804622954

Shakthi Visagan

Stahl Toni

CS 31 Project 6

Problem 1: This problem involves errors related to the use of pointers.  
  
a. This program is supposed to write 1000 100 10 1 , but it doesn't. Find all of the bugs and show a fixed version of the program:   
 int main()  
 {  
 int arr[4] = { 0, 1, 2, 3 };  
 int\* ptr = arr;  
  
 \*ptr = arr[ 1 ]; // set arr[0] to 1  
 \*ptr + 1 = arr[ 0 ] \* 10; // set arr[1] to 10  
 ptr += 2;   
 ptr[0] = arr[ 1 ] \* 10; // set arr[2] to 100  
 ptr[1] = 1000; // set arr[3] to 1000  
  
 while (ptr >= arr)  
 {  
 ptr--;   
 cout << " " << \*ptr; // print values  
 }  
 cout << endl;   
 return( 0 );   
 }

Answer (Bugs will be written in red):

1. Declares and defines an array of size 4 with integers.
2. Arrays are just pointers to the first element of the array.
3. Create a pointer that points to the address location of the first element of arr.
4. Dereferencing the pointer, that is the first element, and making it equal to the element in arr[1].
   1. \*ptr = arr[0] = arr[1] = 1.
5. The line, \*ptr + 1 = arr[ 0 ] \* 10; , is incorrectly written.
   1. It should read as, \*(ptr + 1) = arr[ 0 ] \* 10; , the reason being that \*ptr gives us the value at the address location stored in ptr which can’t be modified in that manner. Using parentheses clarifies what we want.
6. ptr += 2; is the same as writing ptr = ptr + 2;. This is equivalent to writing that ptr = &arr[2].
7. It doesn’t make sense to use ptr as an array. We should dereference it since all ptr is is a pointer to an address location.
8. The line, ptr[0] = arr[ 1 ] \* 10; , is correctly written.
   1. It should read as, \*ptr = arr[ 1 ] \* 10; , the reason being that \*ptr will give us the value stored at the address location of arr[2] which is what the user wants. It is also worth noting that ptr is not an array.
9. It still doesn’t make sense to use ptr as an array. We should dereference it since all ptr is is a pointer to an address location.
10. The line, ptr[1] = 1000; , is correctly written.
    1. It should read as, \*(ptr +1) = 1000; , the reason being that \*(ptr + 1) will now will give us the value stored at the address location of arr[3] which is what the user wants.
11. As of right now, ptr points to the address location of arr[2]. We want, however, by the end of the program for it to point to arr[3].
    1. We can add a line, ptr += 1; to do this for us.
12. Finally, for the loop, we don’t want to exclude the first element of the array so we will subtract one element, that is, move down the allocated for the memory, AFTER we access it, not before.
    1. This can be fixed by moving, ptr--; , after the cout.
13. The new program is written down below on a new page.

int main()

{

int arr[4] = { 0, 1, 2, 3 };

int\* ptr = arr;

\*ptr = arr[1]; // set arr[0] to 1

**\*(ptr + 1) = arr[ 0 ] \* 10;** // set arr[1] to 10

ptr += 2; // moves up memory by 8

\*ptr = arr[1] \* 10; // set arr[2] to 100

ptr[1] = 1000; // set arr[3] to 1000

**ptr += 1;** // moves up memory by 4

while (ptr >= arr)

{

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

**ptr--;**  // moves down memory by 4

}

cout << endl;

return(0);

}

b. The findLastZero function is supposed to find the last element in an array whose value is zero, and sets the parameter p to point to that element, so the caller can know the location of that element holding the value zero. Explain why this function won't do that, and show how to fix it. Your fix must be to the function only; you must not change the main routine below in any way. As a result of your fixing the function, the main routine below needs to work correctly.  
  
