CSC111

Admin

- Assignment 5 released and is due March 13th.
  - Slightly longer/more difficult -> start on the earlier side
  - Asking you to use helper functions
    - print_array
      - Used to print integer arrays
      - You should create another to print double arrays
- Final Exam has been scheduled
  - April 19th at 09:00 – BWC B150

Pseudo Code – assignment tip

- Plain language description of the steps of an algorithm
- A good way to start implementing an algorithm
- Explains what needs to be done in the program
- Uses standard programming structures

Pseudo Code - Demo

Write a function that accepts array of student grades, and the passing mark for the class. The function should print out the max and minimum grades, in addition to the class average.

Debugging

- Don’t assume things work the way they’re meant to:
  - Determine what the output of the code should be, then manually execute the code and trace the variables.
  - Insert print statements in strategic places in your code to print out/track variables or to see if you reach that spot in the code
    - Remember you need to remove these!
  - Comment out code – reduce the problem area
  - Fix one issue at a time. Then recompile and test (and test, test, test)…

Arrays and pointers

Recall: declaring and initializing an array

- Declaring an array of a specified size
  - Allocates space for the array of given size, with initial garbage values

```
int my_ints[10];
double my_doubles[5];
```

- Declaring + initializing an array of a specified size
  - Allocates space for the array of given size, with initial values given {…}

```
int my_ints[3] = {1, 2, 3};
double my_doubles[2] = {5.6, 3.2};
```

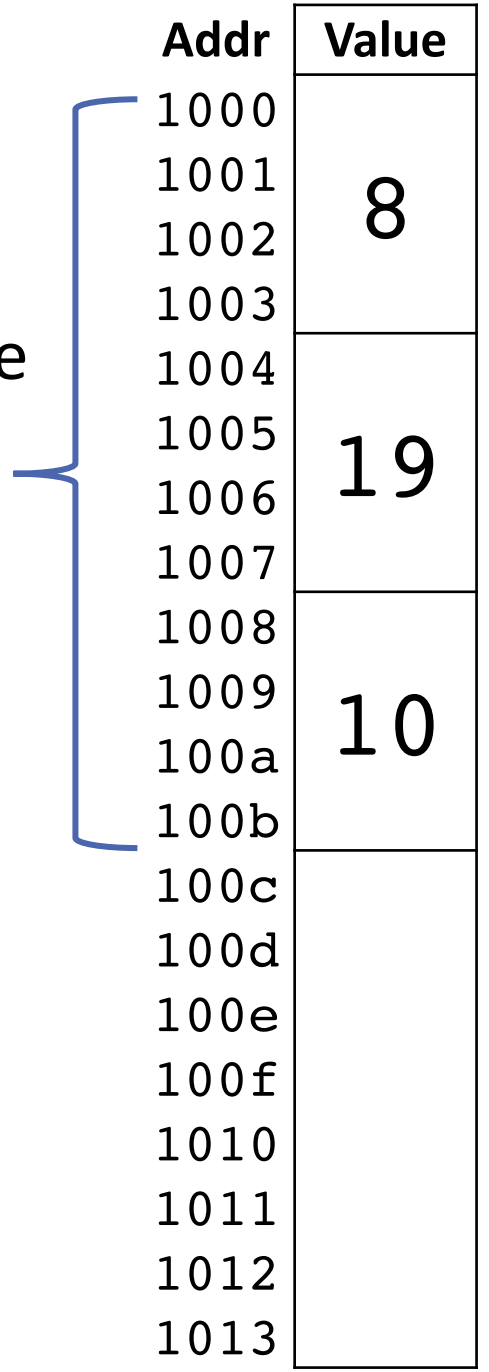
Array Access (and writing)

```
int my_ints[3] = {8,19,10};
//instead of duplicating code:
printf("value at index 0: %d\n", my_ints[0]);
printf("value at index 1: %d\n", my_ints[1]);
printf("value at index 2: %d\n", my_ints[2]);

//put the repeating code within the body of a loop:
int i;
int length = 3;
for(i=0; i<length; i++){
     printf("value at index %d: %d\n", i, my_ints[i]);
}
```

Arrays in memory…

- A contiguous block of memory is allocated
- space for values of specified type and size
- 3 integers here

```
int my_ints[3] = {8,19,10};

printf("%d", my_ints[0]); //8
printf("%d", my_ints[1]); //19
printf("%d", my_ints[2]); //10
```



