Dynamic Memory Allocation

- So far, variables we have defined and used reside in program memory (on a program stack)
  - Pros: fast, automatically managed by the CPU
  - Cons: program memory limited, cannot resize variables
- Dynamic memory allows runtime declaration of mutable variables (e.g., arrays) of any reasonable size
  - Comes from an area of general-use, unmanaged memory for a program called the heap
malloc()

- Dynamically allocates specified # of contiguous bytes in memory and returns a void pointer to it (bytes are uninitialized!)

- Note: Always check returned pointer! If NULL then memory allocation failed (rare but possible if requested # of bytes is huge)

- Be sure to use #include <stdlib.h>

- void * malloc(size_t numBytes)
Example

//create dynamically allocated array of 10 ints
int * arr = (int *) malloc(10 * sizeof(int));

//check for NULL
if(arr == NULL) {
    printf(“Cannot allocate memory for array\n”);
    return EXIT_FAILURE;
}

*(arr + 0) = 10;
*(arr + 1) = 20;
arr[2] = 30;
arr[3] = 40;
calloc()

- Similar to malloc()
  - but separate arguments for # elements and element size
- And calloc initializes allocated memory bits to 0
- void * calloc(size_t numEl, size_t elSize)
Example

//create dynamically allocated array of 10 ints
int * arr = (int *) calloc(10, sizeof(int));
//check for NULL
if(arr == NULL) {
    printf("Cannot allocate memory for array\n");
    return EXIT_FAILURE;
}

//THIS IS SAFE
printf("%d %d\n", arr[1], *(arr + 3));
Dynamically allocated memory is not automatically released in the heap when a variable “dies”

I.e., it is unavailable for reuse until the program explicitly frees it back to the heap

Called a memory leak

`free()` returns dynamically allocated memory to the heap

`void free(void * ptr)`
Suggestion

- Whenever you write a `malloc()` or `calloc()`, immediately write a matching `free()`
Example

//create dynamically allocated array of 10 ints
int * arr = (int *) malloc(10 * sizeof(int));

... //use the array

free(arr);

arr = (int *) malloc(500 * sizeof(int));

... //use the array for something else

free(arr);
valgrind

- Not calling free before a dynamic memory variable dies is common source of memory leaks
- Valgrind on the elk computers can help find memory leaks in your program
  - valgrind <program name and args>
Resizing an Array

- One approach:
  1. Dynamically allocate space for new array
  2. Copy contents from old to new array
  3. Free old array
Or `realloc()`

- Does all of that for you and returns pointer to the newly sized array
  - Modifies `ptr` parameter thus same as return value
  - Check return value for NULL (if so, `ptr` is unmodified)
  - New elements are uninitialized

```c
void * realloc(void * ptr, size_t newSize)
```
Other Memory Functions

- “mem” functions: byte-level functions for copying, comparing, filling, and searching
  - `memcpy`: copy bytes from source to dest
  - `memcmp`: compare bytes
  - `memset`: fill memory block with a value
  - `memchr`: locate a byte in memory block
Other String Functions

- **strncpy**: copy up to n chars from source to dest
- **strncat**: concatenate up to n chars ...
- **strncmp**: compare up to n chars ...
- **strchr**: locate first occurrence of char in a string
- **strstr**: locate a substring in a string
- **strtok**: split string into tokens (a bit advanced)
Exercise 2

- Refactor pair program to use dynamic memory
  - Copy it into a separate folder, called pair2
- Refactor pair.c functions to return a struct * with the result
- Will need to refactor pairs.c a little
- Test by comparing output between the programs