C pointers

Pointers

Dynamic memory

Geoff's self-checklist:
❑ ~~Start iClicker Cloud~~
❑ Record lecture

Announcements

- Lab quizzes this week! Please attend your registered section
  - arrays
  - data types
  - loops
  - file input using `fgets` and `sscanf`
- See Piazza post @124 for procedure
  - Send a private message to your Zoom TA in case of technical issues

Modifying parameters with call-by-reference

- Consider these two “swap” functions and their calls
  - Trace the call stack to see what is printed after each call

```
void swap_v(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

```
void swap_r(int* a, int* b) {
  int temp = *a;
  *a = *b;
  b = temp;
}
```

```
...
int x = 3;
int y = 9;
swap_v(x, y);
printf("x: %d, y: %d", x, y);
```

```
...
int p = 3;
int q = 9;
swap_r(&p, &q);
printf(“p: %d, q: %d", p, q);
```

Pointer to a pointer?

```c
int main() {
  int x = 5;
  int* p = &x;
  *p = 6;
  int** q = &p;
  int*** r = &q;

  printf("%d\n", *p);
  printf("%d\n", *q);
  printf("%d\n", *(*q));
}
```

- "You can keep adding levels of pointers until your brain explodes or the compiler melts – whichever happens soonest"
  - stackoverflow user JeremyP

Back to call-by-reference

- Consider the following function that adds two parameters supplied by reference

```
int  add(int* num1, int* num2) {
  int sum = *num1 + *num2;
  return  sum;
}

int main() {
  int  a = 2;
  int  b = 4;
  int  c = add(&a, &b);
  printf("sum = %d",  c);
}
```

- Can we modify the add function so that it uses a pointer to return the answer?

Returning pointers

- Will it work if we just change the return type to pointer, and return the sum variable's address?

```
int* add(int* num1, int* num2) {
  int sum = *num1 + *num2;
  return &sum;
}

int main() {
  int  a = 2;
  int  b = 4;
  int* c = add(&a, &b);
  printf("sum = %d", *c);
}
```

This will have problems!
Think about what is (or was) on the call stack

Passing array elements as parameters

- Arrays are passed by reference by default

```
double getMaximum(double data[], int size); // prototype
double getMaximum(double* data, int size); // equivalent prototype

double answer = getMaximum(myarr, length); // function call
```

- Note that we do not need to provide "&" when specifying the address of the entire array (i.e. the address of the first element)
- If we want to specify the address of an individual element of the array, we would need the address operator
  - e.g. `&data[4]`

Pointer arithmetic

- If we know the address of the first element of an array, we can compute the addresses of the other array elements
  - (or whatever comes after, if it is meaningful)

```
int A[5];
int* q = &A[0];
printf("q address: %p\n", q);
A[0] = 2;
A[1] = 4;

printf("value of q: %d\n", *q);
printf("value of q+1: %d\n", *(q + 1));
```

```
int x = 5;
int* p = &x;
printf("p address: %p\n", p);
printf("value of p: %d\n", *p);
printf("value of p+1: %d\n", *(p + 1));
```

Dynamic memory

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? Change code and recompile
  - What if user only needs 5? 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

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

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

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

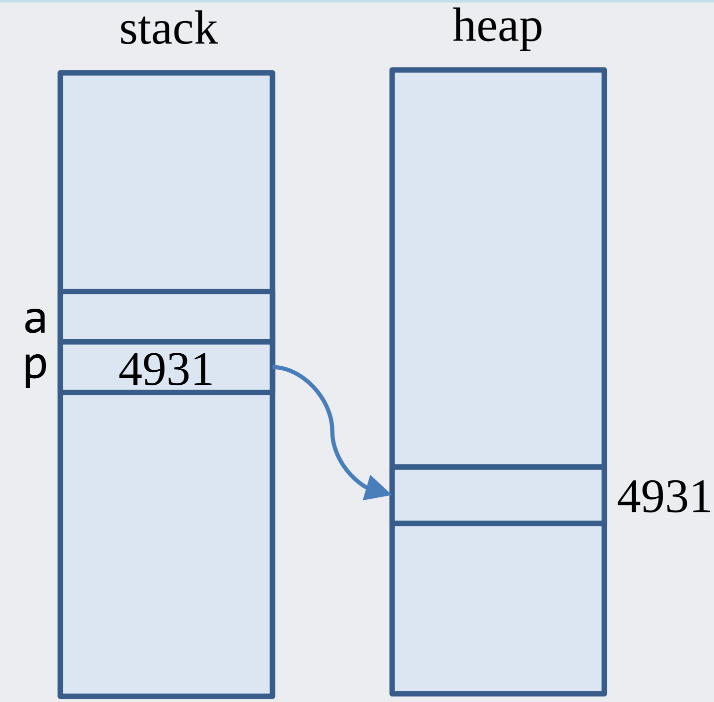
- Function `free` returns the memory block (previously allocated with `malloc`) and pointed to by `ptr` to the memory heap:

