Agenda

Pointer definition

Data in memory

Hexadecimal and printing pointers

Dynamic allocation: malloc/free and the heap

Arrays revisited

- 1D Arrays
- 2D Arrays

Memory and pointers

A **pointer** is a data type that stores an address in memory

Memory is data stored sequentially in hardware

Analogies

- bookcase filled with boxes
- rows of houses

Each box/house can hold any type of data
Each box/house has an address
Each box/house is 1 byte in size

Exercise: How big is `b` in memory?

```
struct bar {
   char str[4];
   float x,y,z;
};
int main() {
  struct bar b;
  b.str[0] = 'l';
  b.str[1] = 'o';
  b.str[2] = 'l';
  b.str[3] = '\0';
  b.x = 0.0f;
  b.y = 1.0f;
  b.z = 2.0f;
}
```

Exercise: Draw `b` in memory

```
struct bar {
   char str[4];
   float x,y,z;
};
int main() {
  struct bar b;
  b.str[0] = 'l';
  b.str[1] = 'o';
  b.str[2] = 'l';
  b.str[3] = '\0';
  b.x = 0.0f;
  b.y = 1.0f;
  b.z = 2.0f;
}
```

Hexadecimal

- Compact way of representing binary numbers
  - Recall: a binary number is 0 or 1 (base 2)

- Base 16: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Dec	Bin	Hex	Dec	Bin	Hex	Dec	Bin	Hex	Dec	Bin	Hex
0			4			8			12		
1			5			9			13		
2			6			10			14		
3			7			11			15		

Todo: Flashcard Game

Printing pointers

```
struct bar {
   char str[4];
   float x,y,z;
};
int main() {
  struct bar b;
  b.str[0] = 'l';
  b.str[1] = 'o';
  b.str[2] = 'l';
  b.str[3] = '\0';
  b.x = 0.0f;
  b.y = 1.0f;
  b.z = 2.0f;
}
```

Rule of Thumb: Look at the least significant digits!

```
data 1: 0x7ffe2f945460 l
data 2: 0x7ffe2f945461 o
data 3: 0x7ffe2f945462 l
data 4: 0x7ffe2f945463
data 5: 0x7ffe2f945464 0.000000
data 6: 0x7ffe2f945468 1.000000
data 7: 0x7ffe2f94546c 2.000000
```

Pointer or not?

Pointer syntax: which of these variables are pointers?

- char test[3];
- char* test;
- char test;
- const char* test;
- char** test;
- char* test[5];
- char test[];
- int test;
- int test[1000];
- int* test;

Example: Command line arguments

```
#include <stdio.h>

int main(int argc, char** argv)
{
  // Draw the stack here
  for (int i = 0; i < argc; i++)
  {
    printf("%d) %s\n", i, argv[i]);
  }
}
```

Draw the function stack for the following command:

$ ./a.out apple banana carrot

What would the command line arguments be for this program?

NOTE: We can also declare main like so: int main(int argc, char* argv[])

Dynamic allocation

Idea: Ask OS for more memory when the program is running

Use cases:

- Don't know sizes of arrays until runtime
- Need a variety of input sizes
- Want to only use as much memory as necessary

Heap memory

Dynamic memory is allocated from the heap

Process:

1. Ask for block of memory
2. Assign pointer to keep track of it ("anonymous memory")
3. Free memory when you're finished

