CS 3410 Lab 3

Fall 2025

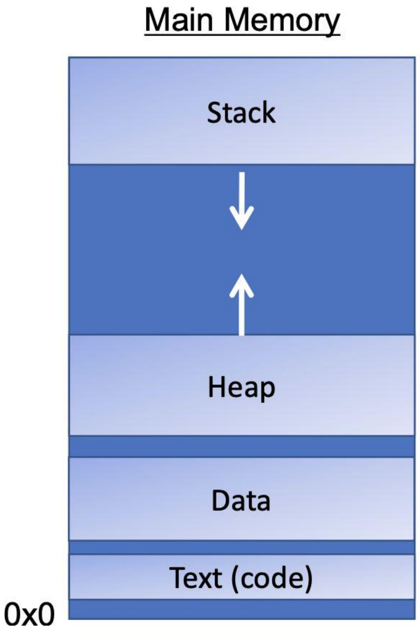
Agenda

1. C Memory Management
2. Linked List
3. Implementing Linked List in C
4. A3 Tips

C Memory Management

The Heap

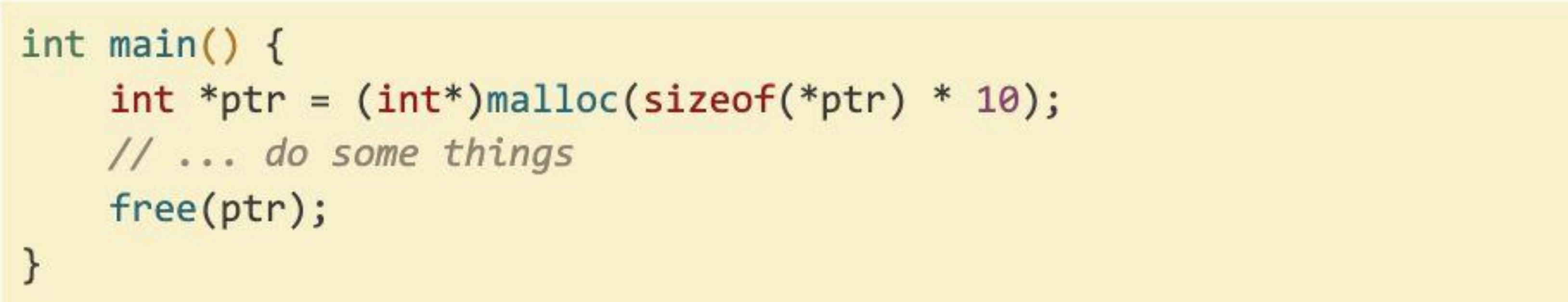
- The heap is an area of memory below the stack that grows up towards higher addresses
- Unlike the stack, where memory goes away when a function finishes, the heap provides memory that persists until the caller is done with it



How do we access memory on the heap?

- `malloc():` Request a pointer to a contiguous block of memory on the heap
- `free():` Release or deallocate the allocated memory back to the operating system

`malloc()` and `free()` syntax



```c
int main() {
    int *ptr = (int*)malloc(sizeof(*ptr) * 10);
    // ... do some things
    free(ptr);
}
```

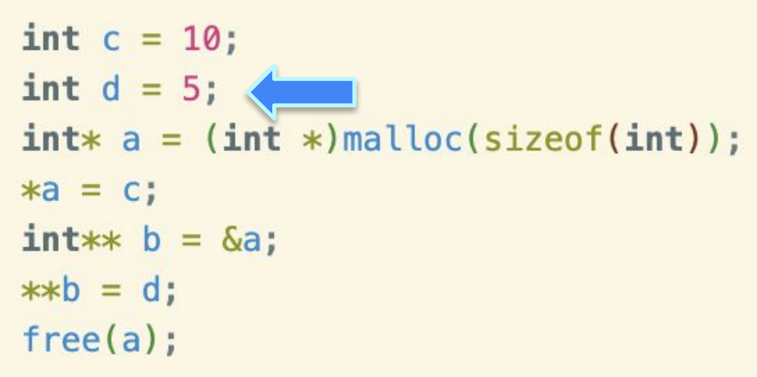
Rule of Thumb

Every call to `malloc()` should have a corresponding call to `free()`

- An allocation that is not freed by the time the program ends is called a **Memory Leak**

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.



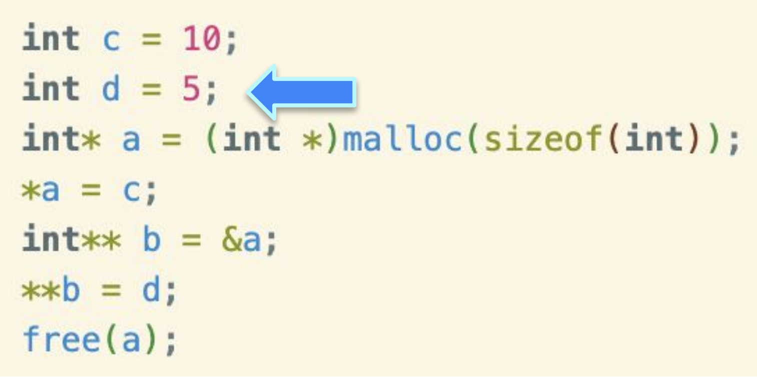
```
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

Stack		
Address	Name	Value
0xFFFF		
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB		
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7		
0xFFF6		
0xFFF5		
0xFFF4		

0x0008		
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		
Address	Name	Value
Heap		

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.



