Name: _____

CS 161 Week 6 Worksheet:

References and Pointers, Memory Model, and 1-D Arrays

References vs. Pointers:

1. Explain these terms:
   - Reference (noun) – an alternative name (alias) that refers to an existing variable
   - Pointer (noun) – a variable that holds a memory address of a variable
   - Dereference (verb) – to access the value in the memory location contained by a pointer

2. Answer the following questions conceptually, then provide code examples.
   1) How is a reference created? How is a pointer created?

      Both are created by saying they will point/refer (hold an address) to a particular type

      ```
      int  i = 10;
      int* p = &i;  /* create a pointer to an integer */
      int& r =  i;  /* create a pointer to an integer */
      ```

   2) When and how can they each be assigned values?

      Must assign (initialize) a reference at the time it is declared (like a constant)

      ```
      int  i = 10;
      int* p;
      int& r;      /* show the compiler won't let you do this! */
      r = i;
      int& r = &i; /* fix to replace the above two lines */
      p = &i;
      ```

   3) Can you create a reference to a reference? A pointer to a pointer?

      Cannot make a reference to another reference, but you can make a pointer to another pointer

      ```
      int*  p;
      int&  r =  i;
      int** q = &p;  /* pointer to an integer pointer */
      int&& s =  r;  /* reference to a reference is not allowed */
      ```

   4) How do you modify the value that a reference refers to? How do you modify the value that a pointer points to?

      ```
      *p=20; cout << i << endl;
      r=50; cout << i << endl;
      ```

Memory Model

1. Define these terms:
   a. Compile-time (static) memory – memory set up during compile time (on stack)
   b. Runtime (dynamic) memory – memory allocated during runtime (on heap)
   c. Allocate – reserve memory for variable storage. In C++, use "new" to allocate dynamic memory during runtime. Static memory is allocated when a variable is declared.
   d. Deallocate – delete (free) memory. In C++, use "delete" to deallocate dynamic memory during runtime. You cannot (and need not) explicitly deallocate static memory.

2. Compare and contrast the stack and the heap.
   a. What they do? They both store data in memory.

   b. Where are they positioned in computer memory? They start at opposite ends in memory and grow toward each other.

```
int* p = new int;
cout << &p << "   " << p << endl;
int* q = new int;
cout << &q << "   " << q << endl;
```

   c. What happens if the stack gets too large? Stack overflow.

   d. What happens if the heap gets too large? Out of memory error (malloc: can't allocate).

   e. Can variables on the stack or the heap both be deleted? Cannot delete off stack but you can off heap

```
int* p = &i;
delete p;   /* Attempt to free memory at address in p,
             * which is the address of i, which is on the stack.
             * Not allowed. */
int* p = new int;
delete p;   /* Free memory stored at address in p,
             * which is on the heap, which is allowed. */
```

3. What is a memory leak, and how do you prevent them?
   When you have memory allocated on the heap that was never freed.

```
int* p = new int;
p = new int;
delete p; /* only deletes the second new int, not the first one */
```

   Run valgrind to show two created and only one deleted.

4. How do you create 1-D arrays on the stack? On the heap?
   1-D arrays can be created on the stack or the heap.

```
int a[10];              /* array of integers on the stack */
int* a2 = new int[2];   /* array of integers on the heap */
a2[0] = 10;
a2[1] = 20;
delete [] a2;           /* delete array that a2 points to */
```

   Use valgrind to show the difference in strings vs. int arrays if you omit the [] with delete.

```
string* s2 = new string[2];   /* array of strings on heap */
```

1-D Arrays

5. Write a function called "get_array()" that takes in a size, then return an array of integers on the heap of that size. Consider what return type this function should have. Show how you would call it from main().

```
int* get_array(int size) {
      int* arr = new int[size];  /* allocate on the heap */
      return arr; /* returns address of (first item of) the array */
}

int main() {
      int n = 0;
      cin >> n;
      int* arr = get_array(n);
      delete [] arr;
      return 0;
}
```

6. Why wouldn't it work to instead allocate the array on the stack in the function above?
   If we create the array on the stack, then once we leave the function, the stack array will be automatically deleted, since the stack memory is deleted once out of scope.

7. Looking at the function we made in question 5, could we have made this function a void function? What are your two options in C++? Write both functions and function calls.

```
void get_array1(int** arr, int size) {
      *arr = new int[size];
}
void get_array2(int*& arr, int size) {
      arr = new int[size];
}

int main(){
      int* arr = NULL, n = 0;
      cin >> n;
      get_array1(&arr, n); /* pass the address of the pointer */
      get_array2(arr, n);  /* pass the pointer by reference */
      return 0;
}
```

C-style strings vs. C++ strings

8. What are the differences between a C++ and C-style string?
   - C++ string: object with size and vector member; C-style string: character array
   - C-style string: must end in \0 (null)
   - C++ string: declared as type "string"; C-style string: declared as type "char*"
   - C++ string: include <string>; C-style string: include <cstring/string.h> libraries
   - C++ string: overloaded assignment and relational operators; C-style string: strcpy() and strcmp() functions
   - C++ string: s.length(); C-style string: strlen(s)

9. How does the declaration for a function differ when using these two types of strings? How does the declaration for a C++ or C-style string argument change if you want to modify the contents of the string in the function?
   - `void fun(string s);` vs. `void fun(char *s)` or `void fun(char s[])`
   - To modify, pass the string by reference: C++ string: `void fun(string& s)`; C-style string declaration doesn't change (why?)

10. How do you make an array of C++ strings? How would you make an array of C-style strings?
   - C++ string: `string s[10];` or `string* s;` to create a dynamic number of C++ strings
   - C-style string: `char s[10][10];` or `char** s;` to create a dynamic number of C-style strings with a dynamic number of characters for each C-style string.