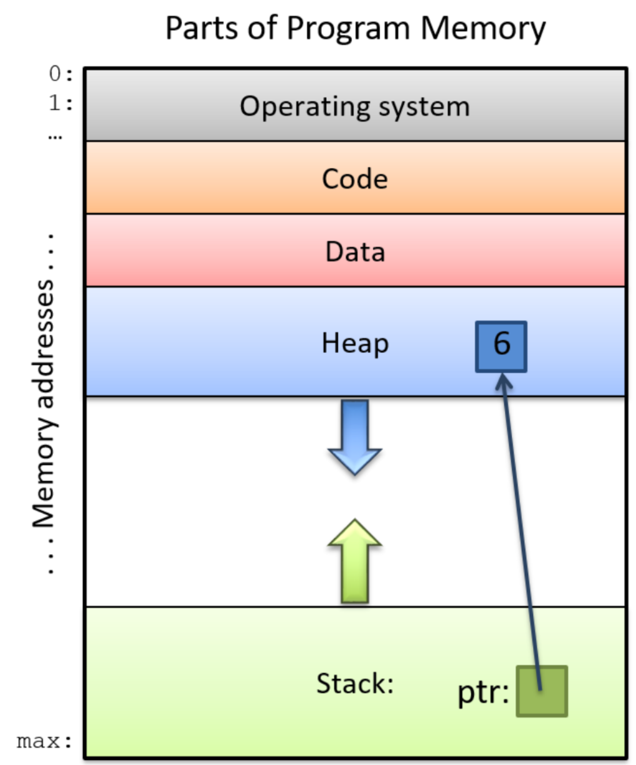


Figure 17. A pointer on the stack points to a block of memory that was allocated from the heap.

Failure to return memory is called a **memory leak**

Malloc and Free

malloc = “memory allocation”

If **malloc** is successfull

  Returns address to beginning of memory block
  (e.g. the **base address**)

Else

  malloc returns NULL

free returns the memory to the heap where it can be reused

Example: Malloc and Free

```
#include <stdio.h>
#include <stdlib.>

int main() {
  int *arr;
  char *c_arr;

  // allocate an array of 20 ints on the heap:
  arr = malloc(sizeof(int) * 20);

  // allocate an array of 10 chars on the heap:
  c_arr = malloc(sizeof(char) * 10);
  free(arr);
  free(c_arr);
  return 0;
}
```

Rules of Thumb:

- Pointer types should match the data type
- Use sizeof() to get the data type size
- For every malloc, make sure you have a corresponding free

Example: Malloc and Free

Draw the stack diagram for this program.

```
#include <stdio.h>
#include <stdlib.>

int main() {
  int *arr;



  arr = malloc(sizeof(int) * 20);
  arr[1] = 10;



  arr = malloc(sizeof(int) * 10);
  free(arr);
 return 0;
}
```

Example: Malloc and Free

Draw the stack diagram for this program.

```
#include <stdio.h>
#include <stdlib.>

int main() {
  int *arr;
  char *c_arr;

  // allocate an array of 20 ints on the heap:
  arr = malloc(sizeof(int) * 20);

  // allocate an array of 10 chars on the heap:
  c_arr = malloc(sizeof(char) * 10);
  free(arr);
  free(c_arr);
  return 0;
}
```

Example: malloc and free

This code crashes. Why?

```
int *arr = NULL;
char *c_arr = NULL;

arr = malloc(sizeof(int) * 20);
c_arr = malloc(sizeof(char) * 10);

free(arr);
arr[0] = 2;
c_arr[0] = ‘d’;

free(c_arr);
```

Example: malloc and free

This code crashes. Why?

```
int *arr = NULL;
char *c_arr = NULL;

arr = malloc(sizeof(int) * 20);
c_arr = malloc(sizeof(char) * 10);

free(arr);
free(c_arr);
free(arr);
```

Example: Passing simple types to functions

```
int init(int a, int size) {
   float value = a * size;
   return value;
}

int main() {
  int b = 10;
  int x = 3;
  int temp = init(b, x);
  // draw stack here

  printf(“temp: %d\n”, temp);
}
```

Example: Passing arrays to functions

```
void init_array(int *arr, int size) {
  for (int i = 0; i < size; i++) {
    arr[i] = i;
  }
}

int main() {
  int arr1[4];

  init_array(arr1, 4);
  // draw stack here

}
```

Example: Stack and heap

```
void init_array(int *arr, int size) {
  for (int i = 0; i < size; i++) {
    arr[i] = i;
  }
}

int main() {
  int* arr1 = NULL;

  arr1 = malloc(sizeof(int) * 4);
  if (arr1 == NULL) {
    printf("malloc error\n");
    exit(1);
  }

  init_array(arr1, 4);
  // draw stack here
  free(arr1);
}
```

Creating arrays

Static arrays are created with a fixed size at compile time and use stack memory.

