Name: _____

CS 161 Week 7 Worksheet:

Pointers and Arrays

Pointers:

1. What is one difference between pointers and references?
   Pointers can be reassigned to a new memory address; references cannot.
   Pointers can be incremented or decremented.
   References must be initialized when declared; pointers do not have this requirement.
   To access the value that is stored, reference names are used directly but pointers must be dereferenced.

2. Predict the output of this code:

```
short* leaf = new short;
*leaf = -10;
cout << *leaf << endl;              -10

short* tree = new short[4];
tree[0] = *leaf;
tree[1] = *leaf + 1;
tree[2] = tree[0] * 20;
tree[3] = 15 - tree[1];
for (int i=0; i<4; i++) {
   cout << tree[i] << " ";          -10 -9 -200 24
}
cout << endl;

*tree += 2;
short* rock = &(tree[2]);
cout << *rock << endl;              -200
rock++;
cout << *rock << endl;              24
```

3. Write a C++ statement that uses `rock` (without changing `rock`) to change `tree[1]` to 15. (Think creatively!)

```
*(rock-2) = 15;
```

4. What `delete` statements should come after the above code segment to clean up the heap and avoid memory leaks?

```
delete leaf;
delete [] tree; /* or */ rock = tree; delete [] rock;
```

5. What happens if you `delete tree;` instead of `delete [] tree;` ?
   Memory is not properly released. valgrind will give an error : "Mismatched free() / delete / delete []".
   When using objects (instead of base types), e.g. string, you will get a runtime error ("invalid pointer").

1-D and Multidimensional Arrays:

1. What error was made in this program? How would you correct it?

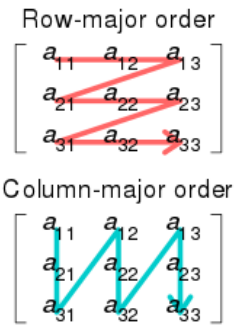
```
int* get_ducks() {
      int duck[10] = {};
      duck[3] = 47;
      return duck;
}

int main() {
      int* my_ducks = get_ducks();
      cout << my_ducks[3];
      int goose = 3;
      cout << my_ducks[3];
      return 0;
}
```

The duck array is created on the stack, local to the get_ducks() function. The g++ compiler will give this warning:
warning: address of local variable '**duck**' returned [-Wreturn-local-addr]
but will still compile. However, the memory is released when the function returns, so later allocations (like the goose variable) may overwrite the contents of my_ducks.
To fix it, allocate an array off the heap inside get_ducks():

```
int* duck = new int[10];
```

2. Define these terms:
   a. Multi-dimensional array
      An array of arrays
   b. Row-major
      Consecutive elements in each row are adjacent in memory
   c. Column-major
      Consecutive elements in each column are adjacent in memory
   d. Stride
      Number of items (or locations) that occur between successive array elements. e.g. for a 2D array in C++, the stride of the array is the size of the second dimension.



3. Give an example of something from the real world that could be modeled with a multi-dimensional array.
   2D: matrices, spreadsheet, minesweeper, battleship
   3D: multiple spreadsheet (x, y, z)
   4D (x, y, z, time) system
4. Does C++ use row-major or column-major array layout in memory?
   Row-major

5. Static vs. dynamic 2D arrays:
   a. How do you create each kind?
      Static:

```
int array[2][3];
```

      Dynamic:

```
int** array = new int*[2];
for(int i = 0; i < 2; i++)
     array[i] = new int[3];
```

b. Where are they located and how are they laid out in memory?

Everything for the static array is on the stack.
For 2D dynamic array, the double pointer is on the stack; the row pointers and elements are on the heap.

c. How do you pass each kind of array to a function?

```
int array_1[2][3];
int** array_2 = new int*[2];
for (int i = 0; i < 2; i++)
     array_2[i] = new int[3];
```

Static:

```
void pass(int a[2][3]);
void pass(int a[][3];
```

Dynamic:

```
void pass(int* a[], int row, int col);
void pass(int** a,  int row, int col);
```

6. Write a void function that will assign every element in a static 3 by 5 integer 2D array to 42. What parameters would we need for this function? How does the function prototype and function call differ between statically versus dynamically allocated arrays?

```
void fun(int array[][5]) {  // must have stride
   for (int i=0; i<3; i++)
      for (int j=0; j<5; j++)
         array[i][j] = 42;
}
void fun(int array[3][5]) {
   for (int i=0; i<3; i++)
      for (int j=0; j<5; j++)
         array[i][j] = 42;
}
```

Dynamically:

```
void fun(int* array[], int rows, int cols); /* or */
void fun(int** array,  int rows, int cols);
```

Function calls: Need to pass sizes for dynamic array: `fun(a, 3, 5); // where a is a int**`

7. Write a function to allocate memory for a 2D array in a function, given any number of rows and columns. Show an example of how to call your function with your favorite number of rows and columns. Show how to clean up afterwards to avoid a memory leak.

```
int** create_2D(int rows, int cols) {
   int** array = new int*[rows];
   for (int i = 0; i < rows; i++)
      array[i] = new int[cols];
   return array;
}
```

Function call:

```
int** array = create_2D(5, 10);
```

Deletion:

```
for (int i = 0; i < 5; i++)
   delete [] array[i]; // delete memory created by each row pointer
delete [] array;       // delete all row pointers
```