Problem 1.5-1.6 Report File

Alex Helfrich

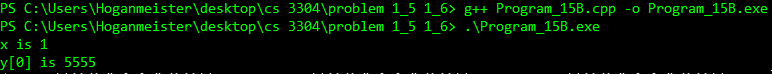
CS 3304 Section 04

For the first program, 1.5A, the code initializes an array with 5 numbers, then uses a method, called printArray, to print the items in the list by using a loop to print each out individually.

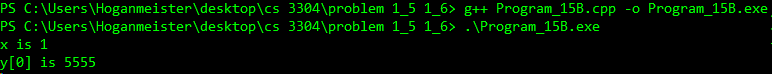


The second program shows that passing an int to a method creates a new int (i.e. writing m(x) in main and having a method m(int y) will create a new int y, thus not changing the value of the integer x in the main method) but passing an array will actually pass the address for the array, instead of creating a new one. Therefore, when a number is changed in the array numbers in the method m(), it actually changes the corresponding spot in y[], back in the main method.

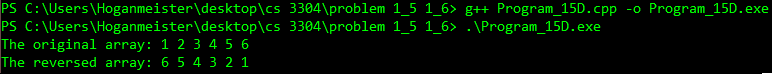
Main creates an int x = 1, and int y[10], and sets y[0] = 1; Then the function m() is called, passing x and the address for y. In m, y[0] == numbers [0], and numbers[0] is set to 5555. Then, when the main method prints out x and y[0], the output is x is 1 and y[0] is 5555.



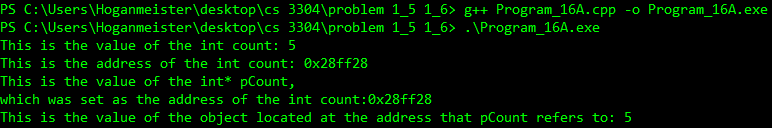
Problem 1.5C shows that trying to modify a constant integer array, or any constant, can’t be done as the object is now read only. Trying to compile comes up with an error: assignment of read-only location ‘\* list.’



Program 1.5D makes use of a reverse function and a printArray function to show how to use loops and array sizes to modify an array to be backwards. The method reverse() takes a input array and prints out the backwards form to a given empty array, both of which are then printed out in the main method using printArray().



Program 1.6A creates a int count = 5 and a int\* pCount = &count. This means that there is an int object of value 5 at the address of &count, and there is a pointer pCount that has the value of &count, thus pointing at the int count. The output of the main() method gives the values of count, &count, pCount, and \*pCount, which are an int, address, address, and int respectively.



Program 1.6B shows different ways to swap the values of variables. The first swap function does nothing because the original variables num1 and num2 aren’t given. Rather, the exact values are given. The second swap function fixes this problem by creating references int& n1 and n2 that reference addresses of num1 and num2. That way, when the swapping occurs, the addresses of num1 and num2 are switched, meaning that num2 has the same address num1 used to, and vice versa. The third swap function creates pointers int\* p1 and p2 that point to num1 and num2. Then, in the method, the actual value of num1 and num2 are found using \*p1 and \*p2, and those values are switched. The fourth swap function takes p1 and p2 from the main method, which are pointers that have been given the address of num1 and num2 using the & modifier. The address of p1 and p2 are then given to the swap4() method, by writing int\* &p1, int\* &p2, and then swapped using a int\* temp. Now the addresses of p1 and p2 in the main() method have been switched.