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

Dynamic arrays are created while the program is running and use heap memory.

```
int* array = malloc(sizeof(int) * 10);
free(array)
```

Deleting arrays

Static arrays are automatically deleted when the function frame is destroyed.

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

Dynamic arrays must be manually deleted using free.

```
int* array = malloc(sizeof(int) * 10);
free(array)
```

Array elements are contiguous in memory

Each element in an array of type T is sizeof(T) apart in memory,

- e.g. `int arr[10]` stores 10 integers consecutively in memory

Exercise: Reverse Array

$ **./reverse_numbers**
Enter the size of the array: **4**
Enter a number: **2**
Enter a number: **4**
Enter a number: **6**
Enter a number: **8**

The array in reverse order is: 8 6 4 2

2D arrays

A 2D array is an array of arrays

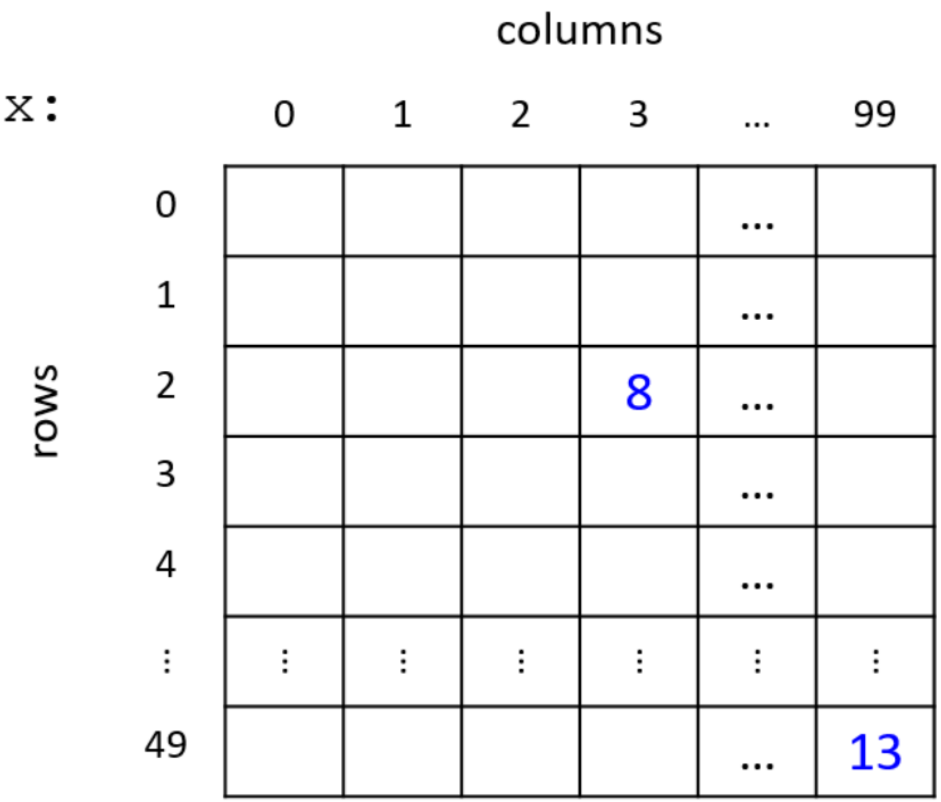
```
int matrix[50][100];
```

Each row is a pointer to an array of columns

2D arrays as contiguous blocks

```
int matrix[50][100];
```

`matrix:`




```
matrix[2][3] = 8;
matrix[49][99] = 13;
```

2D arrays as contiguous blocks

Mental model is to think of the 2D array as a grid but the reality is that memory is stored in **row-major order** as a NROWS * NCOLS array

```
int v1 = arr[2][1];
int v2 =  *(*arr + 2*4 + 1);

// location = row * NCOLS + col
```