```
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

Stack

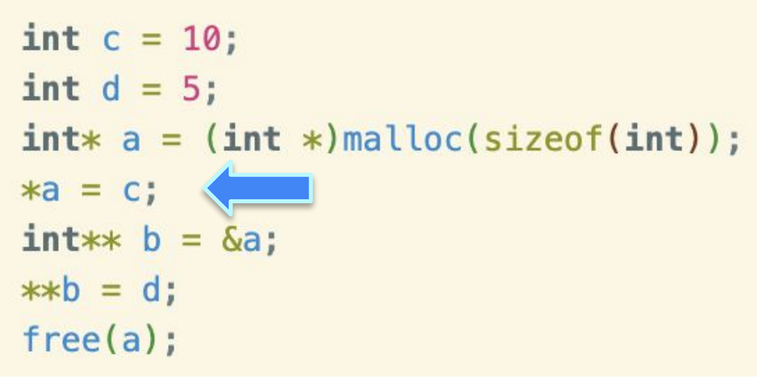
Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7		
0xFFF6		
0xFFF5		
0xFFF4		

Address	Name	Value
0x0008		
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		

Heap

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.



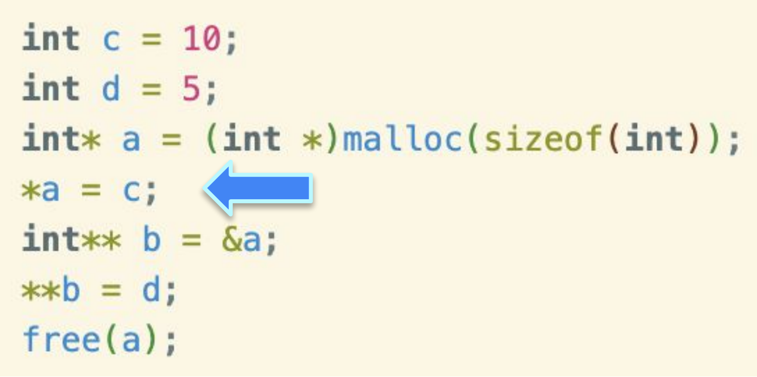
```
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

Stack		
Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7		
0xFFF6		
0xFFF5		
0xFFF4		

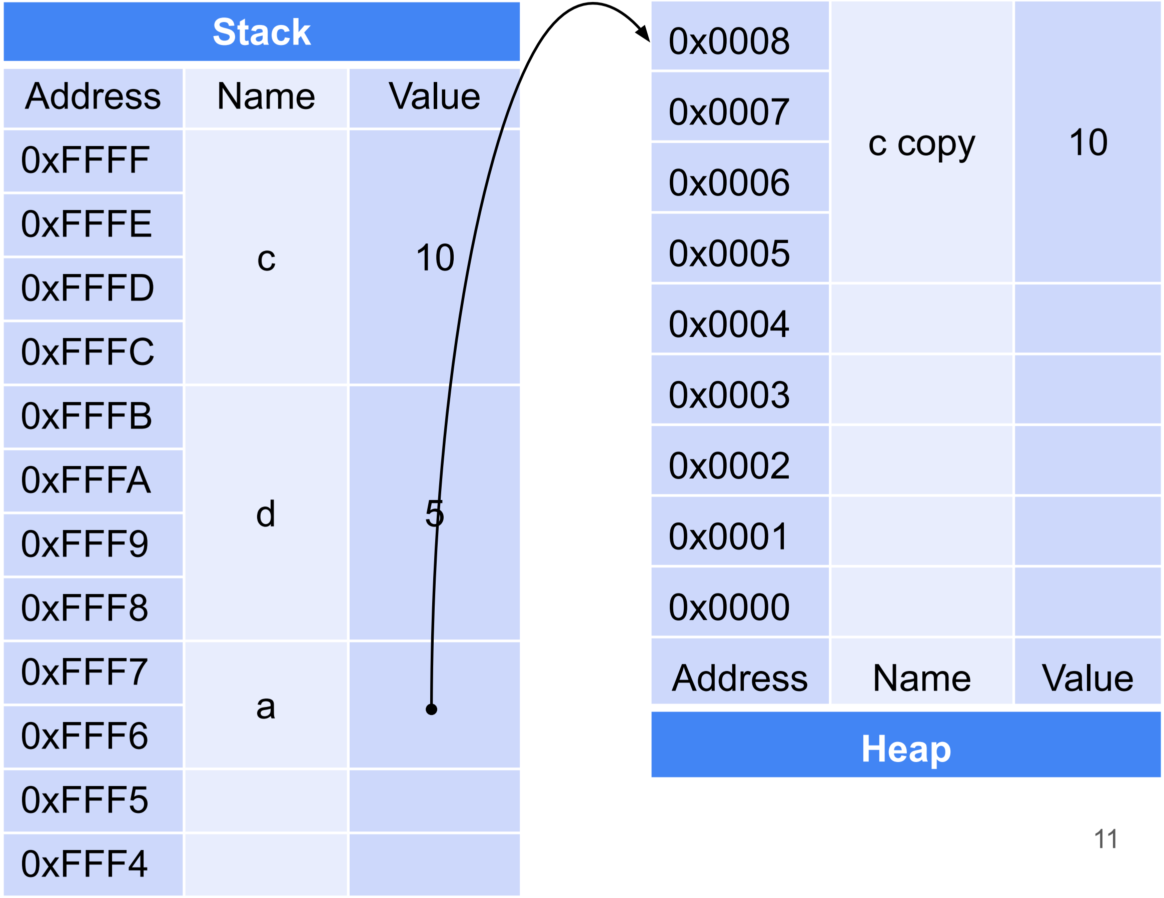
0x0008		
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		
Address	Name	Value
Heap		

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.



