Zach North

603 885 768

Project 6

Problem 1

1a:

int main()

{

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

int\* ptr = arr;

\*ptr = 10;

\*(ptr + 1) = 20;

ptr += 2;

ptr[0] = 30;

for (int k = 0; k < 3; k++)

cout << \*(arr + k) << endl;

}

1b: The reason this function doesn’t work is because the pointer is not passed by reference to findMax. Thus, any changes made to the pointer in the function do not make any changes to the pointer in main.  
A correct implementation would be:  
  
void findMax(int arr[], int n, int\* &pToMax)

{

if (n <= 0)

return; // no items, no maximum!

int max = arr[0];

pToMax = &arr[0];

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

{

if (arr[i] > max)

{

max = arr[i];

pToMax = &(arr[i]);

}

}

}

1c. The problem with *ptr* is that it needs to be initialized to some memory address before the value in the memory address can be modified. If it points to nothing, the function will be unable to modify the value being pointed to. My fix is this:  
  
int main()

{

int\* ptr;

int n;

ptr = &n;

computeCube(5, ptr);

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

}

1d. There are numerous problems with this function. For starters, the while loop has incorrect conditions. Instead of breaking the loop when the pointers reach a value of 0, it should break when the value pointed to by the 2 pointers reach the null character – this would indicate the end of the cstring. The if statement is extraneous and unneeded -- in fact, the entire interior of the function can be vastly simplified. My version is as follows:

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

{

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

{

if (\*str1 == \*str2) //if the 2 chars are equal

{

str1++; // advance to the next character

str2++;

}

else

return false;

}

return true;

}

This steps through the letters/elements in str1 and str2, and if at any point they are different it returns false. If they are the same up until the null string at the end, the function will return true.

1e. The problem with this program is that the first function leaves a dangling pointer. It initializes an array, sets the pointer ptr to point to the first value of the array, and then ends. Then once f() is run, the operating system overwrites the array values with the garbage values created by f() and the pointer is still stuck pointing to them.  
  
To fix this problem , the programmer would have to initialize the array within the main function to make sure it is not overwritten in memory by later called functions.  
  
2.

//A

double \* scoop;

//B

double vanilla[5];

//C

scoop = &vanilla[4];

//D

\*scoop = 17;

//E

\*(vanilla + 3) = 42;

//F

scoop -= 3;

//G

scoop[1] = 33;

//H

scoop[0] = 25;

//I

bool b = (\*scoop == \*(scoop + 1));

//J

bool d = (scoop == vanilla);

3.  
a.  
double computeMean(const double\* scores, int numScores)

{

const double\* ptr = scores;

double tot = 0;

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

tot += \*(ptr + k);

return tot/numScores;

}

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 NULL;

}

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

{

if (\*str == chr)

return str;

else

str++;

if (\*str == '\0')

return NULL;

}

4. The function begins by declaring an integer array of 6 elements:  
{5 3 4 17 22 19}  
  
Then it calls the function *Maxwell*, which sets the pointer ptr to point to the whichever has a higher value between the first and third elements of array. The first element is 5, and the second element is 4, so ptr points to array[0].  
  
Then it sets the value pointed to by ptr to -1. The array is now:  
{-1 3 4 17 22 19}  
After this it increments ptr by 2. Ptr now points to the third element in array.  
  
Next, it sets the value of ptr[1] to 9. Ptr[1] refers to the value of the (3+1)th element of array, or the fourth element. The array is now:  
{-1 3 4 9 22 19}  
  
Now, it sets the value of the (array+1)th element to 79. Array by default points to the first element, so this sets the value of the second element to 79. The array is now:  
{-1 79 4 9 22 19}  
  
The function now prints the memory location of the last element of array, minus the memory location pointed to by pointer (the 3rd element.) This will print **3**, because these elements are 3 memory “chunks” apart from each other.  
  
Now the function calls swap1, and initializes it with the addresses of the values pointed to by array in the first and second element of array. Swap1 swaps these two pointers; however, since they are not passed by reference, the swapped pointers disappear at the end of the function. Swap1 does nothing.

Swap2 is called next with a pointer to the first and third elements of array. This function swaps the *values* *pointed to* by these pointers, rather than the pointers themselves. At the end of swap2 the array now contains:  
{4 79 -1 9 22 19}  
  
Now the program simply executes a for loop to output every element of array followed by an endl. This output will be:  
**4  
79  
-1  
9  
22  
19**.  
  
***So, when the program is run, the FINAL OUTPUT will be:  
3  
4  
79  
-1  
9  
22  
19***  
  
5.   
void removeS(char\* ptr)

{

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

{

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

{

char\* temp = ptr;

for(;;)

{

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

ptr++;

if (\*ptr == '\0')

{

ptr = temp;

break;

}

}

}

else

ptr++;

}

}