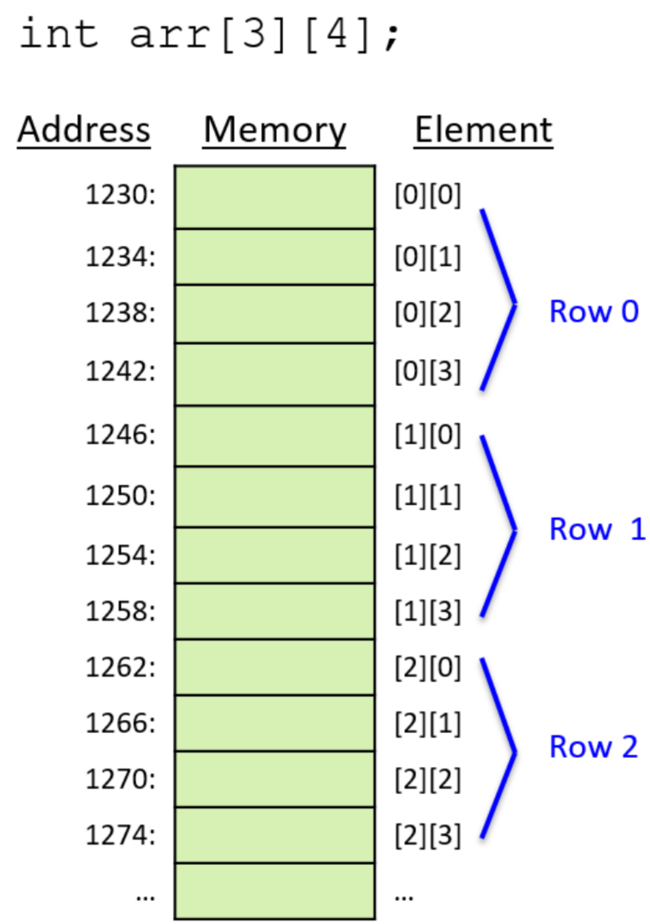


Figure 21. The layout of a two-dimensional array in row-major order.

Dynamic 2D arrays

Two approaches:

Use single call to malloc (more efficient)

Use multiple calls to malloc: one to create the 2D array followed by a malloc to allocate each row

Dynamic 2D array: Method 1
“Flat 2D array”

```
int *two_d_array;

two_d_array = malloc(sizeof(int)*(3*4));
```