```
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```



Stack

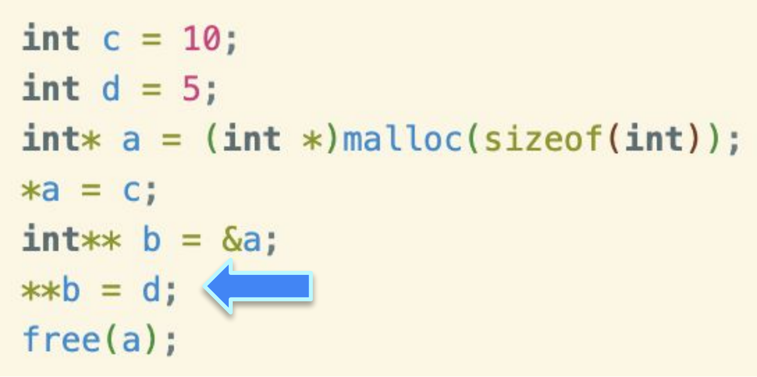
Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7	a	
0xFFF6		
0xFFF5		
0xFFF4		

Address	Name	Value
0x0008	c copy	10
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		

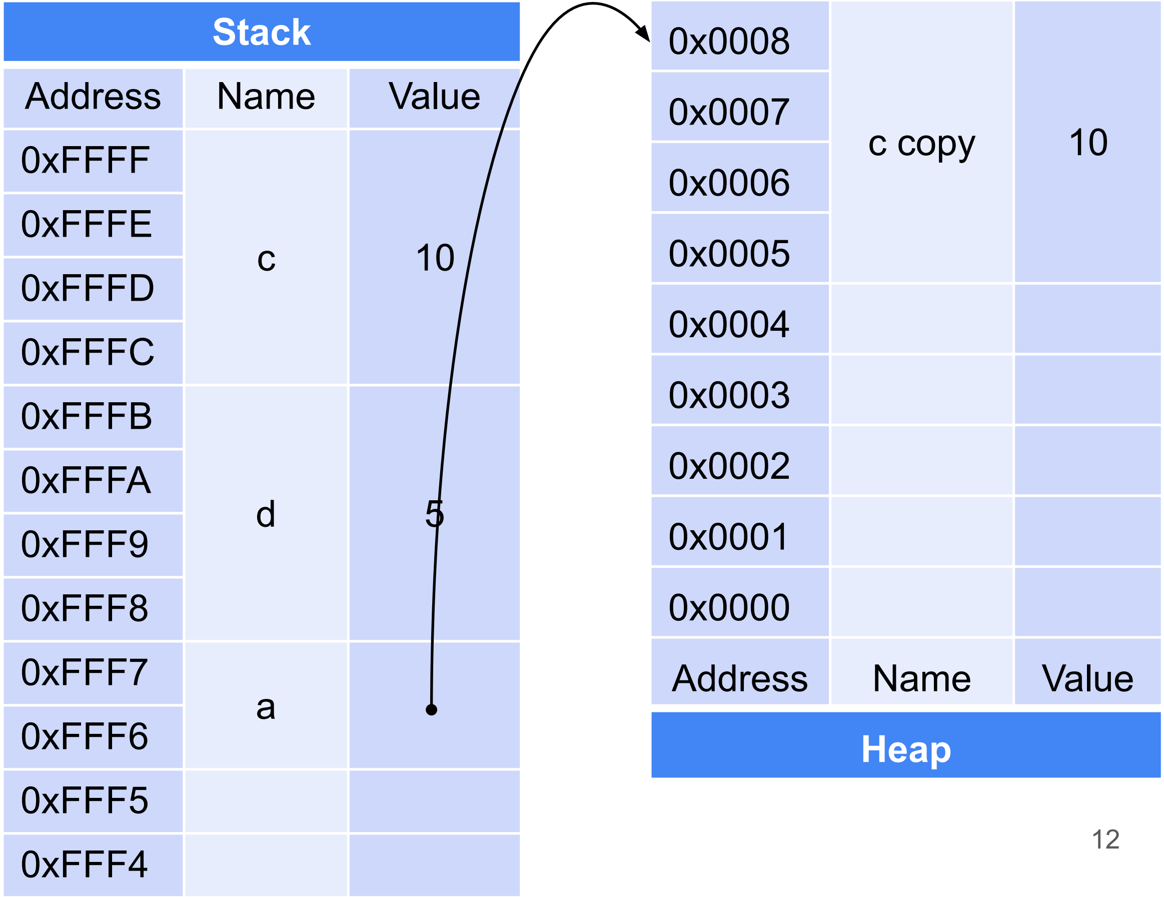
Heap

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 8 bytes.



```
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```



Stack

Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7	a	
0xFFF6		
0xFFF5		
0xFFF4		

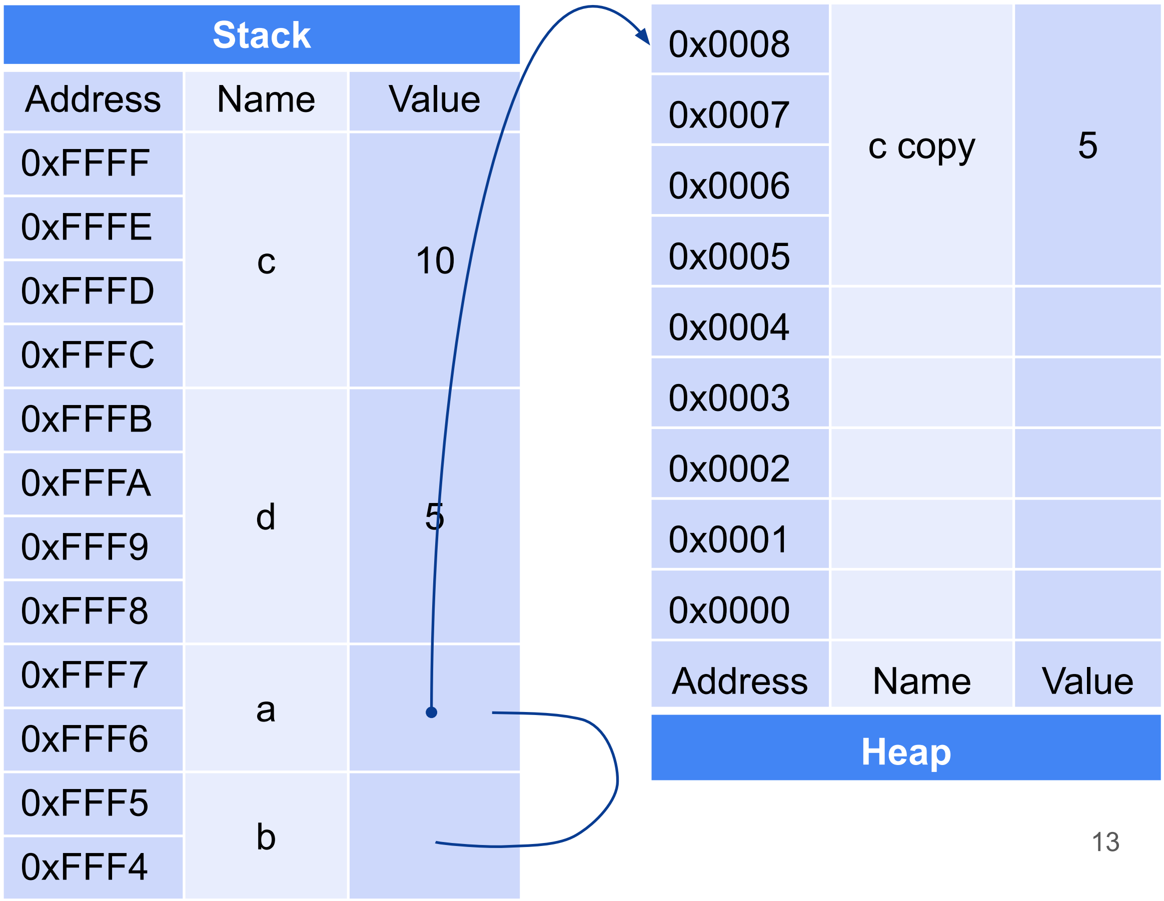
Heap

Address	Name	Value
0x0008	c copy	10
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.

