1. a.

int main() {

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

int\* ptr = arr;

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

ptr++;

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

ptr++;

\*ptr = 30; // set arr[2] to 30

while (ptr >= arr)

{

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

ptr--;

}

cout << endl;

}

b. Since the pointer p is not passed by reference into the function, when the function changes what the pointer points to, the pointer ptr in the main function is unchanged. To fix this we pass the pointer by reference. This is shown in the following, printed from xcode.

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. The pointer p is not pointing to anything when it is called into the function, thus when the function is called, it does not know where to store the value that is found. The way to fix this is to create a double that the pointer can point to, shown below.

#include <iostream>

#include <cmath>

using namespace std;

void hypotenuse(double leg1, double leg2, double\* resultPtr)

{

\*resultPtr = sqrt(leg1\*leg1 + leg2\*leg2);

}

int main()

{

double answer = 0.0;

double\* p = &answer;

hypotenuse(1.5, 2.0, p);

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

}

d. Much of what this code tries to do does not make sense. For example, the incrementation of str1 and str2, which is not possible. Also comparing arrays using == and != does not make sense. Also, rather than using the zero byte for each of the C strings, the code just uses the integer 0. We can use pointers to fix the implementation of this function, as is shown below.

// return true if two C strings are equal

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

{

const char \*pstr1 = str1;

const char \*pstr2 = str2;

while (\*pstr1 != '\0' && \*pstr2 != '\0') // zero bytes at ends

{

if (\*pstr1 != \*pstr2) // compare corresponding characters

return false;

pstr1++; // advance to the next character

pstr2++;

}

return \*pstr1 == \*pstr2; // 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. When the program outputs the values of ptr, the values are seemingly nonsense. The reason this is is because the program is trying to find values that no longer exist at this point in the code. When ptr is initialized, it is set equal to the memory address of what the function computeSquares returns. However, the function returns an array that is local to the function, which stops existing once the function is finished running. Therefore, when the main function tries to call values at the points that ptr is pointing to, is gives random memory addresses, as there is no longer a value that exists there.

2.

a. string \*fp;

b. string fish[5];

c. fp = &fish[4];

d. \*fp = “yellowtail”;

e. \*(fish+3) = “salmon”;

f. fp -= 3;

g. fp[2] = “carp”;

h. fp[] = “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;**

**int i = 0;**

**while (i < nScores)**

**{**

**tot += \*(ptr+i);**

**i++;**

**}**

**return tot/nScores;**

**}**

b.

// This function searches through str for the character chr.

// If the chr is found, it returns a pointer into str where

// the character was first found, otherwise nullptr (not found).

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

{

const char\* p = str;

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

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

return (p+k);

return nullptr;

}

c.

// This function searches through str for the character chr.

// If the chr is found, it returns a pointer into str where

// the character was first found, otherwise nullptr (not found).

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

{

for (;\*str != '\0'; str++)

if (\*str == chr)

return str;

return nullptr;

}

4. The output of the code is the following:

diff=1

4

79

5

9

-1

19

The first line is diff=1 because of the line that has

cout << "diff=" << &array[5] - ptr << endl;

-At this line, ptr has a value of 4, while &array[5] has a value of 5. When we subtract the two we get 1.

Rather than explain each line of output, I will explain why the array ends with these values in these positions, as the last lines simply print each value of the array.

The first line in the main function creates the array in the order see in the code.

The next line creates the ptr pointer, which is given the value of the result of the function minimart, with the parameters seen. This function returns the address of the 4 in the array, meaning that ptr now points to 4.

The next line sets the value at the position after ptr (which is 17 at index 3 at this point) to 9.

Ptr is then incremented by 2, and the value at this address (index 4 in the array) is given a new value of -1. The array is now: {5, 3, 4, 9, -1, 19}

The next line sets the value at (array+1) to 79, which at this point is the value 3 in the array index 1.

Next, swap1 is called, which swaps the pointers that point to array[0] and array[1] (4 and 79).

Then, swap2 is called, which swaps the values at array and array[2] (4 and 5).

The array now reads {4, 79, 5, 9, -1, 19}.

Each value is printed and the line ends, resulting in the output we see above.

5.

void deleteG (char c[]){

char\* pointer = c;

for (;\*pointer != '\0'; pointer++)

{

if (\*pointer == 'G' || \*pointer == 'g')

{

for (;\*pointer != '\0'; pointer++)

{

\*pointer = \*(pointer+1);

}

pointer = c;

}

}

}