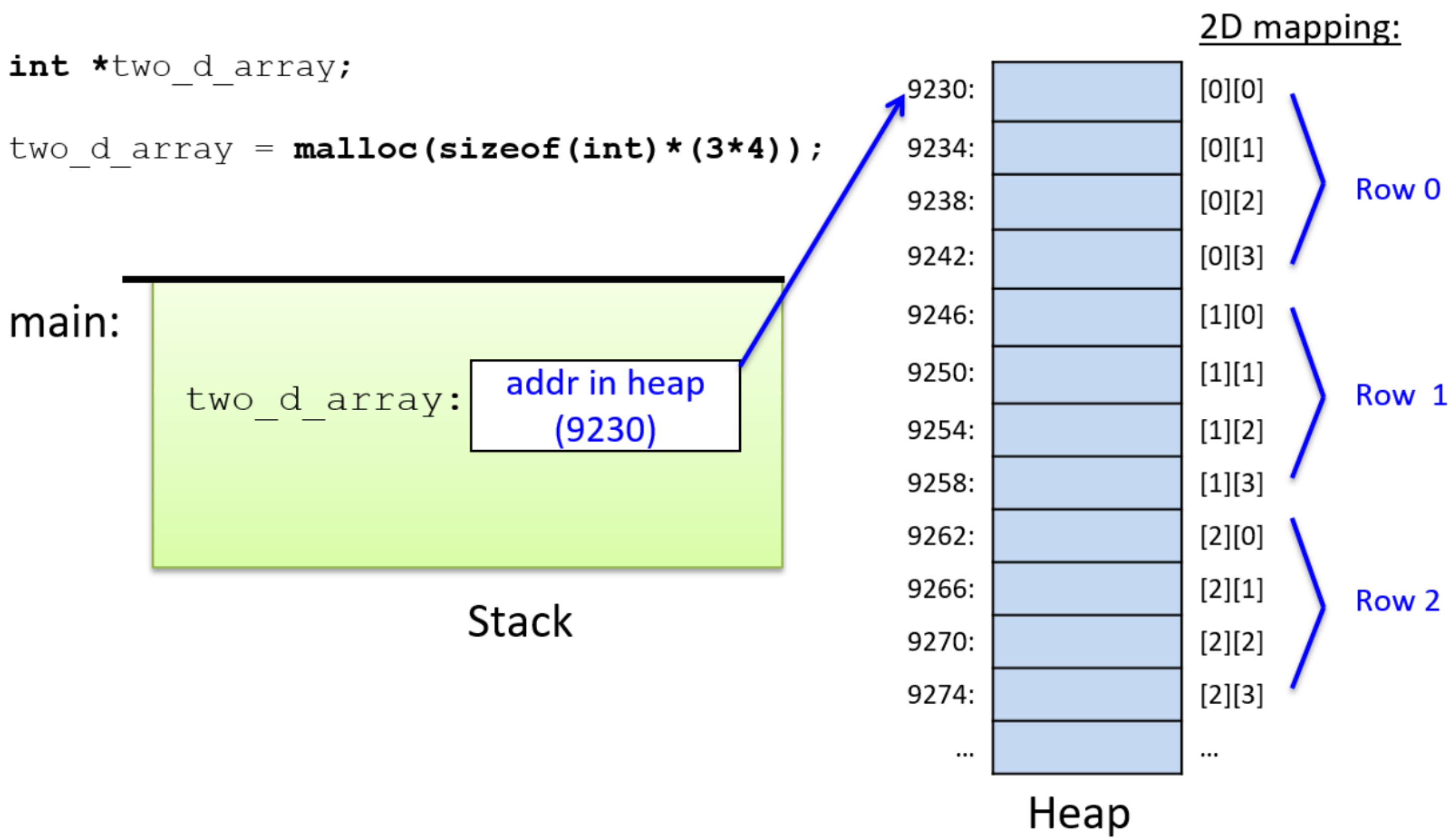


Figure 22. The results of allocating a 2D array with a single call to malloc.

How to index?

Flat 2D array: Computing indices

Suppose we use a flat array to store a 4x4 matrix.
What is the index for element (2, 3)?

Suppose we use a flat array to store a 3x10 matrix.
What is the index for element (2, 3)?

Suppose we use a flat array to store a 10x3 matrix.
What is the index for element (4, 0)?

Flat 2D array: Computing row, col given an index

What is the general formula for converting from (i,j) to an index for an NxM matrix stored as a flat array?

What is the general formula for converting from a 1D index to a 2D index (i,j) for an NxM matrix stored as a flat array?

Dynamic 2D array: Method 2
“Array of arrays”

```
int **two_d_array;

two_d_array = malloc(sizeof(int *) * N);
for (i=0; i < N; i++) {
    two_d_array[i] = malloc(sizeof(int) * M);
}
```

