CPSC 259

Dynamic memory

No iClickers in this deck!

Dynamic memory

- Arrays declared as local variables must have a known size at compile time
  - but sometimes we don't how much space we need until runtime
- Suppose we expect users to only need to store up to 1000 values, so we hard-code "1000" as an array size
  - What if user needs more?
  - What if user only needs 5?

Change code and recompile

Wastes memory

- If the value 1000 is hard-coded, this is hard to find and change
  - especially if used in multiple locations
- If hardcoded as a symbolic constant, still cannot change without recompiling

Memory management in C

- We have already seen how locally-declared variables are placed on the function call stack
  - allocation and release are managed automatically
- The available stack space is extremely limited
  - placing many large variables or data structures on the stack can lead to stack overflow
- Stack variables only exist as long as the function that declared them is running

```
void MyFunction() {
  int i; // local variable created on stack
} // local variable goes out of scope and is deleted
```

Dynamic memory allocation

- At run-time, we can request extra space on-the-fly, from the *memory heap*

- Request memory from the heap – "allocation"
- Return allocated memory to the heap (when we no longer need it) – "deallocation"

- Unlike stack memory, items allocated on the heap must be explicitly freed by the programmer

Dynamic memory allocation

malloc returns NULL if unsuccessful

- Function `malloc` returns a pointer to a memory block of at least `size` bytes:

use sizeof(....)

```
ptr = (cast-type*) malloc(byte-size);
```

local variable

malloc returns void*

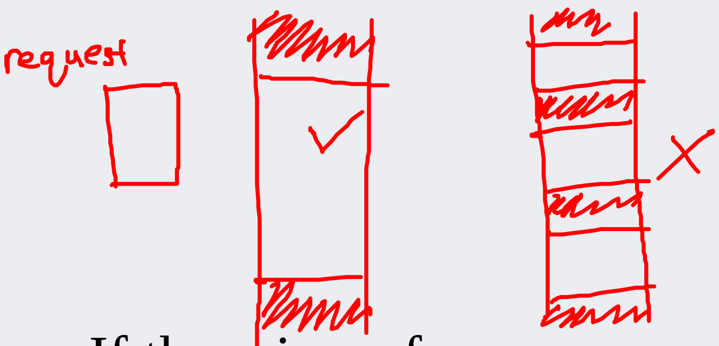
- Function `free` returns the memory block (previously allocated with `malloc`) and pointed to by `ptr` to the memory heap:

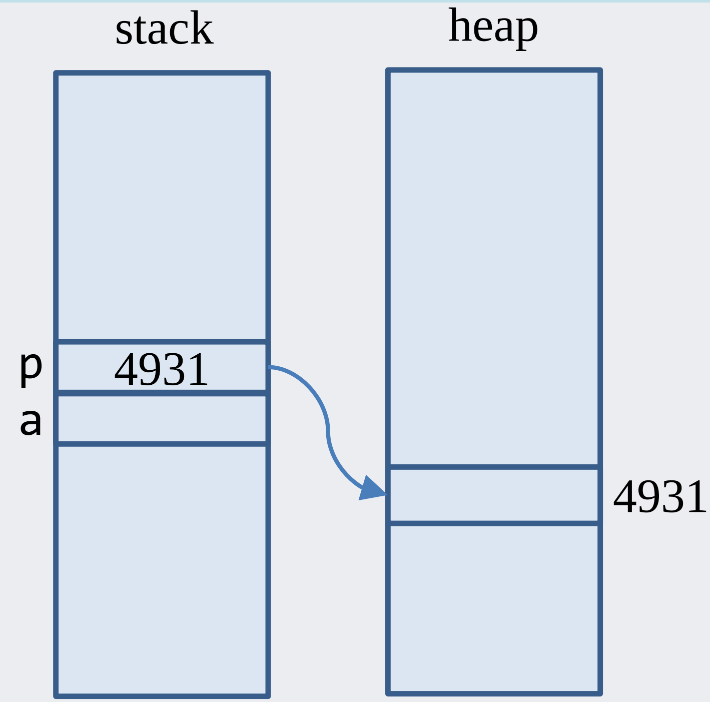
```
free(ptr);
```

- The system knows how many bytes need to be freed, provided we supply the correct address `ptr`

Heap example

```
int main() {
  int a;
  int *p = (int*) malloc(sizeof(int));
  *p = 10;
}
```





- If there is no free memory left on the heap, `malloc` will return a null pointer
- Note: `malloc` only allocates the space but does not initialize the contents.
  - Use `calloc` to allocate and clear the space to binary zeros