 void findLastZero(int arr[], int n, int\* p)  
 {  
 p = nullptr; /// default value if there isn't a 0 in the array at all  
 for (int k = n - 1; k >= 0; k--)   
 {  
 if (arr[k] == 0) // found an element whose value is 0  
 {   
 p = arr + k; // change the value of p  
 break; // stop looping and return  
 }   
 }  
 }   
  
 int main()  
 {  
 int nums[6] = { 10, 20, 0, 40, 30, 50 };  
 int\* ptr;  
  
  
 findLastZero(nums, 6, ptr);  
 if (ptr == nullptr)  
 {  
 cout << "The array doesn't have any zeros inside it." << endl;   
 }   
 else  
 {  
 cout << "The last zero is at address " << ptr << endl;  
 cout << "It's at index " << ptr - nums << endl;   
 cout << "The item's value is " << \*ptr << " which is zero!" << endl;  
 }   
 return( 0 );   
 }

Answer (Bugs will be written in red):

1. Starting with main, define and declare an array of size 6 and type int with various integer values.
2. Declare a pointer called ptr.
3. Call function findLastZero with parameters, the array nums, an integer n=6, and a COPY of the pointer ptr. (This is most likely the error.)
4. Go to the function:
   1. Sets a default address location coming from the literal nullptr to p in the case there are no zeros in the array.
   2. Our for loop starts at 5, and decrements by 1, until it reaches 0 and ends after using 0. This makes sense since we want to find the last 0 in the array.
   3. If we find a 0, change the pointer address to the address of the array element that contains 0.
5. The problem exists within the function parameters. We have to change the actual pointer not just a copy of it. We need to pass it by REFERENCE. The new program is written below on a new page.

void findLastZero(int arr[], int n, **int\* &p**)  
 {  
 p = nullptr; /// default value if there isn't a 0 in the array at all  
 for (int k = n - 1; k >= 0; k--)   
 {  
 if (arr[k] == 0) // found an element whose value is 0  
 {   
 p = arr + k; // change the value of p  
 break; // stop looping and return  
 }   
 }  
 }   
  
 int main()  
 {  
 int nums[6] = { 10, 20, 0, 40, 30, 50 };  
 int\* ptr;  
  
  
 findLastZero(nums, 6, ptr);  
 if (ptr == nullptr)  
 {  
 cout << "The array doesn't have any zeros inside it." << endl;   
 }   
 else  
 {  
 cout << "The last zero is at address " << ptr << endl;  
 cout << "It's at index " << ptr - nums << endl;   
 cout << "The item's value is " << \*ptr << " which is zero!" << endl;  
 }   
 return( 0 );   
 }

c. The biggest function is correct, but the main function has a problem. Explain why it may not work, and show a way to fix it. Your fix must be to the main function only; you must not change the biggest function in any way.  
  
 #include <iostream>  
 using namespace std;  
  
  
 void biggest(int value1, int value2, int \* resultPtr)  
 {  
 if( value1 > value2 )   
 {  
 \*resultPtr = value1;  
 }   
 else  
 {  
 \*resultPtr = value2;   
 }  
 }  
  
 int main()  
 {  
 int\* p;  
 biggest(15, 20, p);   
 cout << "The biggest value is " << \*p << endl;  
 return( 0 );  
 }

Answer (Bugs will be written in red):

1. The error exists within declaring the pointer. It’s better to initialize the pointer with a pre-existing address. Fixed code is presented below on a new page.

#include <iostream>

using namespace std;

void biggest(int value1, int value2, int \* resultPtr)

{

if (value1 > value2)

{

\*resultPtr = value1;

}

else

{

\*resultPtr = value2;

}

}

int main()

{

**int tmparray[1] = {**

**0**

**};**

**int\* p = tmparray;**

biggest(15, 20, p);

cout << "The biggest value is " << \*p << endl;

return(0);

}

d. The match function is supposed to return true if and only if its two C-string arguments have exactly same text. Explain what the problems with the implementation of the function are, and show a way to fix them.  
  
