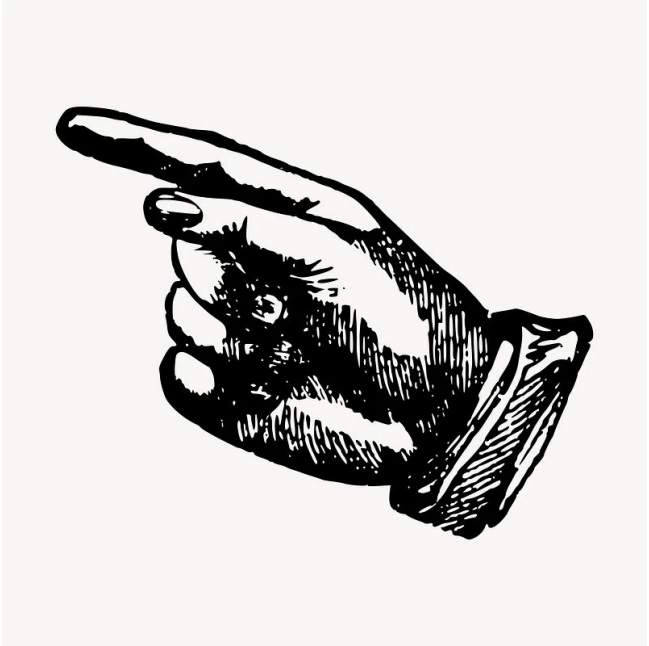
Recitation 2

Computer Architecture (section 1)

Pointer Terminology

- A pointer is a variable that holds a memory address.
  - This address can point to anything - but is typically the address of another variable.



Pointers

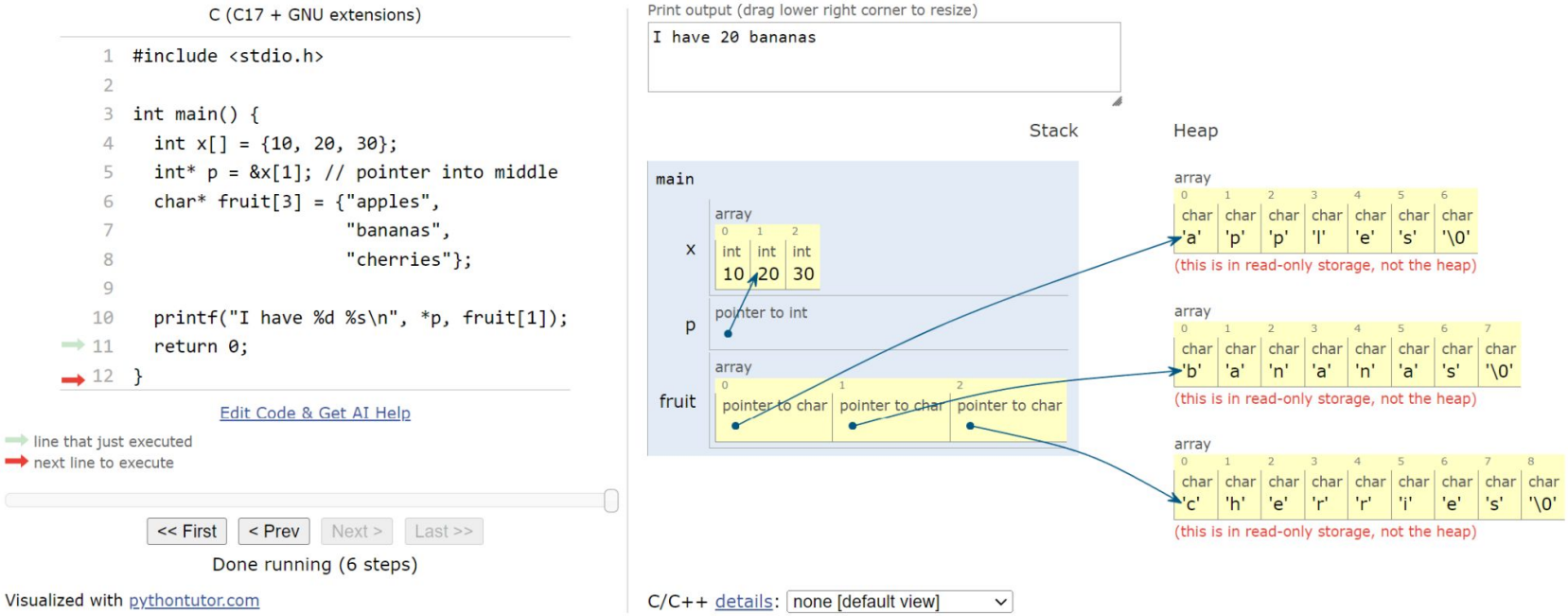
- Lets observe the address of variables on our stack.
- Bonus: What do you observe if you run sequential executions of the program?
  - ASLR: Try playing with these 2 commands on a local machine:
    - echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
    - echo 2 | sudo tee /proc/sys/kernel/randomize_va_space