Arrays are pointers…

```
int my_ints[3] = {8,19,10};
```

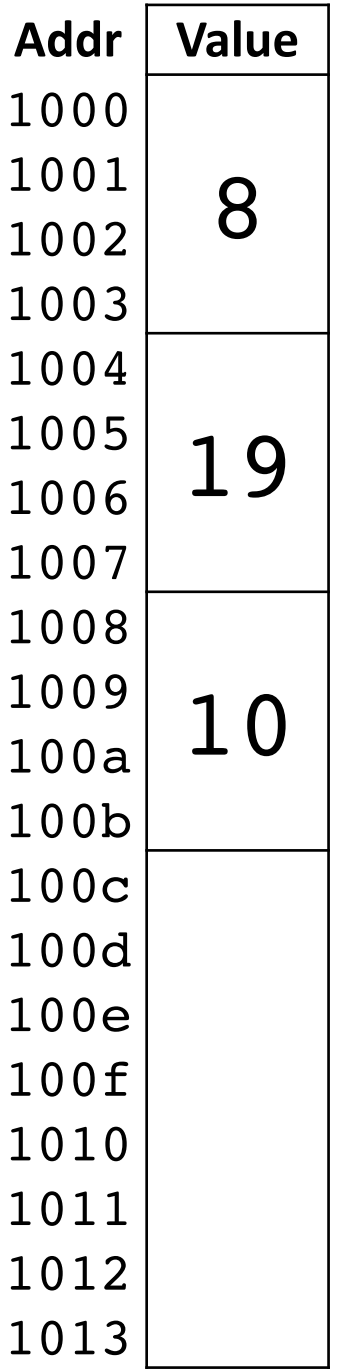
my_ints holds the address of the first array element

If memory was allocated as shown in the diagram here, the output of the following would be:

```
printf("%p", &my_ints[0]);
```
Output: 1000

```
printf("%p", my_ints);
```
Output: 1000

```
printf("%d", *my_ints);
```
Output: 8



Pointer arithmetic

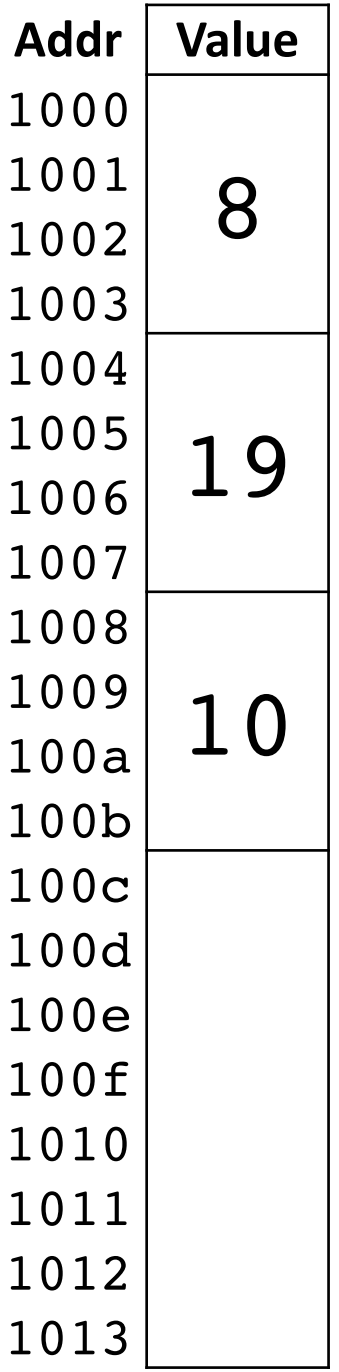
```
int my_ints[3] = {8,19,10};
```

If `my_ints` holds the value `1000` as illustrated in the diagram then `my_ints + 1` is `1004`

Pointer arithmetic takes into account that `my_ints` is an `int*` and adds 4 (the size of an `int`) for every 1 added

therefore, `my_ints + 2` is `1008`

and, `my_ints + 3` is `100c` (recall `c` is 12 in hexadecimal)