 // return true if two C strings are equal   
 bool match(const char str1[], const char str2[])  
 {   
 bool result = true;   
 while (str1 != 0 && str2 != 0) // zero bytes at ends   
 {  
 if (str1 != str2) // compare corresponding characters  
 {  
 result = false;   
 break;   
 }   
 str1++; // advance to the next character  
 str2++;  
 }   
 if (result)   
 {   
 result = (str1 == str2); // both ended at same time?  
 }  
 return( result );  
 }  
  
  
 int main()  
 {  
 char a[10] = "pointy";  
 char b[10] = "pointless";   
  
  
 if (match(a,b))  
 {  
 cout << "They're the same!" << endl;  
 }  
 }

Answer (Bugs are written in bold):

1. The user is forgetting to use the dereferencing operator to access the character elements in the string array. There are several instances of this. The edited code is written below on a new page.

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

{

bool result = true;

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

{

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

{

result = false;

break;

}

str1++; // advance to the next character

str2++;

}

if (result)

{

result = **(\*str1 == \*str2)**; // both ended at same time?

}

return(result);

}

int main()

{

char a[10] = "pointy";

char b[10] = "pointless";

if (match(a, b))

{

cout << "They're the same!" << endl;

}

}

e. This program is supposed to write 1 1 2 3 5 8 13 21 but it probably does not. What is the program doing that is incorrect? (We're not asking you explain why the incorrect action leads to the particular outcome it does, and we're not asking you to propose a fix to the problem.)  
  
 #include <iostream>  
 using namespace std;  
  
 int fibonacci( int n )   
 {   
 int tmp;  
 int a = 1;  
 int b = 1;   
  
 for (int i = 0; i < n-2; i ++)   
 {  
 tmp = a+b;  
 a = b;  
 b = tmp;  
 }   
 return b;  
 }  
  
 int\* computeFibonacciSequence(int& n)  
 {  
 int arr[8];   
 n = 8;  
 for (int k = 0; k < n; k++)  
 {  
 arr[k] = fibonacci( k );   
 }   
 **return arr;**  
 }  
  
 int main()  
 {  
 int m;  
 **int\* ptr = computeFibonacciSequence(m);** for (int i = 0; i < m; i++)  
 {  
 cout << ptr[i] << ' ';  
 }  
 return( 0 );  
 }

Answer:

The array constructed in the function call ceases to exit after said function call. This is pointer that will point to something, not the array we want, in main.

2.

|  |  |
| --- | --- |
| 1. string \* fp; | **f. declares a pointer variable to point to a variable of type string** |
| 2. string fish[ 5 ]; | **g. declares a five element array of string** |
| 3. fp = &fish[ 4 ]; | **a. sets a pointer variable to the last element of an array of five strings** |
| 4. \*fp = "salmon"; | **b. sets the string pointed to by a pointer variable to the value "salmon"** |
| 5. fp -= 3; | **d. moves the pointer fp back three strings in the array it points to** |
| 6. \*(fp + 3) = "salmon"; | **c. sets the fourth element of an array pointed to by the variable fp to the value "salmon"** |
| 7. fp[ 0 ] = "salmon"; | **b. sets the string pointed to by a pointer variable to the value "salmon"** |
| 8. bool b = (fp == fish); | **e. initializes a boolean to true if the pointer variable fp points to the string at the start of the fish array, false otherwise** |
| 9. bool b = (\*fp == \*(fp + 1)); | **h. initializes a boolean to true if fp points to a string whose value matches the string immediately following the string pointed to by fp,  false otherwise** |

3. What does the following program print and why? Be sure to explain why each line of output prints the way it does to get full credit.   
  
