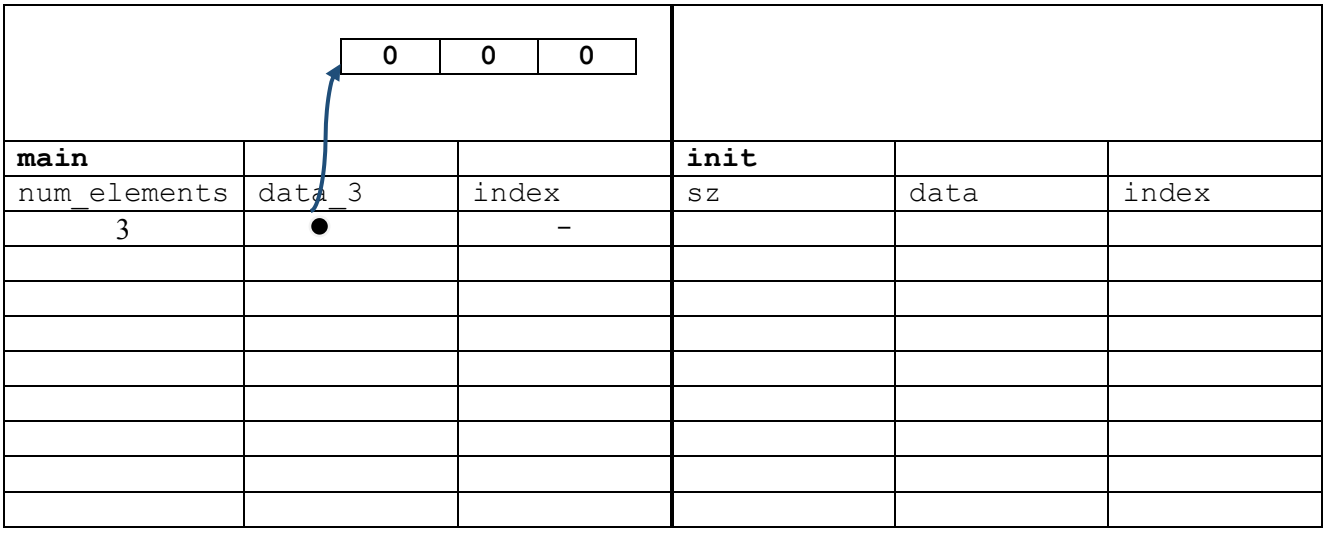
## Arrays and Pointers

1. Consider the following code segment:

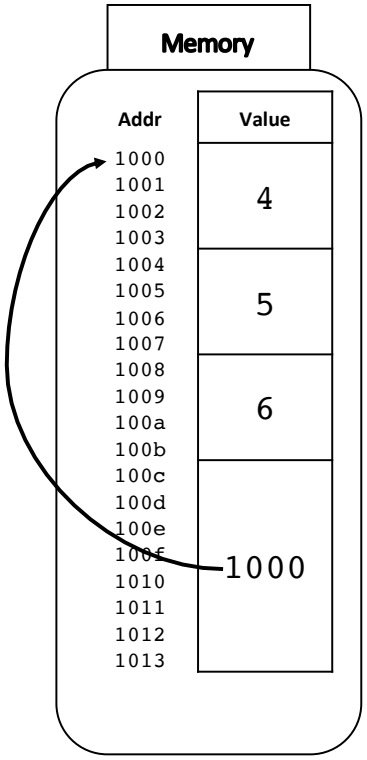
```
#include <stdio.h>
#define SIZE 3

void init( int data[], int sz );

int main( void ) {
    int data_3[SIZE] = { 0, 0, 0 };
    int num_elements = SIZE;
    int index;

    init( data_3, num_elements );

    printf( "Array elements: " );
    for( index = 0; index < SIZE; index++ ){
        printf( "%d ", data_3[ index ] );
    }
    printf( "\n" );

    printf( "num_elements: %d\n", num_elements );

    return 0;
}

/*
 * Purpose: initializes values in data in order from 0 to sz-1
 * Params: int data[], int sz, >=0 and < length of data
 * Returns: int – the count
 */
void init( int data[], int sz ) {
    int index;
    for( index = 0; index < sz; index++ ) {
        data[index] = index;
    }
}
```

Complete the trace table below to determine the values stored in the array data once the call to the function `init` has completed (it's not necessary to continue to trace the program after the call to `init` has ended).




| **main** | | | **init** | | |
|---|---|---|---|---|---|
| num_elements | data_3 | index | sz | data | index |
| 3 | ● | - | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

2. Consider the following code segment given the declarations and initializations of **a** and **ptr** are visually illustrated in this memory image:




```
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);

//a++; // compile error – cannot update a
```

a) What is the output?

b) Given the output, what are the differences/similarities between using the `[]` operator versus declaring and initializing a pointer to an array to access values in the array?

3. Design a **function** that takes an array of positive values of type `int` and the size of that array as its only parameters and multiplies every value in the array that is odd by 2.