```c
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```



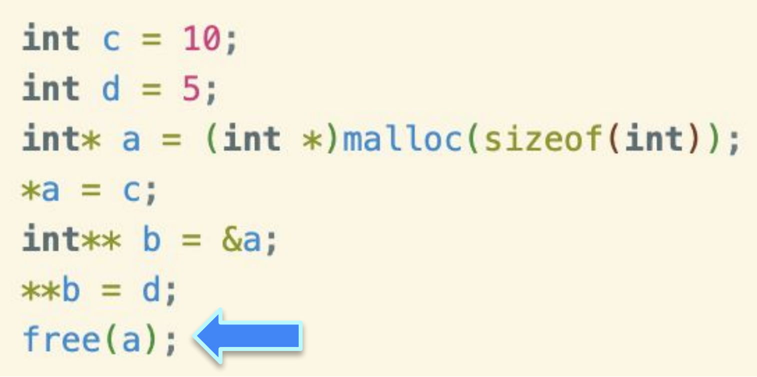
Stack		
Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7	a	
0xFFF6		
0xFFF5	b	
0xFFF4		

Address	Name	Value
0x0008	c copy	5
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		

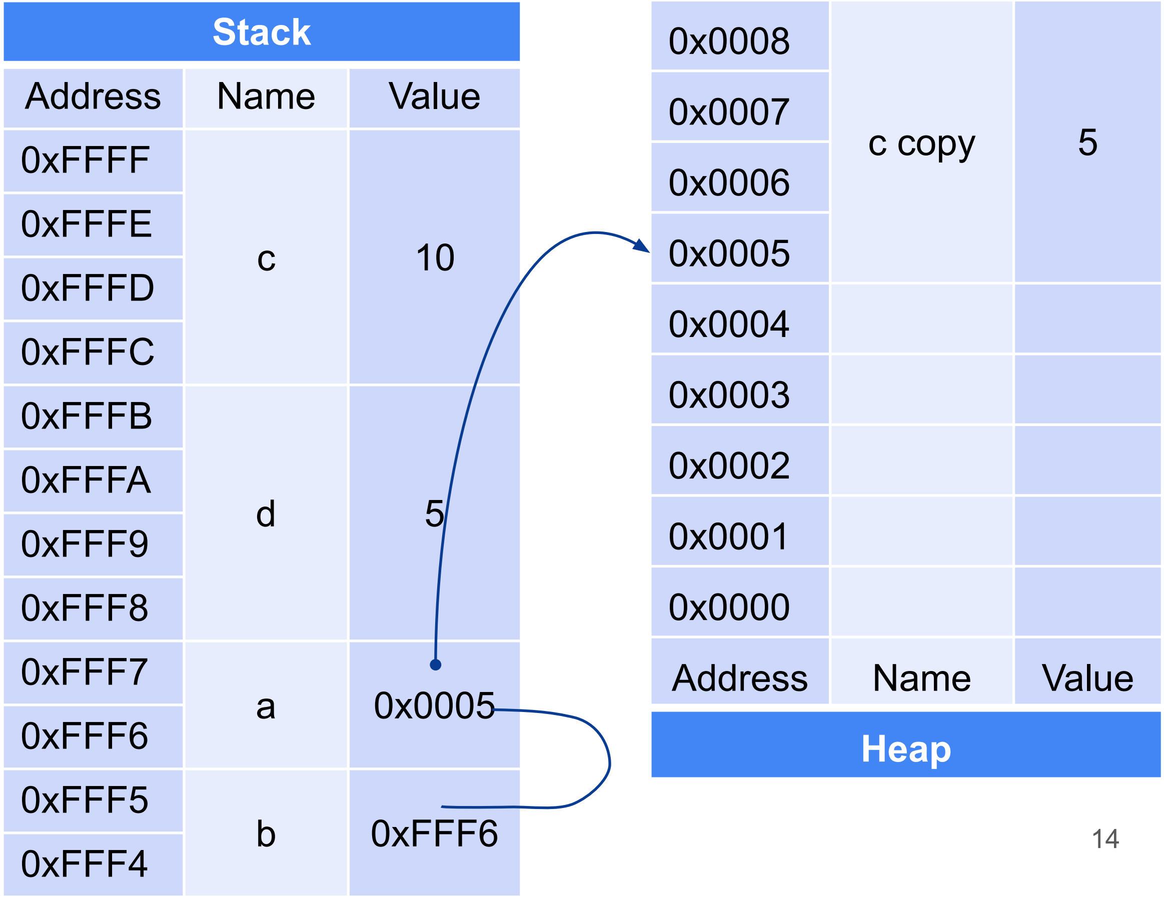
Heap

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.



```
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```



Stack

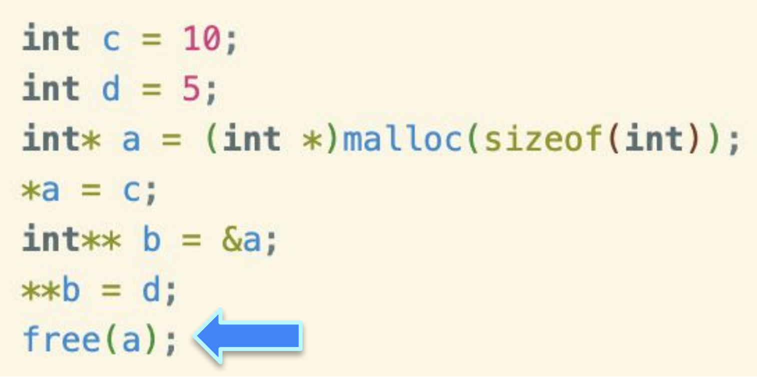
Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7	a	0x0005
0xFFF6		
0xFFF5	b	0xFFF6
0xFFF4		

Address	Name	Value
0x0008	c copy	5
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		

Heap

Exercise: Draw out the references between memory blocks.

Note: Assume all the addresses are of 2 bytes and the integer defaults to 4 bytes.



```
int c = 10;
int d = 5;
int* a = (int *)malloc(sizeof(int));
*a = c;
int** b = &a;
**b = d;
free(a);
```

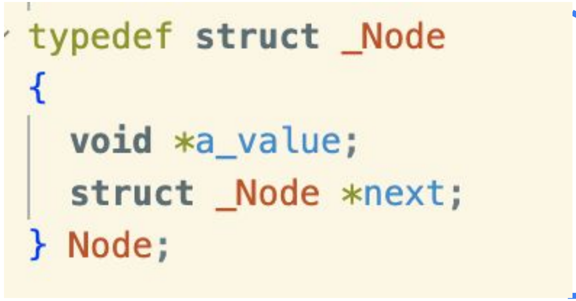