 #include <iostream>  
 using namespace std;   
  
 int\* minimart(int\* a, int\* b)  
 {  
 if (\*a < \*b)  
 return a;  
 else   
 return b;  
 }  
  
 void swap1(int\* a, int \*b)  
 {  
 int\* temp = a;  
 a = b;   
 b = temp;  
 }  
  
 void swap2(int\* a, int \*b)  
 {  
 int temp = \*a;  
 \*a = \*b;   
 \*b = temp;  
 }  
  
 int main()  
 {  
 int array[6] = { 5, 3, 4, 17, 22, 19 };  
  
 int\* ptr = minimart(array, & array[2]);   
 ptr[1] = 9;  
 ptr += 2;  
 \*ptr = -1;   
 \*(array+1) = 79;  
  
 cout << "diff=" << &array[5] - ptr << endl;  
  
 swap1(&array[0], &array[1]);  
 swap2(array, &array[2]);  
  
 for (int i = 0; i < 6; i++)  
 cout << array[i] << endl;  
  
 return( 0 );  
 }

Reading through the code:

1. Starting with main, we declare an array of size 6, that is with 6 elements, of various integers.
2. Make a function call to minimart that returns a pointer and sets that return to a pointer named ptr.
   1. The function minimart takes in two parameters, two pointers.
   2. In this case the pointers it takes in are array, which is a pointer to the first element of the array array, which is another way of writing &array[0], the second pointer being &array[2].
   3. Minimart then takes these pointers and compares the values at their addresses. In our case the value 5 at &array[0] will be compared against the value 4 at &array[2].
   4. 5 is larger than 4, so minimart returns &array[2] which is then stored in ptr.
3. The next line then assigns ptr[1] = 9 which is the same as writing \*(ptr + 1) = 9 = array[3].
4. The next line then moves ptr up by two as it is the same as writing ptr = ptr + 2.
5. If we disregard the previous line (assume it was commented out), this line is the same as writing it as \*(ptr + 2) = -1 = array[4].
6. This line changes the value at array[1] to 79.
7. The calculation finds the difference between the addresses between ptr as it is and the address for array[5].
   1. Minimart made ptr equal to &array[2] and then the line ptr +=2 makes ptr equal to &array[4]. So the difference between &array[5] and &array[4] should be 1.
8. We then make a function call to swap1 which takes in two pointers as parameters, in our case &array[0] and &array[1].
   1. &array[0] still holds 5 as nothing was done to it and &array[1] holds 79 because of what happened at 6.
   2. We set some temp integer pointer variable to the address held at &array[0].
   3. The address held at &array[0] is now the address held at &array[1].
   4. The address held at &array[1] is now the temp integer pointer variable which is &array[0].
9. We then make a function call to swap2 which takes in two pointers as parameters, in our case array (which is equivalent to &array[0]) which holds 5 and &array[2]which still holds 4.
   1. Declare a temp variable that now holds the value at &array[0] = 5.
   2. The value now held at &array[0] is now the value held at &array[2] which is 4.
   3. The value now held at &array[2] is now the value from temp = 5.
10. Now to print out the array we have
    1. array[0] = 4
    2. array[1] = 79
    3. array[2] = 5
       1. These are because of 2. 7. 8. And 9.
    4. array[3] = 9
    5. array[4] = -1
       1. These are because of 3. 4. And 5.
    6. array[5] = 19
       1. This was left untouched.

4. Write a function named deleteCapitals that accepts one character pointer as a parameter and returns no value. The parameter must be a C-string. This function must remove all of the upper case letters from the string. The resulting string must be a valid C-string.  
  
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 and must not use the strlen or strcpy library functions.  
  
 int main()  
 {  
 char msg[100] = "Happy Days Are Here Again!";   
 deleteCapitals(msg);  
 cout << msg << endl; // prints: appy ays re ere gain!  
 }

Answer is on new page below:

#include <iostream>

void deleteCapitals(char\* message) {

char\* pointer = message;

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

if (\*pointer >= 'A' && \*pointer <= 'Z') {

//establish the complement of my desired condition

}

else { // if it is not a capital

\*message = \*pointer; //take it and place it in message again at a lower index

++message; // increase message by one

}

++pointer; // increase pointer by one

}

\*message = '\0'; // cut off message by capping it with the null

}

int main()

{

char msg[100] =

"Happy Days Are Here Again!"

;

deleteCapitals(msg);

std::cout << msg << std::endl; // prints: appy ays re ere gain!

}