**Assume all necessary headers are in place**

1a .

// Bugs: \*ptr+1= 20; is an invalid expression, ptr is first derefenced to 5 then 1 is added, //resulting with 6 = 20 which is invalid

// Also, the ptr deincrementer was in the wrong place, it was being deincremented before //having \*ptr printed.

//correct version

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 //invalid expression

//correct way

\*(ptr + 1) = 20; //creates temporary pointer that points to first element

ptr += 2; //changes the pointer to point to element 2

ptr[0] = 30; // set arr[2] to 30 // NOTE \*(ptr+i) = ptr[i]

//while pointer(starts at 2) is pointing past array or equal to (0)

while (ptr >= arr)

{

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

ptr--; //this was previously above, needs to be underneath

}

cout << endl;

}

1b.

//BUG:

//the pointer was only passed by value

//therefore inside the function, only a copy was changed

//so when we returned ptr, we just passed the original which

//is an uninitialized pointer

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

{

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

{

if (arr[k] < arr[k - 1]) //item at position k is less than one before

{

//set spointer to point to what is the "disorder"

//point to whatever is at k

p = arr + k;

return;

}

}

p = nullptr; // otherwise make the pointer point to nothing

}

int main()

{

int nums[6] = { 10, 20, 20, 40, 30, 50 };

int\* ptr; //creates a pointer that points to an integer

findDisorder(nums, 6, ptr); //passes pointer into function

if (ptr == nullptr)

cout << "The array is ordered" << endl;

else

{

//this just prints the address

cout << "The disorder is at address " << ptr << endl;

//this prints position

cout << "It's at position " << ptr - nums << endl;

//doesn't point to the item but whatever is at the address of the above

cout << "The item's value is " << \*ptr << endl;

}

}

1c.

//When you initialize double\* p, you’re creating a pointer to a random double, and the odds are when you are accessing a random address that you’re not allowed to access it

//You can’t access the memory p is pointing to, it may work, but in general it will not do the right thing

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

{

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

}

int main()

{

double x;

//p can only store addresses "&" is address of operator

double\* p = &x; //a pointer that points to a double

hypotenuse(1.5, 2.0, p);

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

}

1d.

//PROBLEM: previously, the pointers themselves were being compared

//not the characters that are being pointed to

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

{

char\* ptr1 = str1;

char\* ptr2 = str2;

while ((\* ptr1) != '\0' && (\* ptr2) != '\0') // zero bytes at ends

{

//what it does: compares the addresses

if ((\* ptr1) != (\* ptr2)) // compare corresponding characters

return false;

ptr1++; // advance to the next character

ptr2++;

}

return \* ptr1 == \* ptr2; // both ended at same time?

}

int main()

{

char a[10] = "pointed";

char b[10] = "pointed";

if (match(a,b))

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

}

1e.

//problem:

//array is local only to the function it is created within

//array is not created in the main routine, but the function computeSquares, making it undefined behavior when we try to access it

//when we return there's no guarantee the value

//won't be overwritten by something else, the array junk in this case

int\* computeSquares(int& n)

{

int arr[10];

n = 10;

for (int k = 0; k < n; k++)

arr[k] = (k+1) \* (k+1);

return arr;

}

void f()

{

int junk[100];

for (int k = 0; k < 100; k++)

junk[k] = 123400000 + k;

}

int main()

{

int m; //declare an integer called m

//initialize pointer that points

//result of computeSquares

int\* ptr = computeSquares(m);

f();

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

cout << ptr[i] << ' ';

}

2.

string \* fp; // a. declare pointer variable fp that can point to a string

string fish[5]; //b. declare fish to be 5 element array of strings

fp = fish + 4; //c. make fp variable point to last element of fish

\* fp = "yellowtail"; //d. make string pointed to by fp equal to "yellowtail"

\*(fish + 3) = "salmon"; // e. without using fp pointer and without using square

fp -= 3; //f. move the fp pointer back by three strings

\*(fp+1) = "trout";//g. set third element in array to have value trout

fp[0] = "eel"; //h. without \*, or name fish; fp[0] dereferences current pos of ptr

bool d = (fp == fish); //i.

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

3a. Rewrite function so that it returns the same result, but does not increment the variable ptr. Your new program must not use any square brackets, but must use an integer variable to visit each double in the array. You may eliminate any unneeded variable.

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

{

const double\* ptr = scores; //points to first element in array

double tot = 0;

int idx = 0; //instead increment idx

while (idx < nScores)

{

tot += \*(ptr+idx);

idx++;

}

return tot/nScores;

}

3b. Rewrite the following function so that it does not use any square brackets (not even in the parameter declarations) but does use the integer variable k. Do not use any of the <cstring> functions such as strlen, strcpy, etc.

char\* findTheChar(char\* str, char chr) //return type is a ptr that points to a char

{

//creates a pointer that points to str+k, increment k

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

if (\* (str+k) == chr) //dereference ptr and compare the values

return str+k; //if equal, return pointer that points to match

return nullptr;

}

3c. Now rewrite the function shown in part b so that it uses neither square brackets nor any integer variables. Your new function must not use any local variables other than the parameters. Do not use any of the <cstring> functions such as strlen, strcpy, etc.

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

{

//set ptr = to address of first ele in str

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

//deref ptr, compare to char

if (\* ptr == chr){

return ptr; //if matching return

}

}

return nullptr; //no match and expended cString

}

4.

**Line by line trace (helps explain printed statements)**

Note: int array[6] = { 5, 3, 4, 17, 22, 19 };

(array, &array[2]) are passed into minimart which dereferences the two values and returns the pointer that points to the smaller value, in this case it is &array[2]

So the pointer ptr points to 4

“ptr[1] = 9;” sets the value one after 4 equal to 9, changing 17 to 9

New array: int array[6] = { 5, 3, 4, 9, 22, 19 };

“ptr += 2;” now shift pointer by 2 positions, pointer now points to 22

“\*ptr=-1;” set what pointer is pointing to (22) equal to -1, 22 becomes -1

New array: int array[6] = { 5, 3, 4, 9, -1, 19 };

“\*(array+1) = 79; change value at position 1 in array equal to 79

New array: int array[6] = { 5, 79, 4, 9, -1, 19 };

**What’s printed:**

The first line that will be printed will be: “diff=1” (the next printed statement will be on a newline)

This is because it subtracts the position of the last element in the array from what the pointer is pointing to which is the second to last element in the array currently (where -1 is)

**Note that:** swap1 does nothing because all it does is switch copies of the pointers, not the pointers themselves. Because the pointers aren’t passed by reference, their actual values aren’t changed.

Swap 2, switches what the pointers are pointing to using the dereference “\*” so it is fine. It switches what is at position 0 and what is at position 2 (4 and 5).

The new array should be : int array[6] = { 4, 79, 5, 9, -1, 19 };

**What’s printed:**

The forloop prints out the values in the array, each on its own line.

So it prints:

“9

4

7

79

-1

19”

5.

Your function must declare no more than one local variable in addition to the parameter; that additional variable must be of a pointer type. Your function must not use any square brackets. Do not use any of the <cstring> functions such as strlen, strcpy, etc.

void deleteG(char\* cString){

// if the next char is not null byte

while(\*(cString) != '\0'){

//if a g is found, shift everything left

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

char \* ptr = cString;

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

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

ptr++;

}

}

else{

cString++;

}

}

}