Stack		
Address	Name	Value
0xFFFF	c	10
0xFFFE		
0xFFFD		
0xFFFC		
0xFFFB	d	5
0xFFFA		
0xFFF9		
0xFFF8		
0xFFF7	a	
0xFFF6		
0xFFF5	b	0xFFF6
0xFFF4		

Address	Name	Value
0x0008	~~c copy~~	~~5~~
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		
Heap		

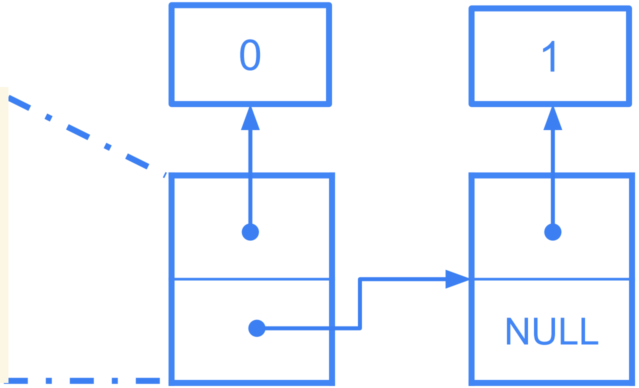
Linked List

Allocating space for more complex data structures

Consider the definition of a Linked List Node as follows:



```
typedef struct _Node
{
    void *a_value;
    struct _Node *next;
} Node;
```



Allocating space for more complex data structures

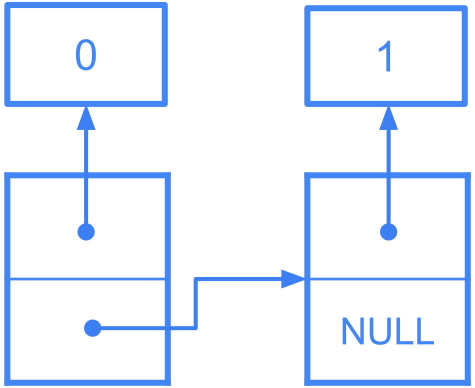
Exercise: Draw out the memory allocation for the tiny linked list.

Mind that an address takes up 2 bytes

```
Node *n0 = (Node *) malloc(sizeof(Node));
Node *n1 = (Node *) malloc(sizeof(Node));
int *v0 = (int *)malloc(sizeof(int));
int *v1 = (int *)malloc(sizeof(int));
*v0 = 0; *v1 = 1;

n0->a_value = (void *)v0;
n1->a_value = (void *)v1;
n0->next = n1;
n1->next = NULL;