```
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

Allocating dynamic arrays

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

```
#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);
  ...
}
```

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);
}
```

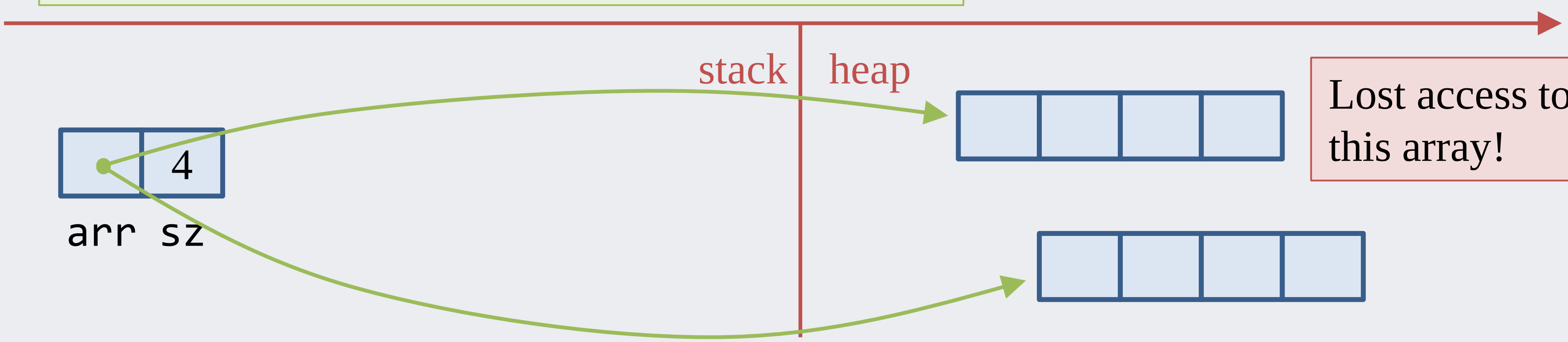
The space is marked as free, but the value remains until it is overwritten

- 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

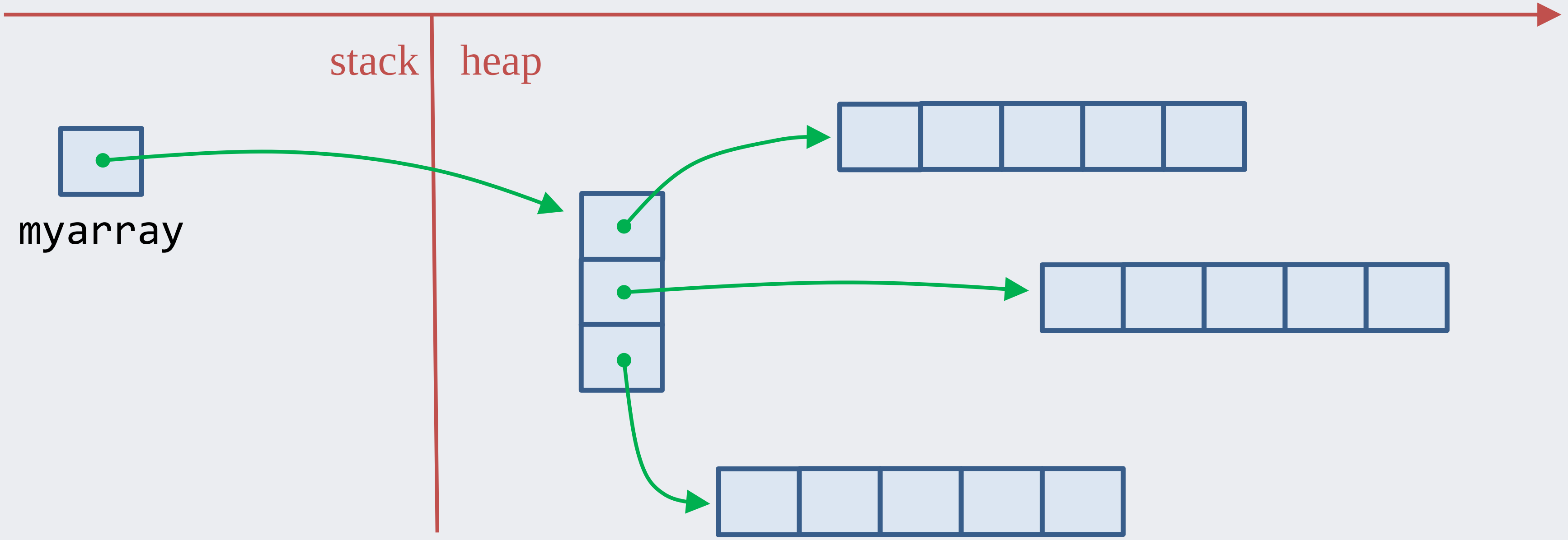
```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*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
```



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