1a)

For starters, incorrect method of accessing array values via pointers. Also didn’t set the pointer to the memory address of the last value of the array, which makes iterating over it backwards difficult if not impossible.

#include <iostream>

using namespace std;

int main() {

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

int \*ptr = arr;

\*ptr = 10;

\*(ptr + 1) = 20;

\*(ptr + 2) = 30;

ptr = &arr[2];

while (ptr >= arr) {

cout << " " << \*ptr--;

}

cout << endl;

}

1b)

The pointer was being passed as a copy, ensuring that any modifications done to the pointer in the function scope wouldn’t be reflected once it returned to the main function.

void find\_disorder(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;

}

1c)

The pointer isn’t pointing to an actual memory location, it’s just there. We need to create an actual var which the pointer will point to.

int main() {

double p;

double \*p\_ptr = &p;

hypotenuse(1.5, 2.0, p\_ptr);

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

return 0;

}

1d)

When passing an array into a function, using the actual argument name is just using the entire array, so if you want to actually iterate over the array to check its contents, you either need to iterate it’s index via ‘arr[i]’ or you need to use the proper syntax such as ‘\*arr’. Otherwise you really aren’t doing anything.

bool match(const char s1[], const char s2[]) {

while (\*s1 != 0 && \*s2 != 0) {

if (\*s1 != \*s2) {

return false;

}

\*s1++;

\*s2++;

}

return \*s1 == \*s2;

}

1e)

From first glance, it seems like its attempting to return the memory address of a variable which has gone out of scope and thus destroyed. So it’s literally pointing to nothing.

2)

a) string \*fp;

b) string fish[5];

c) fp = &fish[4];

d) \*fp =”yellowtail”;

e) \*(fish + 3) = “mackerel”;

f) fp = fp - 3;

g) fp[1] = “code”;

h) fp[0] = “eel”;

I) bool d = \*fp == fish[0] ? true : false;

j) bool b = \*fp == \*(fp + 1) ? true : false;

3a)

double compute\_average(const double \*scores, int n\_scores) {

const double \*ptr = scores;

int count = 0;

while (count != scores + n\_scores) {

tot += \*(ptr + count);

count += 1;

}

return tot/n\_scores;

}

3b)

const char \*find\_the\_char(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 \*find\_the\_char(const char \*&str, char chr) {

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

if (\*str == chr) {

return str;

}

}

return nullptr;

}

4)

*int \*ptr = mini\_mart(array, &array[2]);*

- create a pointer to array[2] (4), since the function call returns array[2] if array[0] is less than array[2]

*ptr[1] = 9;*

- have the next value of the pointer set to 9, which would be array[3]

*ptr += 2;*

- increment the pointer two steps

*\*ptr = -1;*

- set the value of the pointer become -1, which would be array[4]

*\*(array + 1) = 79;*

- since an array is essentially a pointer pointing to the head, set the value one after the head to 79

*cout << “diff=” << &array[5] – ptr << endl;*

- output how far apart array[5] is from the pointer (5 – 4)

*swap(&array[0], &array[1]);*

*-* The function aims to swap the two entries in the array, but doesn’t dereference the pointers when doing so, and as such, doesn’t alter the value in the previous scope

*swap2(array, &array[2]);*

- The function swaps the entries at array[0] and array[2] with one another, and properly dereferences them both so that the change persists across the scope of the function.

5)

void delete\_g(char \*str) {

char \*temp = str;

while (\*str != 0) {

if (\*str != ‘g’ && \*str != ‘G’) {

\*temp = \*str;

\*temp++;

}

\*str;

}

\*temp = ‘\0’;

}