free(n0->a_value);
free(n1->a_value);
free(n0);
free(n1);
```

Stack

Address	Name	Value
0xFFFF		
0xFFFE		
0xFFFD		
0xFFFC		



0x000A		
0x0009		
0x0008		
0x0007		
0x0006		
0x0005		
0x0004		
0x0003		
0x0002		
0x0001		
0x0000		
Address	Name	Value

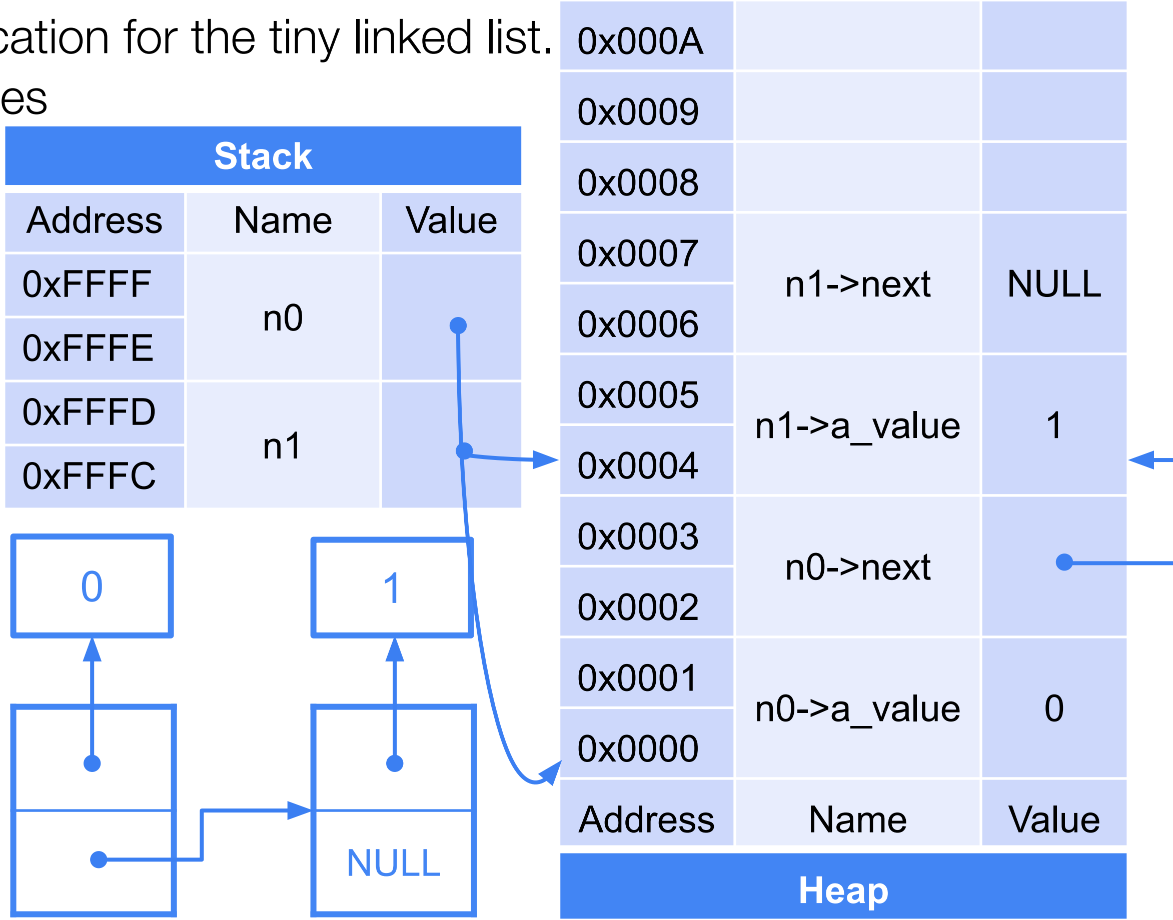
Heap

Allocating space for more complex data structures

Exercise: Draw out the memory allocation for the tiny linked list.

Mind that an address takes up 2 bytes

```
Node *n0 = (Node *) malloc(sizeof(Node));
Node *n1 = (Node *) malloc(sizeof(Node));
int *v0 = (int *)malloc(sizeof(int));
int *v1 = (int *)malloc(sizeof(int));
*v0 = 0; *v1 = 1;

n0->a_value = (void *)v0;
n1->a_value = (void *)v1;
n0->next = n1;
n1->next = NULL;

