1.

a. **int** main() {

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

**int**\* ptr = arr;

\*ptr = 10; // set arr[0] to 10

\*(ptr + 1) = 20; // set arr[1] to 20

ptr += 2;

ptr[0] = 30; // set arr[2] to 30

**while** (ptr >= arr)

{

cout << ' ' << \*ptr; // print values

ptr--;

}

cout << endl;

}

b. p is passed by value to the function, so any changes made to p in the function will not be applied in main. ptr will always point to the nums[0]. To fix this, pass p by reference, changing int\* p to int\* &p.

void findDisorder(int arr[], int n, int\* &p)

{

for (int k = 1; k < n; k++)

{

if (arr[k] < arr[k-1])

{

p = arr + k;

return;

}

}

p = nullptr;

}

c. p does not point to anything, therefore when it is passed to the function the function does not know where to store the result of the arithmetic expression, causing undefined behavior. To fix this, declare a double in main before the pointer (ex.double g;). Then set p to point to g (double\* p = &g;). Now the function will store the result in g.

int main()

{

Double g;

double\* p = &g;

hypotenuse(1.5, 2.0, p);

cout << "The hypotenuse is " << \*p << endl;

}

d. First, in the while loop, str1 and str2 are pointers to each C string, so it does not make sense to check if they are 0. Instead we want to check if that each character is not equal to 0, and break out of the loop when we do hit the 0 byte. We fix this by checking that the character that str1 and str2 point to are not 0.

Second, in the if statement we are again checking if str1 and str2 are equal, when they are pointers to the next character in each C string. We should be checking if the corresponding characters are equal. We fix this by instead comparing the characters that str1 and str2 point to.

Third, in the return statement, we are checking again if the str1 and str2 are equal, when they are pointers. We should instead check if the the chracters they point to are the same. If they ended at the same time, the pointers should both point to the 0 byte.

**bool** match(**const** **char** str1[], **const** **char** str2[])

{

**while** (\*str1 != 0 && \*str2 != 0) // zero bytes at ends

{

**if** (\*str1 != \*str2) // compare corresponding characters

**return** **false**;

str1++; // advance to the next character

str2++;

}

**return** \*str1 == \*str2; // both ended at same time?

}

**int** main() {

**char** a[10] = "Ding";

**char** b[10] = "Dong";

**if** (match(a,b))

cout << "They're the same!\n";

}

e. arr is declared inside the function so its scope only extends in the function. Hence, after the function call ptr points to an address in memory that is no longer reserved for arr and may be overwritten.

2.

a. string\* fp;

b. string fish[5];

c. fp = fish+4;

d. \*fp = "yellowtail";

e. \*(fish + 3) = "salmon";

f. fp = fp - 3;

g. fp[1] = "carp";

h. fp[0] = "smelt";

i. **bool** d = (fp == &fish[0]);

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

3.

a. **double** computeAverage(**const** **double**\* scores, **int** nScores)

{

**const** **double**\* ptr = scores;

**double** tot = 0;

**for** (**int** i = 0; i < nScores; i++){

tot += \*(ptr+i);

}

**return** tot/nScores;

}

b.

**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**;

}

c.

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

{

**while** ( \*str != 0){

**if** (\*str == chr){

**return** str;

}

str++;

}

**return** **nullptr**;

}

4. It prints

**diff=1**

**4**

**79**

**5**

**9**

**-1**

**19**

Line by line this is what happens in the main function:

1- array is initialized to 5, 3, 4, 17, 22, 19

2- Minimart is called, and returns b/the address of array[2]. This is because array points to the 0th element of the array (5) which is not less than the array[2] (4). ptr is set to point to array[2].

3- ptr[1], which is the same as array[3], is set to 9. array is now 5, 3, 4, 9, 22, 19.

4- ptr is set to point 2 places ahead, or to array[4].

5- the element ptr points to, array[4] is set to -1. array is now 5, 3, 4, 9, -1, 19.

6-The element that array+1 points to, which is array[1], is set to 79. array is now 5, 79, 4, 9, -1, 19.

7- Prints diff=1 and new line. ptr points to array[4] so the address of array[5] minus ptr is 1, since we do pointer arithmetic in units of pointers.

8-swap1 function is called on pointer to array[0] and array[1]. The swap1 function does not change the values of array at all. It does swap the local variable pointers a and b, but these pointers are destroyed after the function call anyways, so essentially no changes are made.

9-swap2 function is called on pointer to array[0] and array[2]. The swap2 function does swap the values of array[0] and array[2]. This because the function refers to the elements a and b point to when it swaps, rather than the pointers, and temp holds the int value, rather than pointer. array is now 4, 79, 5, 9, -1, 19.

10- the for loop prints out the values of array, with a new line after each element.

5. **void** deleteG(**char**\* ptr){

**char**\* ptr2 = ptr;

**while** (\*ptr != 0){

**if** (\*ptr == 'g' || \*ptr == 'G'){

ptr++;

**continue**;

}

\*ptr2 = \*ptr;

ptr++;

ptr2++;

}

\*ptr2 = 0;

}