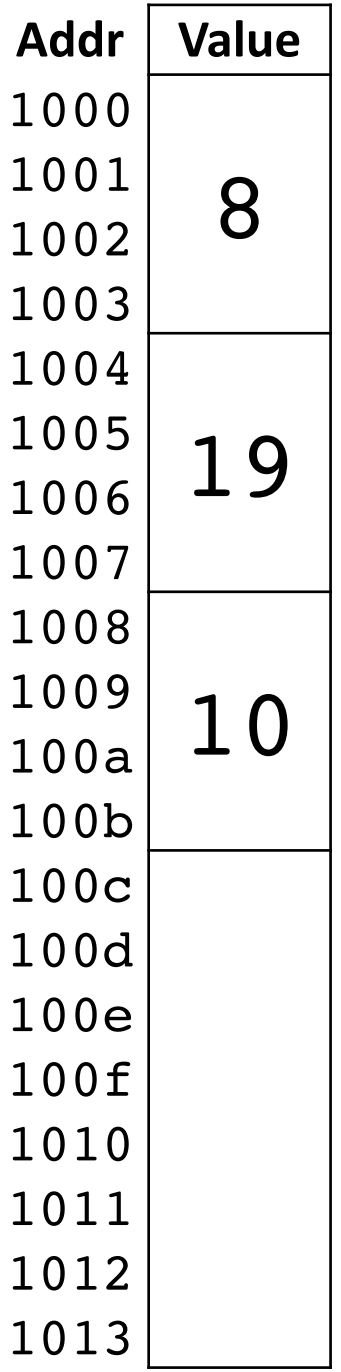
Dereferencing and pointer arithmetic

```
int my_ints[3] = {8,19,10};
```

`*my_ints` evaluates to 8

`*(my_ints+1)` evaluates to 19

`*(my_ints+2)` evaluates to 10



```c
#include <stdio.h>
void print_array(int numbers[], int length);

int main ( ) {
      int a4[4] = {4, 5, 6, 7};
      print_array(a4, 4);
      return 0;
}

/* Purpose: print the values in numbers
 * Parameters: int numbers[] – array of integers
 *             int length – number of elements in numbers
 */
void print_array (int numbers[], int length) {
      int i;
      for (i=0; i<length; i++) {
            printf("%d ", numbers[i]);
      }
}
```

RECALL: we can pass arrays as arguments to functions

```
// print_array function definition omitted for space
// assume it prints the values in the given array
void print_array(int numbers[], int length);
void double_vals(int numbers[], int length);

int main ( ) {
        int a3[3] = {4, 5, 6};
        print_array(a3, 3);
        double_vals(a3, 3);
        print_array(a3, 3);

        return 0;
}

void double_vals(int numbers[], int length) {
        int i;
        for (i=0; i<length; i++) {
                numbers[i] = numbers[i] * 2;
        }
}
```

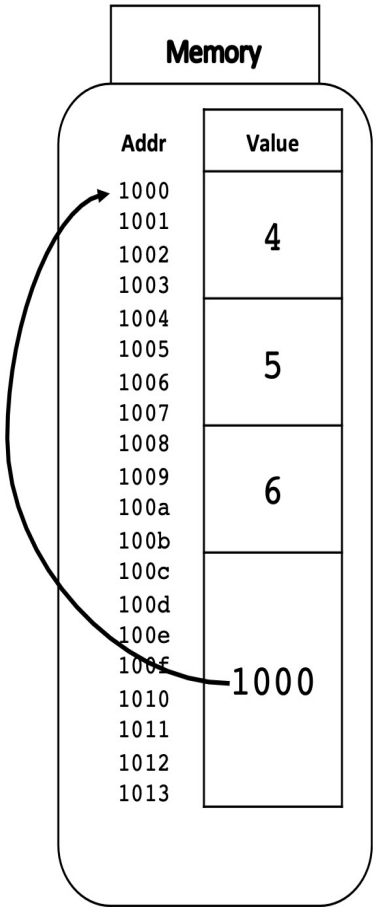
OUTPUT:

4 5 6

8 10 12

What is the output of the following code?

```
int a[3] = { 4, 5, 6 };
printf("a: %p, address of a: %p\n", a, &a);

int *ptr = a;
printf("ptr: %p\n", ptr);

printf("a[0]: %d, *a: %d, *ptr: %d\n", a[0], *a, *ptr);

ptr++;
printf("after ptr++: a:%p, ptr: %p\n", a, ptr);
printf("a[0]: %d, *a: %d, *ptr: %d\n", a[0], *a, *ptr);
```



Complete the function copy_non_neg

Complete the function copy_non_neg that takes an input array of integers, the size of the array and a second array the same size as a result array. The function should copy all the values from the input array to the result array but replacing the negative values with zeros. The function should NOT alter the input array

```
 //Example: data { 4, -3, 2, 7, 0, -1, 9 } results -> non_neg { 4, 0, 2, 7, 0, 0, 9 }

/* Purpose:  copies values in data to non_neg, replacing all negative values
 * with zero.
 * Params: int data[] – the input array,
 *         int size – the number of elements in data and capacity of non_neg,
 * int non_neg[] – the result array
 */
void copy_non_neg( int data[], int size, int non_neg[] ) {

}
```

Modify the pervious question to return the number of values replaced