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int a = 123;
    int b = 456;

    printf("a: %p\n",&a);
    printf("b: %p\n",&b);
    return EXIT_SUCCESS;
}
```

Pointers are difficult

pythontutor.com: Use this online tool to visually debug your code.



Dynamic Memory Allocation

- Request memory at *runtime.*
- For this we use `malloc()`
  - Does NOT initialize elements to 0.
  - Don't forget to `free()`.

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int* an_array = malloc(sizeof(int)*100);
    if(an_array == NULL) return EXIT_FAILURE;
    an_array[20] = 300;
    printf("%d %d\n",an_array[20],an_array[21]);
    free(an_array);
    return EXIT_SUCCESS;
}
```

Complications of Dynamic Memory Allocation

- Memory leakage
  - The condition caused by a program that fails to correctly free allocated memory.
- The definition *really* should be defined within the context of a program.

```
void do_work(void)
{
    int* an_array = malloc(sizeof(int)*100);
    if(an_array == NULL) abort();
    an_array[20] = 300;
    return;
}
```

Sieve of Eratosthenes for Large *n*

```
long long sum_primes(long long limit){
   long long sum = 0;
   bool* marks = malloc(sizeof(bool)*(limit+1));
   if(marks == NULL)
   {
      perror("Malloc failed!");
      abort();
   }
   for(long long i=0; i < limit+1; i++){
      marks[i] = false;
   }

   for(long long i = 2; i <= limit; i++){

      if(!marks[i]){
         sum+=i;

         for(long long m=2*i; m <= limit; m += i)
         {
            marks[m] = true;
         }
      }
   }
   free(marks);
   return sum;
}
```

Debugging with GDB

- ctrl+x + ctrl+a
  - TUI mode
- **`run`**
  - Run program until breakpoint
- **`start`**
  - Run a program until main
- **`next`**
  - Execute one line of source code (steps over functions)
- **`step`**
  - Execute one line of source code (steps into functions)
- **`finish`**
  - Finish execution of current function
- **`continue`**
  - Execute program until a breakpoint
- **`bt`**
  - Print backtrace
- **`b`** (line number,function name)
  - Breakpoint
- **`print`** (/x for hex, \s for string)
  - Print values. Can also use **`display`**
- **`x/`**(format (i.e 30g)) <addr>
  - Print list of memory address values
- **`set`**
  - Set variable or register

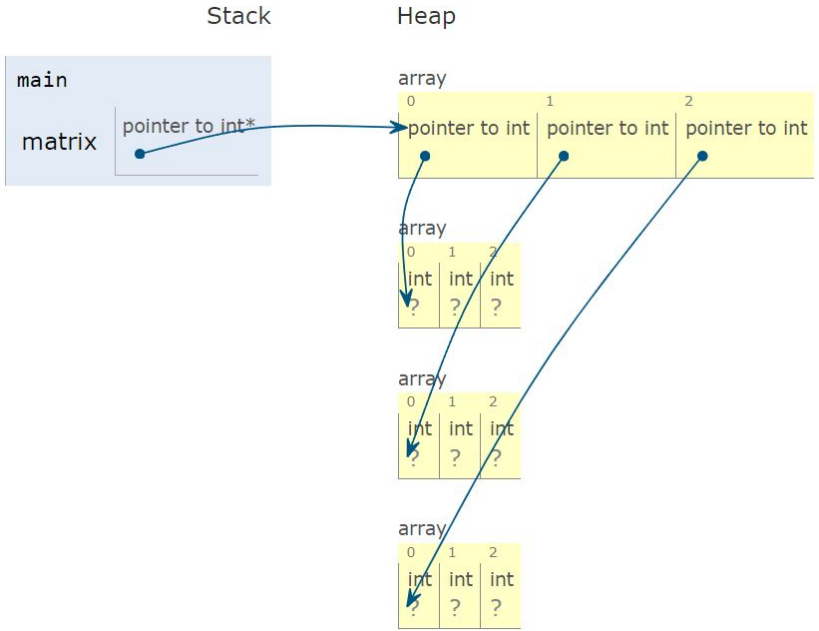
Format	
a	Pointer.
c	Read as integer, print as character.
d	Integer, signed decimal.
f	Floating point number.
o	Integer, print as octal.
s	Try to treat as C string.
t	Integer, print as binary (*t* = „two“).
u	Integer, unsigned decimal.
x	Integer, print as hexadecimal.