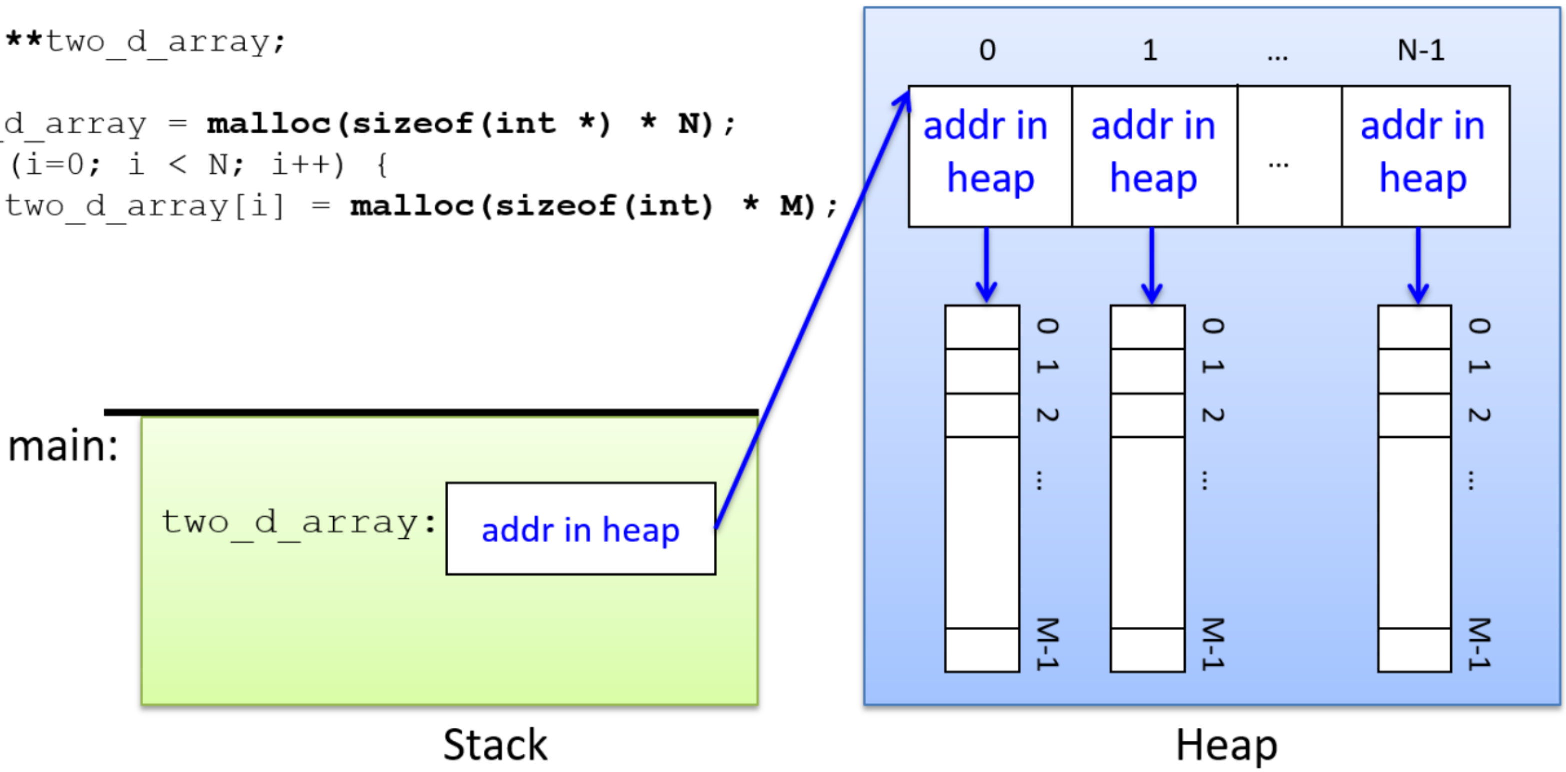


Figure 23. The arrangement of memory after allocating a 2D array with N+1 malloc calls.

How to index?

Exercise: 2D array using flat 2D array

Sketch a program, **ask_matrix_flat.c**, that creates and frees a 3X4 matrix.

Exercise: 2D arrays using “array of arrays”

Sketch a program, **ask_matrix_aa.c**, that creates and frees a 3X4 matrix.