Com Sci 31 – Winter 2019

Project 6

Name – Suyash Kumar

UID – 605108040

Discussion – 1C

# Problem 1a

Indicating Bugs

#include <iostream>  
    using namespace std;  
      
    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

// bug number 1

        while (ptr >= arr)  
        {  
            ptr--;  // bug number 2

            cout << " " << \*ptr;    // print values  
        }  
        cout << endl;   
        return( 0 );   
    }

1. The pointer currently points at the third item of the array, so the 4th element, 1000, won’t be printed. Thus, the pointer should be updated to one position forward so that printing could start from the 4th element of the array, that is 1000.
2. The decrement step should come after the print statement. The current position of the decrement step results in the printing to start from the 2nd array item, and leads to unprecedented behavior at the end of the loop when the pointer comes to point at a position one before the first element of the array, and we aren’t actually aware of what value is stored at that address.

Corrected version

#include <iostream>  
    using namespace std;  
      
    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

ptr++; // Position of the pointer was updated to the last element in the array.

        while (ptr >= arr)  
        {  
            cout << " " << \*ptr;    // print values

ptr--; // Correct positioning of the decrement step to ensure proper printing.  
        }  
        cout << endl;   
        return( 0 );   
    }

# Problem 1b

Identifying bugs in the function

#include <iostream>  
using namespace std;  
    void findLastZero(int arr[], int n, int\* p) // bug in reference type  
    {  
        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 );   
    }

The bug is that in the implementation of the findLastZero() function, the parameter p has been passed by value, so p isn’t actually modified. Now in the driver code, that is the main function, p has just been declared, not initialized to anything (not even nullptr), so it is dangling, and when the driver code runs, the control will transfer to the else statement and output will be unprecedented, undefined. The fix is therefore to pass the parameter p by reference.

Corrected version

#include <iostream>  
using namespace std;  
  
    void findLastZero(int arr[], int n, int\* & p)// pointer p passed by reference  
    {  
        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 );   
    }

# Problem 1c

Finding the bug in main

#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; // bug related to initialization  
        biggest(15, 20, p);   
        cout << "The biggest value is " << \*p << endl;  
        return( 0 );  
    }

The bug here is that the pointer p, as it is declared in the main function, is uninitialized. Essentially, the function is trying to dereference the value contained in the memory address pointed at by the pointer. And invoking the function on p tries to dereference the value contained at an unspecified, undefined address. Hence the code doesn’t build at all. Another point to be noted is that although the parameter pointer in the function isn’t passed by reference, all the function is trying to do is dereference the value held at the memory location pointed at by the pointer and isn’t trying to change the pointer itself. Thus, there is nothing wrong in the function.

Thus, the idea is to initialize the pointer p to a specific memory location, either in the stack or the heap.

Fixed version one – using stack memory

#include <iostream>  
using namespace std;

    void biggest(int value1, int value2, int \* resultPtr)  
    {  
        if( value1 > value2 )   
        {  
               \*resultPtr = value1;  
        }   
        else  
        {  
               \*resultPtr = value2;   
        }  
    }

    int main()  
    {

int i = 0;

int\* p = &i; // results in the p pointing at a specific memory address in the stack.  
        biggest(15, 20, p);   
        cout << "The biggest value is " << \*p << endl;  
        return( 0 );  
    }

Fixed version two – using heap memory

#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 = new int(5); // assigning a specific location in heap memory.  
        biggest(15, 20, p);   
        cout << "The biggest value is " << \*p << endl;

delete(p); // deleting to prevent any data leak.  
        return( 0 );  
    }

# Problem 1d

Identifying bugs

// 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, bug number 1, related to dereference  
        {  
            if (str1 != str2)  // compare corresponding characters, bug number 2, also related to dereference  
            {  
                result = false;   
                break;   
            }   
            str1++;            // advance to the next character  
            str2++;  
        }   
        if (result)    
        {   
            result = (str1 == str2);    // both ended at same time? Bug number 3, also related to dereference  
        }  
        return( result );  
    }

    int main()  
    {  
        char a[10] = "pointy";  
        char b[10] = "pointless"; 

        if (match(a,b))  
        {  
            cout << "They're the same!" << endl;  
        }  
    }

1. We should be applying the dereference operator to both str1 and str2 and checking whether the character pointed to by these pointers is a null character ‘\0’
2. We should be comparing the characters of the two C-Strings at the given positions, and the statement as it is actually comparing the memory addresses pointed at by the two pointers str1 and str2, so we should actually compare the values that their respective memory locations hold.
3. The same problem is encountered here, we should actually compare the value that the pointers’ respective memory locations hold and not the memory addresses themselves.

The fix is to simply use the dereference operator on the pointer instead of the pointer itself.