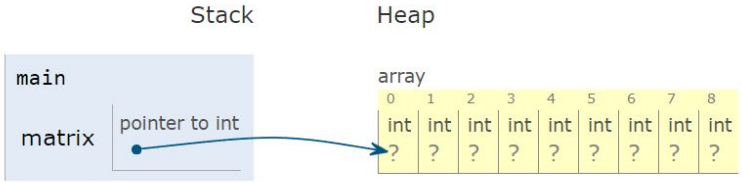
2 Flavors of Matrix Allocation

```
int** one_way(size_t rows, size_t cols)
{
    int** matrix = malloc(sizeof(int*)*rows);
    if(matrix == NULL) abort();
    for(size_t i = 0; i < rows; i++)
    {
        matrix[i] = malloc(sizeof(int)*cols);
        if(matrix[i] == NULL) abort();
    }
    return matrix;
}
```

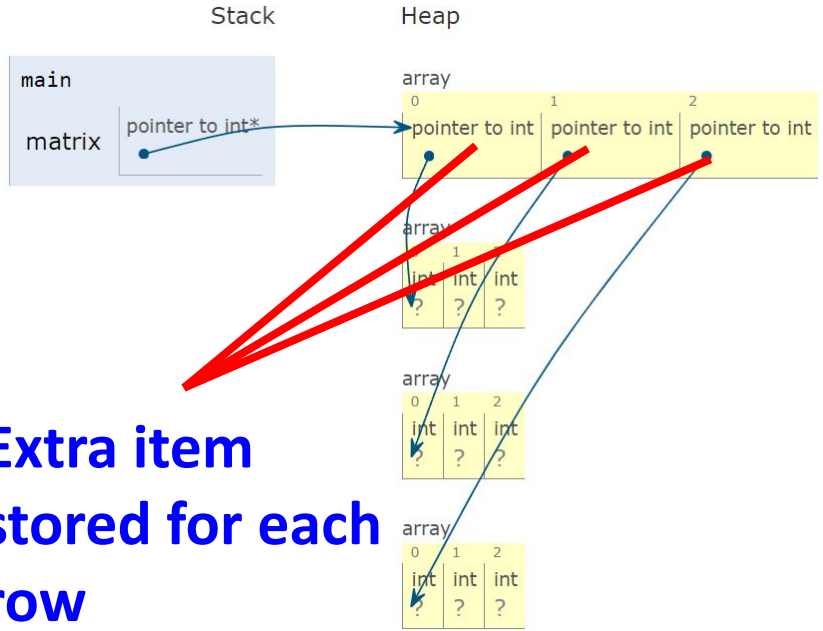
```
int* another_way(size_t rows, size_t cols)
{
    int* matrix = malloc(sizeof(int)*rows*cols);
    if(matrix == NULL) abort();
    return matrix; //index with matrix[i*cols +
j]
}
```

