Project 6 – Joyce Chen

**1a.**

int main() {

int arr[3] = { 5, 10, 15 };

int\* ptr = arr;

\*ptr = 30;

\*(ptr + 1) = 20;

ptr += 2;

ptr[0] = 10;

ptr = arr;

while (ptr <= arr + 2)

{

cout << \*ptr << endl;

ptr++;

}

}

**1b.**

The mistake here is that the pointer passed into the findMax function won’t be changed, since pointers are default pass by value. Therefore, when you call the pointer in the main function after findMax, the pointer is still pointing at the 1st element in the array, unchanged. In order to fix this bug, all we have to do is pass the pointer by reference using the ampersand (&) symbol. The function header should look like **void findMax(int arr[], int n, int\*& pToMax)** instead.

**1c.**

The main function will not work because the pointer passed into computeCube is uninitialized, so an error (or undefined behavior) would occur when computeCube tries to dereference it. One way to fix this is to initialize the pointer to the address of an int variable. The computeCube function would now be able access the value of the int variable when dereferencing the pointer, changing it to 125.

int main() {

int num;

int\* ptr = &num;

computeCube(5, ptr);

cout << "Five cubed is " << \*ptr << endl;

}

**1d.**

The first problem is that the while loop condition checks whether the pointers str1 and str2 were zero bytes. It should instead be checking whether the values inside the C strings were zero bytes, which can be fixed simply by dereferencing the str1 and str2 pointers with an asterisk (\*). The second problem is similar, in that the if statement was comparing the pointers str1 and str2, instead of the characters they pointed to. To access the characters in the C string, we’d need to dereference the pointers with the asterisk again. The final problem happens in the return statement, where it compares whether the two pointers str1 and str2 were equal. However, this will always return false because the pointers will never point to the same address, as they’re pointing to different char arrays in the first place. It would make more sense to check whether the characters they point to are equal to each other, because the while loop would stop executing once we hit the zero byte in either of the arrays. So \*str1 == \*str2 would only return true if we reach the zero byte in both arrays, which works since that indicates we’ve reached the end of both strings. Otherwise, it’d return false if one string reaches the zero byte earlier than the other, which correctly implies that the strings are not equal.

bool strequal(const char str1[], const char str2[])

{

while (\*str1 != 0 && \*str2 != 0)

{

if (\*str1 != \*str2)

return false;

str1++;

str2++;

}

return \*str1 == \*str2;

}

**1e.**

The program seems to be deleting the memory allocated to the array *anArray* declared in *getPtrToArray*, and setting junk values to the original memory slots of the array elements. Once a function ends, any local variables declared inside the function goes away. When I try to access the value that *ptr* points to the first time, it returns the correct value but after that it returns garbage values. For instance, when I print out *ptr[0]* right after the initialization of *ptr*, it gives me the right value of 100, but when I print *ptr[0]* again, it returns a random value. Furthermore, while the f() function call seems like it’s doing nothing in particular, it’s actually allocating junk values to memory slots originally occupied by *anArray* (specifically the first memory slot). One potential fix to this is to make *anArray* a global array, instead of a local one inside the function so that its memory allocation is fixed.

**2a.** double\* cat;

**2b.** double mouse[5];

**2c.** cat = mouse + 4;

**2d.** \*cat = 25;

**2e.** \*(mouse + 3) = 54;

**2f.** cat -= 3;

**2g.** cat[1] = 42;

**2h.** cat[0] = 17;

**2i.** bool d = (cat == mouse);

**2j.** bool b = (\*cat == \*(cat + 1));

**3a.**

double mean(const double\* scores, int numScores)

{

const double\* ptr = scores;

double tot = 0;

for(int i = 0; i < numScores; i++)

{

tot += \*(ptr + i);

}

return tot/numScores;

}

**3b.**

const char\* findTheChar(const char\* str, char chr)

{

for (int k = 0; \*(str + k) != 0; k++)

if (\*(str + k) == chr)

return str + k;

return nullptr;

}

**3c.**

const char\* findTheChar(const char\* str, char chr)

{

while (\*str != 0){

if (\*str == chr){

return str;

}

str++;

}

return nullptr;

}

**4.**

The program first prints out 3 and a newline. In the first half of the code, a pointer is initialized to the address of the first element in the array after the maxwell function call, because \*array (aka &array[0]) is 5 and &array[2] is 4 and 5 > 4. The first element of the array is changed to -1, and the pointer then increments by 2 to point to the 3rd element in the array. The 4th element of the array is set to 9, and the 2nd element is set to 79. Since the pointer is located at the 3rd element, and &array[5] is a pointer to the 6th element, 6-3=3 which gives us the first output of 3.

The swap1 function call didn’t change the array’s contents because pointers are default pass by value, and it was attempting to swap the pointers within the function. The original pointers would not have been changed after the function call since the function is just swapping copies of the pointers from the parameter.

Meanwhile, the first and third elements of the array are swapped due to the swap2 function call; *array* is a pointer to the first element, and *&array[2]* is a pointer to the third element in the array. The swap2 function call sets a local variable temp to the value of the first element (by dereferencing *array*), and then sets *array* to the value that *&array[2]* points to. The value that *&array[2]* points to is then set to the local temp variable, finishing the swap. The program then prints out the elements in the array, each on a new line: 4, 79, -1, 9, 22, 19.

Final output:

3

4

79

-1

9

22

19

**5.**

void removeS(char\* s){

char \*i = nullptr;

while(\*s != 0){

if(\*s == 'S' || \*s == 's'){

for(i = s + 1; \*i != 0; i++){

\*(i - 1) = \*i;

}

\*(i - 1) = 0;

s--;

}

s++;

}

}