Fixed Version

// 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, checking null character at the end of the two C-Strings using dereference operator.  
        {  
            if (\*str1 != \*str2)  // compare corresponding characters, using dereference  
            {  
                result = false;   
                break;   
            }   
            str1++;            // advance to the next character  
            str2++;  
        }   
        if (result)    
        {   
            result = (\*str1 == \*str2);    // both ended at same time? Use dereference to check  
        }  
        return( result );  
    }

    int main()  
    {  
        char a[10] = "pointy";  
        char b[10] = "pointless"; 

        if (match(a,b))  
        {  
            cout << "They're the same!" << endl;  
        }  
    }

# Problem 1e

#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+1 );   
        }   
        return arr; // not possible to return a whole array  
    }

    int main()  
    {  
        int m;  
        int\* ptr = computeFibonacciSequence(m);   
        for (int i = 0; i < m; i++)  
        {  
            cout << ptr[i] << ' ';  
        }  
        return( 0 );  
    }

The problem here is that there is no way to return a complete array as a return value of a function, this is what the program is doing wrong. What is happening here is that the function is only returning the array pointer (at the first array element) and thus the value at the pointed memory location, and not the whole array. This is why looping from the pointer and ahead results in returning arbitrary values for all elements except the first, since the whole array wasn’t returned.

# Problem 2

1 – f

2 – g

3 – a

4 – b

5 – d

6 – c

7 – b

8 – e

9 – h

# Problem 3

The explanation for each function, and each line of the driver code (line numbers indicated in the comments on each line of driver code), is given below the code.

#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 }; // line1

        int\* ptr = minimart(array, & array[2]);  // line2  
        ptr[1] = 9; // line3  
        ptr += 2; // line4  
        \*ptr = -1;  // line5  
        \*(array+1) = 79; // line6

        cout << "diff=" << &array[5] - ptr << endl; // line7

        swap1(&array[0], &array[1]); // line8  
        swap2(array, &array[2]); // line9

        for (int i = 0; i < 6; i++) // line10  
            cout << array[i] << endl; // line 11

        return( 0 ); // line 12  
    }

Explanation of code

1. minimart() function – accepts two pointers, and returns the pointer which points to a memory address which carries a smaller value. In case the values contained in the linked memory addresses are the same, it will return the second pointer passed as the argument.
2. swap1() function – accepts two pointers, and swaps them using an additional pointer variable called temp.
3. swap2() function – accepts two pointers, and swaps the values contained in their linked memory addresses, and not the pointers themselves.
4. Driver code (line number indicated in comments) –
5. Declares an array of six elements, {5, 3, 4, 17, 22, 19}.
6. Pointer ptr will contain the memory address of the minimum of the 1st and 3rd elements of the array. As the 3rd element (4) is less than the 1st element (5), ptr will point to the 3rd array element, or array[2].
7. ptr[1], or array[3], is set to 9 while ptr still points to array[2]. The array now becomes {5, 3, 4, 9, 22, 19}.
8. ptr now points to array[4], which has the value 22.
9. The value contained at the memory address contained in ptr, that is array[4], is set to -1. Thus, array[4] is now -1. The array now becomes {5, 3, 4, 9, -1, 19}
10. Since the pointer array points at array[0], invoking \*(array+1) = 79 sets array[1] to 79. Thus, array[1] is now 79. Thus, array is now {5, 79, 4, 9, -1, 19}, and ptr is still pointing to array[4].
11. Returns the difference of memory address of array[5] and the memory address pointed to at by ptr, or array[4], and this difference is 1.
12. Invokes the swap1() function, which accepts two pointers &array[0] and &array[1]. However, since both pointers are passed to swap1() by value, the pointers passed as arguments aren’t actually swapped (their copies passed to the function are swapped). Hence, the array remains unchanged by this operation.
13. Invokes the swap2() function, which accepts two pointers array and &array[2]. Since swap2() function involves swapping the values stored at the memory locations pointed at by the pointers, and not the pointers themselves, this swap is effective, the values at array and &array[2] are interchanged. Thus, the array now becomes {4, 79, 5, 9, -1, 19}.

X/ XI/ XII. Runs a loop and prints each element of the modified array on a new line.

# Problem 4

Code for Problem 4

#include <iostream>

using namespace std;

void deleteDigits(char\*data);

int main()

{

char msg[100] = "Happy 2019!";

deleteDigits(msg);

cout << msg << endl; *// prints: Happy !*

return(0);

}

void deleteDigits(char\*data)

{

char \*temp = data; // using exactly one local variable of type pointer, make it start at the same place as the pointer data

while(\*data != '\0') // traversing through supplied C-String

{

if(\*data>='0' && \*data<='9') // if the encountered character in the given C-String is a digit

{

data++; // updates the pointer data while keeping temp at the same place

}

else // if the encountered character in the given C-String isn’t a digit

{

\*temp = \*data; // Will replace the character pointed to by temp with the character pointed to by data

temp++; // updates temp

data++; // updates data

}

}

// consequently, all the characters pointed to at by temp in due course of its updating, which will be less than or equal to the number of characters in the actual C-String itself, will be non-numeric.

\*temp = '\0'; // putting an end to the C-String to right where temp is (after collection of all non-digit characters of the passed C-String), to ensure it stays valid.

}