4. 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                                    non_neg
{ 4, -3, 2, 7, 0, -1, 9 }               { 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[] ) {
```

5. Update the function `copy_non_neg` so that it only copies the values from the input array to the result array that are positive. The function should now return a count of the number of values that were copied to the result array. The function should NOT alter the input array.

   Example:

```
data                                    non_neg
{ 4, -3, 2, 7, 0, -1, 9 }               { 4, 2, 7, 0, 9 } // will return 5
```

**6.** Design the function `index_of_smallest` that takes an array of integers, the size of the array, and an integer `start_index` as its only parameters and that returns the index of the smallest item between index `start_index` and the end of the array.

So, for example, consider the following array:

```
int data[] = { 4, -1, 5, 8, 7, 2, 9 };
```

Here are some sample calls and the value returned (along with an explanation):

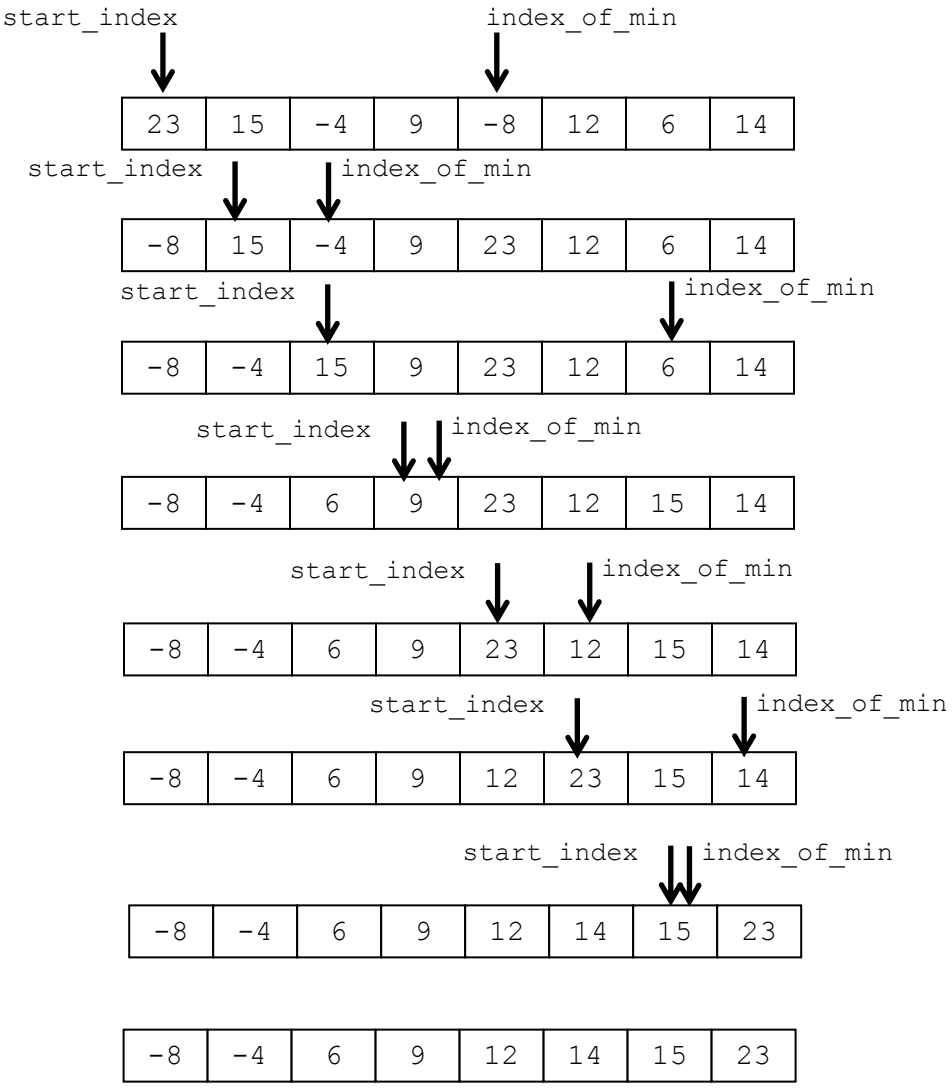
| sample call | return value | explanation |
|---|---|---|
| `index_of_smallest( data, 7, 0 );` | 1 | index of smallest value between index 0 (`start_index`) and the end of the array is 1 |
| `index_of_smallest( data, 7, 2 );` | 5 | index of smallest value between index 2 (`start_index`) and the end of the array is 5 |

Here's a start:

```
/* Purpose: Finds and returns the index of the smallest value in data
 *          between start_index and index (size - 1) inclusive.
 * Params: int data[], int size - number of elements in data, >=0
 *         int start_index - index to start looking for smallest, >=0, <size
 * Returns: int - the index of smallest
 */
int index_of_smallest( int data[], int size, int start_index ) {
```

7. Sorting data in an array is a very important application. For example, it's much easier (and more efficient) to search for an entry in an array if you know that the array is sorted. Imagine looking for a name in a phone book, for example, if the entries were not sorted alphabetically by name!
   In this question, we are going to implement a sort function using an algorithm known as *selection sort*. We assume that `size` represents the number of elements in the array:
   - for each `start_index` from `0` up to, but not including, `size - 1` do:
     - find the index of the smallest value between index `start_index` and the end of the array, call it `index_of_min`
     - swap the items at positions `start_index` and `index_of_min`

   Apply this algorithm to the array shown below to convince yourself that it works! The first step is done.




   Note that array is now sorted!

   Now write a function that implements the selection sort algorithm.
   Your solution to Question 6 plays an important role in the implementation of the selection sort algorithm – use it!

   We have got you started on the following page…

```
/*
 * Purpose: Sorts the entries of data in increasing order
 * Params: int data[], int size - number of elements in data, >=0
 */
void sort( int data[], int size ) {
```