2 Flavors of Matrix Allocation

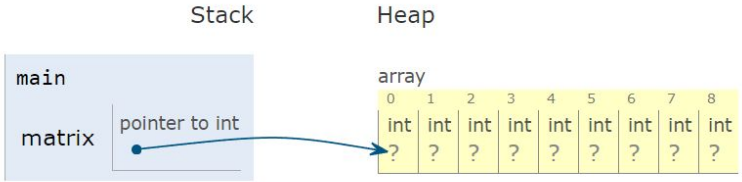




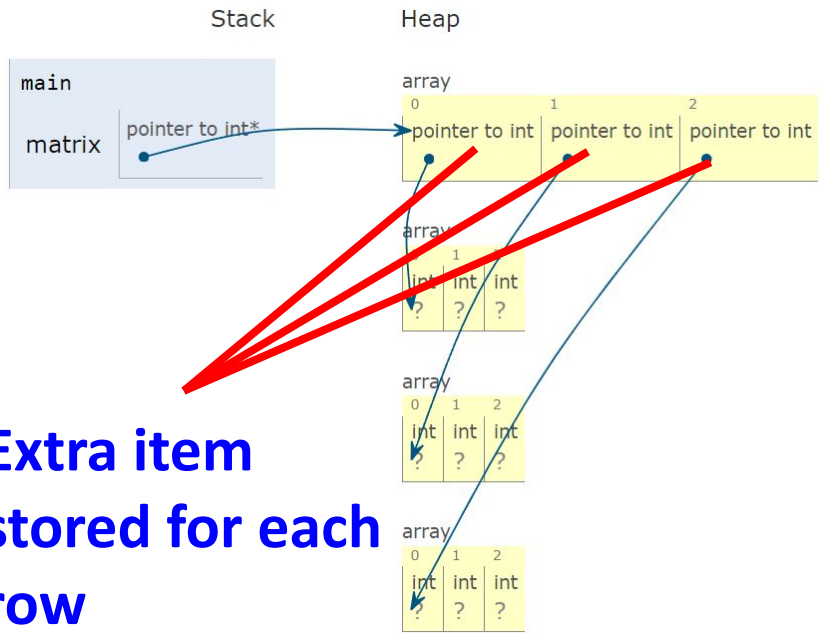
2 Flavors of Matrix Allocation



Extra item stored for each row

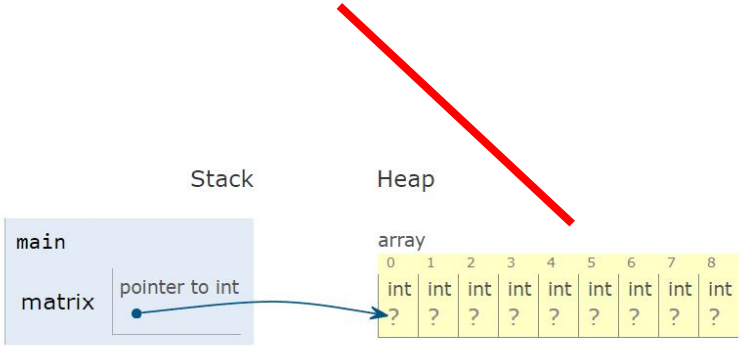


2 Flavors of Matrix Allocation

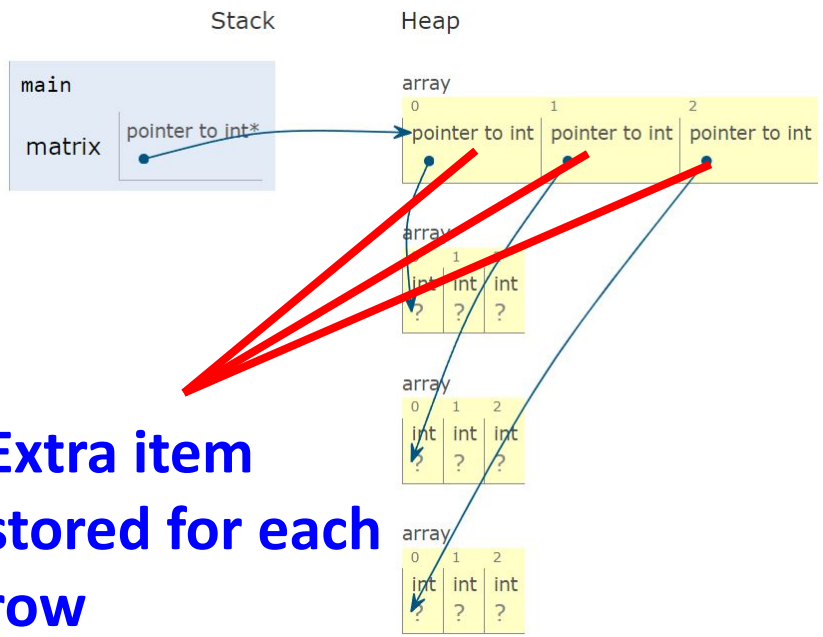


Extra item stored for each row

Must be contiguous

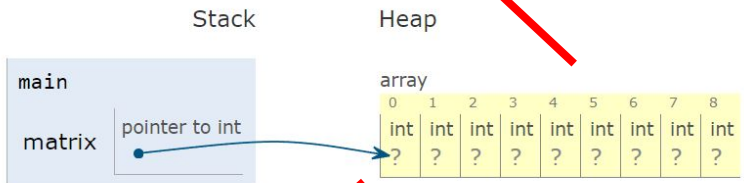


2 Flavors of Matrix Allocation



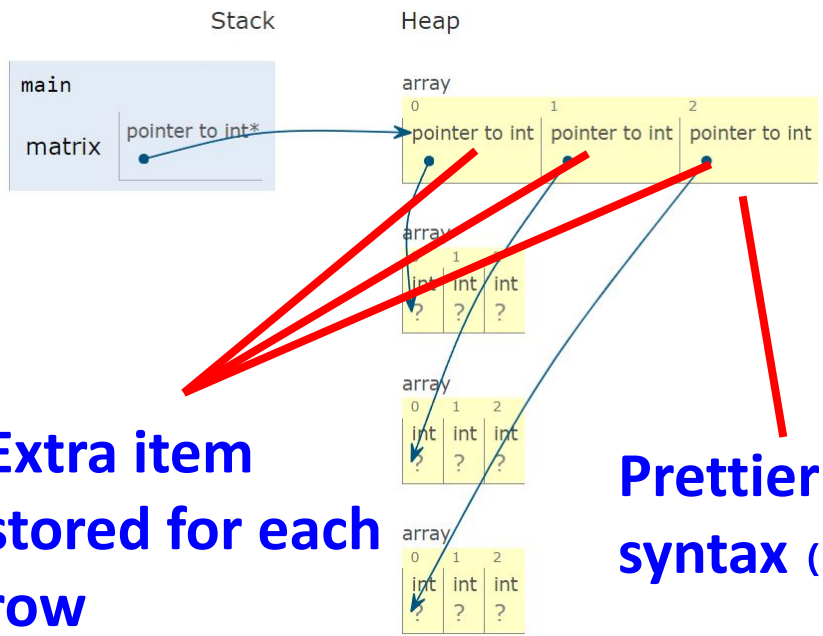
Extra item stored for each row

Must be contiguous



Easy to free

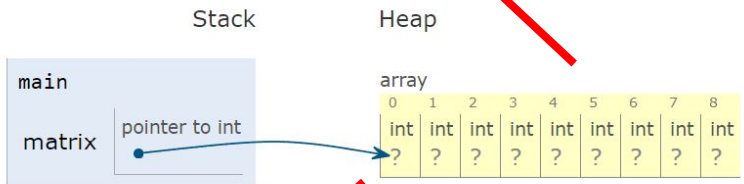
2 Flavors of Matrix Allocation