- Suppose we want to allocate space for exactly 10 integers in an array

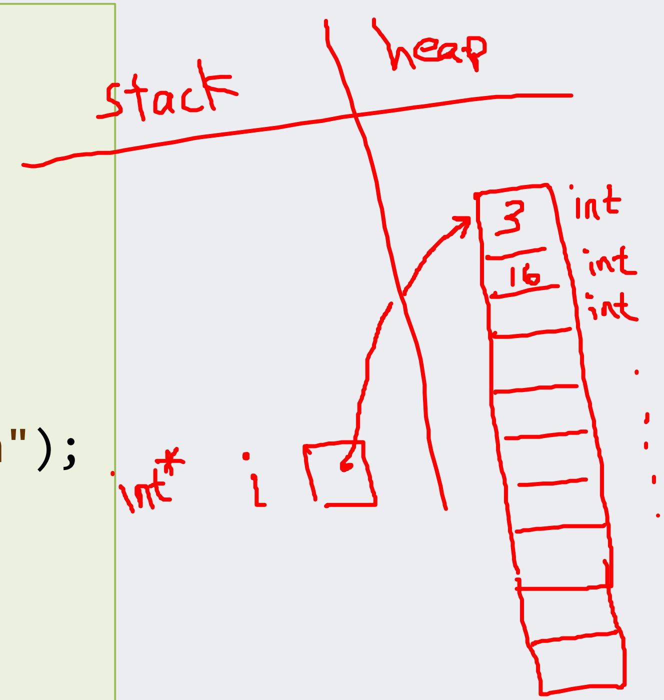
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int* i;
  i = (int*) malloc(10*sizeof(int));
  if (i == NULL) {
    printf("Error: can't get memory...\n");
    exit(1); // terminate processing
  }

  i[0] = 3;  // equivalent: *(i+0) = 3;
  i[1] = 16; // *(i+1) = 16;
  printf("%d", *i);
  ...
}
```

can be variable

free(i);



Allocating dynamic arrays

From user input, variable array size

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

int main() {
  int employees, index;
  double* wages;
  printf("Number of employees? ");
  scanf("%d", &employees);

  wages = (double*) malloc(employees * sizeof(double))
  if (!wages) { // equivalent: if (wages == NULL)
    printf("Error: can't get memory...\n");
  }

  printf("Everything is OK\n");
  ...
}
```

See `dma_examples.c`

Dangling pointers

- When we are done with an allocated object, we free it so that the system can reclaim (and later reuse) the memory

```
int main() {
  int* i = (int*) malloc(sizeof(int));
  *i = 5;
  free(i);


  printf("%d", *i);
}
```

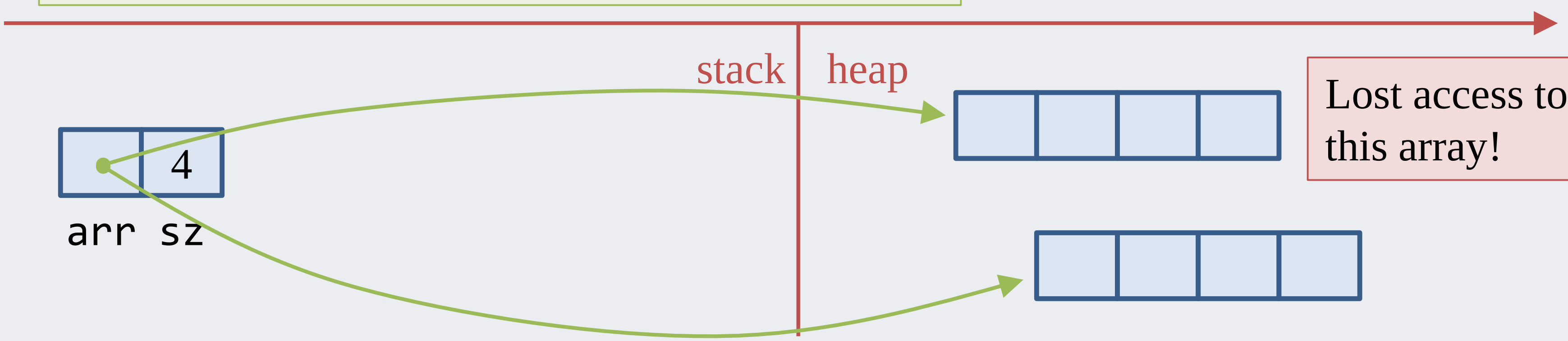
The space is marked as free, but the value remains until it is overwritten

- If the pointer continues to refer to the deallocated memory, it will behave unpredictably when dereferenced (and the memory is reallocated) – a dangling pointer
  - Leads to bugs that can be subtle and brutally difficult to find
  - So, set the pointer to `NULL` after freeing `i = NULL;`

Memory leaks

- If you lose access to allocated space (e.g. by reassigning a pointer), that space can no longer be referenced, or freed
  - And remains marked as allocated for the lifetime of the program

```
int* arr;
int sz = 4;
arr = (int*) malloc(sz*sizeof(int));
arr[2] = 5;
arr = (int*) malloc(sz*sizeof(int));
arr[2] = 7;
```



Lost access to this array!