free(n0->a_value);
free(n1->a_value);
free(n0);
free(n1);
```



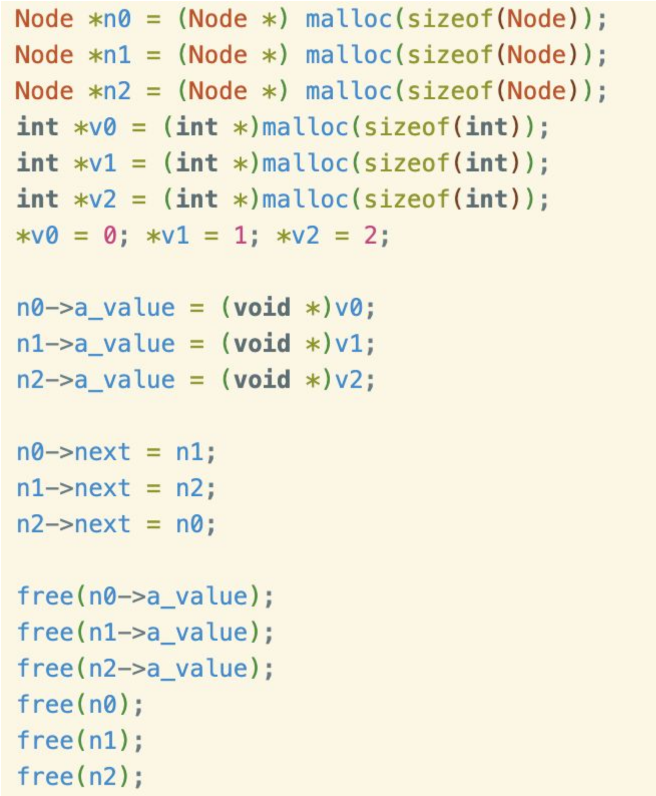
Stack

Address	Name	Value
0xFFFF	n0	
0xFFFE		
0xFFFD	n1	
0xFFFC		

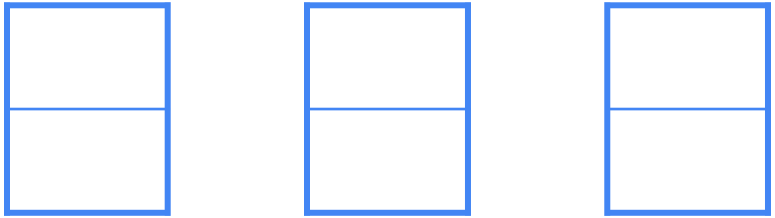
Address	Name	Value
0x000A		
0x0009		
0x0008		
0x0007	n1->next	NULL
0x0006		
0x0005	n1->a_value	1
0x0004		
0x0003	n0->next	
0x0002		
0x0001	n0->a_value	0
0x0000		

Heap

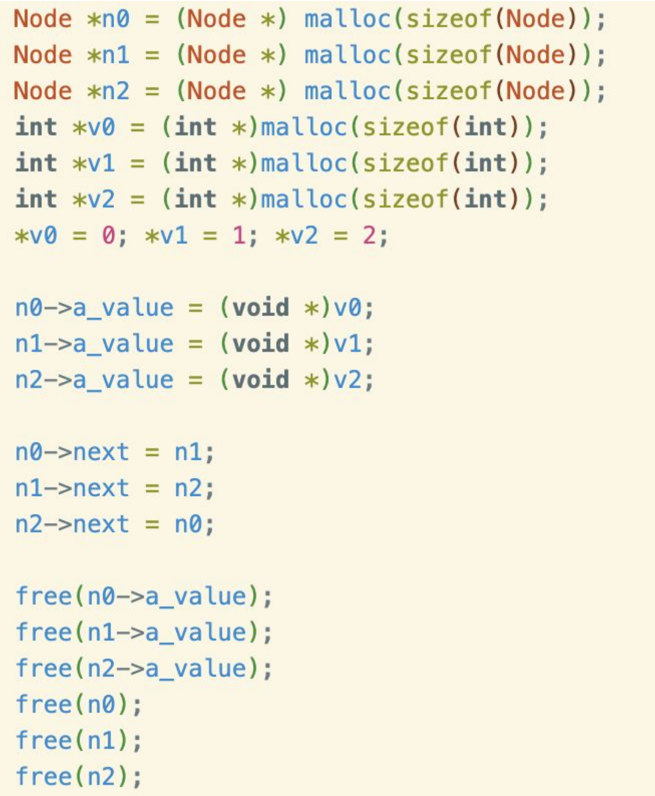
Exercise: draw the linked list created by the following code



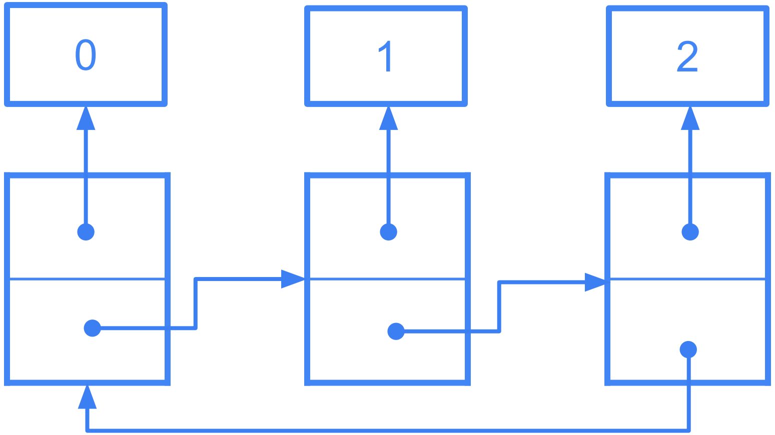
```
Node *n0 = (Node *) malloc(sizeof(Node));
Node *n1 = (Node *) malloc(sizeof(Node));
Node *n2 = (Node *) malloc(sizeof(Node));
int *v0 = (int *)malloc(sizeof(int));
int *v1 = (int *)malloc(sizeof(int));
int *v2 = (int *)malloc(sizeof(int));
*v0 = 0; *v1 = 1; *v2 = 2;

n0->a_value = (void *)v0;
n1->a_value = (void *)v1;
n2->a_value = (void *)v2;

n0->next = n1;
n1->next = n2;
n2->next = n0;

free(n0->a_value);
free(n1->a_value);
free(n2->a_value);
free(n0);
free(n1);
free(n2);
```



Exercise: draw the linked list created by the following code



```
Node *n0 = (Node *) malloc(sizeof(Node));
Node *n1 = (Node *) malloc(sizeof(Node));
Node *n2 = (Node *) malloc(sizeof(Node));
int *v0 = (int *)malloc(sizeof(int));
int *v1 = (int *)malloc(sizeof(int));
int *v2 = (int *)malloc(sizeof(int));
*v0 = 0; *v1 = 1; *v2 = 2;

n0->a_value = (void *)v0;
n1->a_value = (void *)v1;
n2->a_value = (void *)v2;

n0->next = n1;
n1->next = n2;
n2->next = n0;