Extra item stored for each row

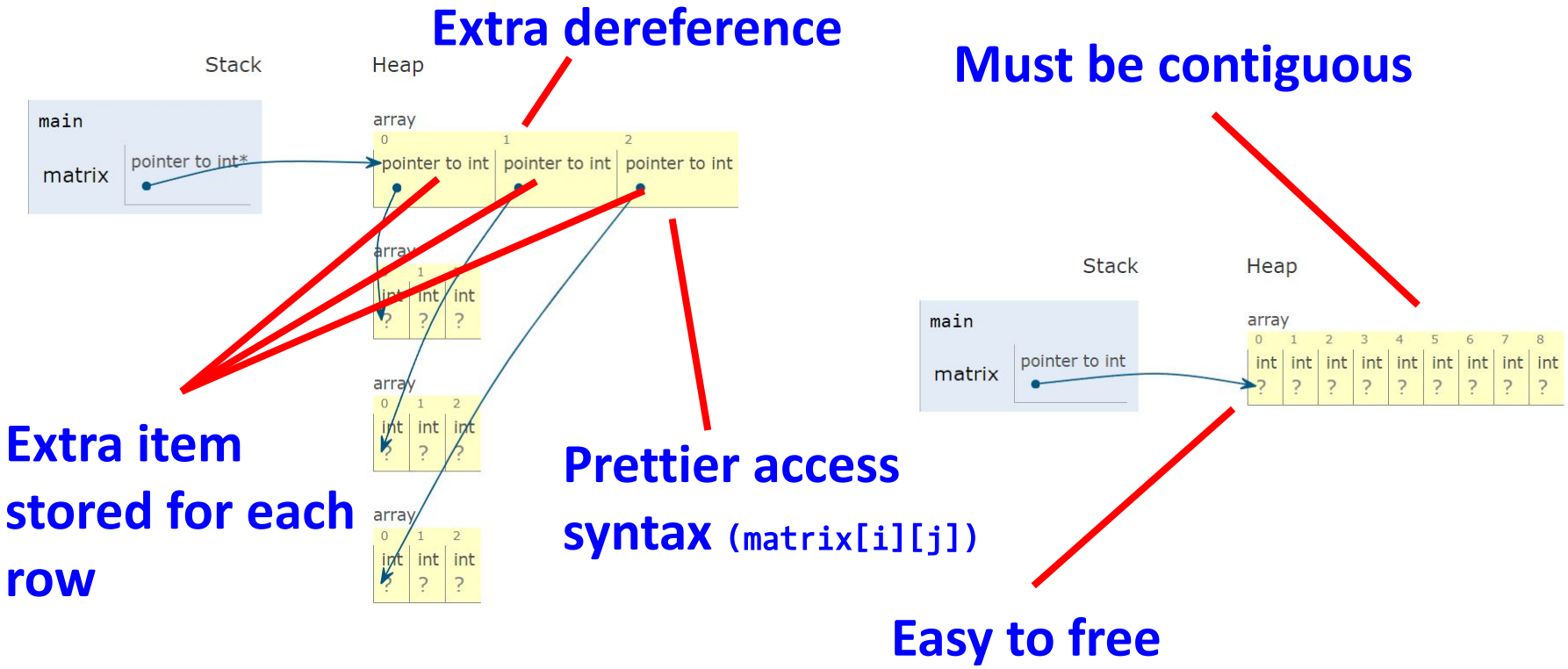
Prettier access syntax (`matrix[i][j]`)

Must be contiguous



Easy to free

2 Flavors of Matrix Allocation



2 Flavors of Matrix Allocation - Performance

Runtime for matrix kernel `m[i][j] = i + j` ***over 1000 trials***

Rows	Columns	int** Runtime (S)	int* Runtime (S)
1000	1000	0.0931	0.0822
2000	2000	0.3543	0.3227
3000	3000	1.2079	1.1200
4000	4000	3.1762	3.0856
5000	5000	5.4211	5.3389
6000	6000	8.1758	8.2131
7000	7000	11.2992	11.1163
100000	1	0.0725	0.0446
200000	1	0.1416	0.0896
300000	1	0.2152	0.1328
400000	1	0.2960	0.1769
500000	1	0.3737	0.2241
600000	1	0.4807	0.2743
700000	1	0.5753	0.3139

2 Flavors of Matrix Allocation - Performance

Runtime for matrix kernel `m[i][j] = i + j` ***over 1000 trials***

Extra dereference has a cost!

Rows	Columns	int** Runtime (S)	int* Runtime (S)
1000	1000	0.0931	0.0822
2000	2000	0.3543	0.3227
3000	3000	1.2079	1.1200
4000	4000	3.1762	3.0856
5000	5000	5.4211	5.3389
6000	6000	8.1758	8.2131
7000	7000	11.2992	11.1163
100000	1	0.0725	0.0446
200000	1	0.1416	0.0896
300000	1	0.2152	0.1328
400000	1	0.2960	0.1769
500000	1	0.3737	0.2241
600000	1	0.4807	0.2743
700000	1	0.5753	0.3139