See `memory_leak.c`

Exercise

- What is printed to the screen?
  - Also clearly identify memory leaks and dangling pointers

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x=%d, *y=%d, z=%d, w=%d\n", *x, *y, z, w);
```

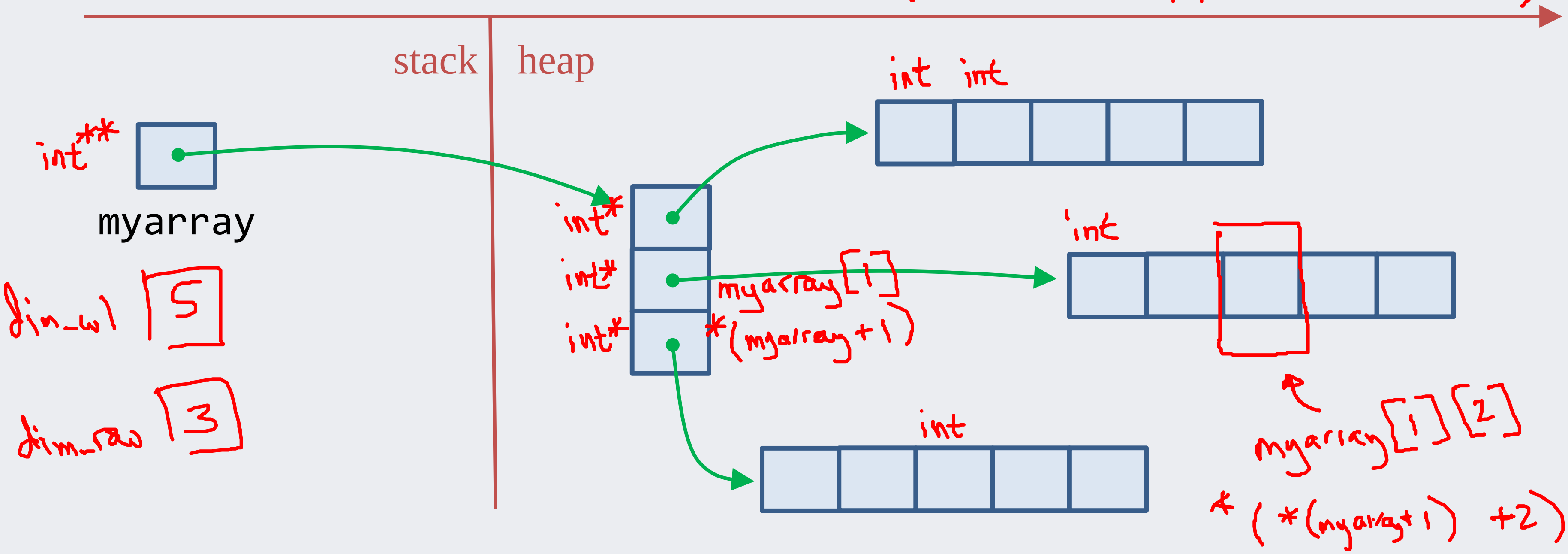
Dynamic allocation of a 2D array

See dma_2d.c for details

```
int dim_row = 3;
int dim_col = 5;
int** myarray; // pointer to a pointer
```

```
myarray = (int**) malloc(dim_row * sizeof(int*));
for (int i=0; i< dim_row; i++) {
    myarray[i] = (int*) malloc(dim_col * sizeof(int));
}
```

(deallocate in reverse order)



Stack memory vs heap memory

- Stack
  - fast access
  - allocation/deallocation and space automatically managed
  - memory will not become fragmented
  - local variables only
  - limit on stack size (OS-dependent)
  - variables cannot be resized
- Heap
  - variables accessible outside declaration scope
  - no (practical) limit on memory size
  - variables can be resized
  - (relatively) slower access
  - no guaranteed efficient use of space
  - memory management is programmer's responsibility

Readings for this lesson

- Thareja
  - Appendices A, B, E

- Next class:
  - Thareja, Chapters 4 – 5