free(n0->a_value);
free(n1->a_value);
free(n2->a_value);
free(n0);
free(n1);
free(n2);
```



Implement a linked list

Exercise: Implement the functionalities below in lab3.c

- `Node *list_create(void *a_value)` Initialize a Node containing the int value.
- `Node *list_push_to_front(Node *a_head, void *a_value)` Wrap the value in a node and make it the new head of the list.
- `Node *list_pop_last(Node *a_head)` Pop the last node of the list.
- `void list_free(Node *a_head)` Free the memory taken by the entire linked list.

Memory Management of the linked list

- Allocate node memory in `list_init` or `list_add_to_front`.
- To free your memory:
  - Remember to manually free all the nodes and the integer pointers that they contain after popping them from the list.
  - Before exiting your main, always free all the nodes left in the queue via `list_free`.

A3 Tips

Start early, start early, start early!

- This assignment has many intricacies. To give yourself enough time to test the functionality end-to-end, you'll want to start early.
- **Test thoroughly!** We've provided a unit-testing framework for you to use with test files already started for you. Add more tests as you need them to make sure your code is correct.
    - This applies to `huffman.c` as the complete test suite for Priority Queue is already given

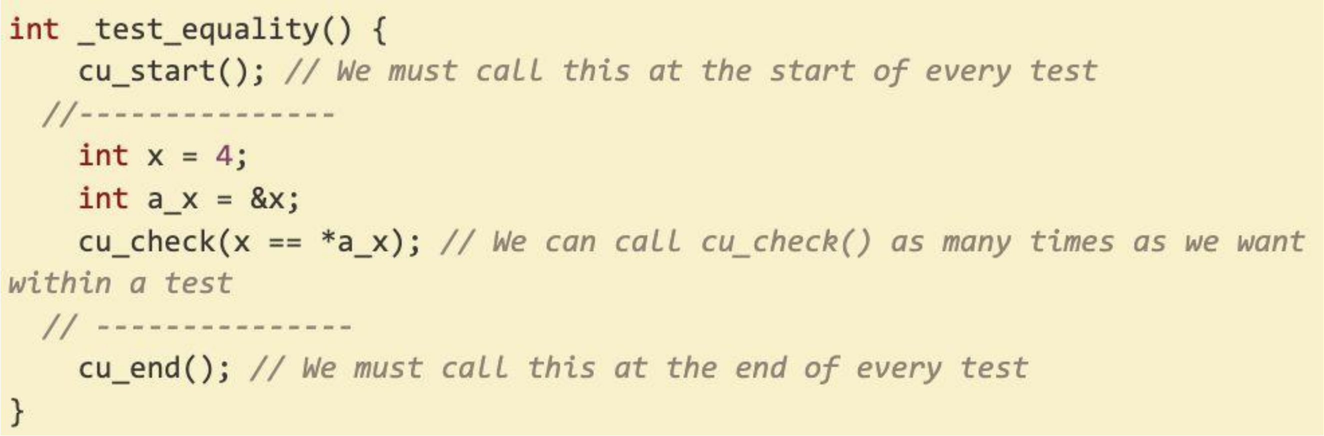
Hints

- In Task 1, you can implement `stack_push` by calling `pq_enqueue` and passing in `NULL` as the compare function. If you do this, you'll need to think carefully about your implementation for `pq_enqueue.`
- In Task 1, you'll also need to write a compare function to order the nodes in your priority queue. Nodes are sorted in *ascending* order, first by frequency, then by ASCII value. Follow this convention when implementing the function:
  - `cmp_fn(a, b) < 0  -> a` is ordered before `b`
  - `cmp_fn(a, b) >= 0 -> a` is ordered after `b`

  The `_cmp_int(...)` function implemented in `test_priority_queue.c` follows this too
- Freeing memory for a `TreeNode` is different from freeing memory for a `PQNode`.
- Use the functions in `utils.h` to print out priority queues. You can make your own custom print function for `TreeNode`.

Test your code!

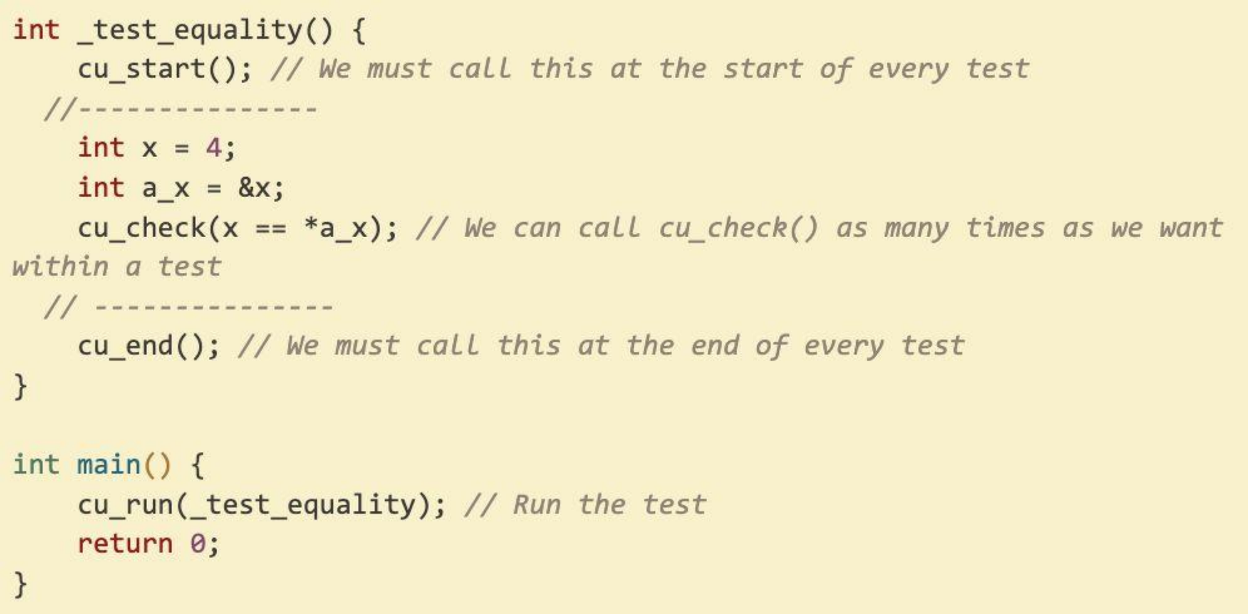
- Testing before you get to Task 2 will be crucial. We’ve provided you with a simple unit-testing library called `cu_unit`.
- The structure of a unit test is as follows:



```
int _test_equality() {
    cu_start(); // We must call this at the start of every test
  //---------------
    int x = 4;
    int a_x = &x;
    cu_check(x == *a_x); // We can call cu_check() as many times as we want
within a test
  // ---------------
    cu_end(); // We must call this at the end of every test
}
```

Test your code!

- We can run the test by adding it to `main()`:



```
int _test_equality() {
    cu_start(); // We must call this at the start of every test
  //---------------
    int x = 4;
    int a_x = &x;
    cu_check(x == *a_x); // We can call cu_check() as many times as we want
within a test
  // --------------
    cu_end(); // We must call this at the end of every test
}

int main() {
    cu_run(_test_equality); // Run the test
    return 0;
}
```

Test your code!

- Tests will not be graded for this assignment, but it’s a good idea to test your code before moving on to the next task, as we’ll be grading your code for each section individually.
- Test your priority queue with different types, comparison functions, etc.
- Test your Huffman tree on different files (C programs, for instance), and with a variety of characters.

Remember!

- **In A3 (not this lab), you must** use our wrapper functions `my_malloc()` and `my_free()` for any allocations or deallocations.
- They follow the exact same syntax as `malloc()` and `free()` and perform the same kind of allocation/deallocation.
- The only thing extra is that they log the operation, which will help you and us detect and pinpoint memory leaks in your implementation.

Good luck!