MTH 4300: Algorithms, Computers and Programming II

Spring 2026

Section: SMWA

Midterm 1 Practice Exam 1 — Solution Key

Instructions:

- **Tracing:** Write the exact output, showing intermediate variable values step by step.
- **Debugging:** For each bug — state the line, explain the problem, and write the corrected code.
- **Programming:** Write pseudocode in Part A first, then a complete C++ implementation in Part B.

Note: Submit handwritten responses as a PDF.

Question 1: Tracing — Pointers, References & Functions

What is the output of the following program? Trace through step by step, paying close attention to how each argument is passed.

```cpp
#include <iostream>

void modify(int a, int& b, int* c) {
    a += 10;
    b += 20;
    *c += 30;
    std::cout << "Inside modify: a=" << a << ", b=" << b << ", *c=" << *c << std::endl;
}

int main() {
    int x = 5;
    int y = 10;
    int z = 15;

    std::cout << "Before: x=" << x << ", y=" << y << ", z=" << z << std::endl;

    modify(x, y, &z);

    std::cout << "After: x=" << x << ", y=" << y << ", z=" << z << std::endl;

    int* ptr = &x;
    int& ref = y;

    *ptr = ref + z;
    ref = *ptr - 10;

    std::cout << "Final: x=" << x << ", y=" << y << ", z=" << z << std::endl;

    return 0;
}
```

Solution

Step-by-step trace:

Initial state: `x=5, y=10, z=15`

Line 47 prints: `Before: x=5, y=10, z=15`

`modify(x, y, &z)` is called:

- `a` receives a **copy** of `x` → `a=5`. Changes to `a` do not affect `x`.
- `b` is a **reference** to `y`. Changes to `b` change `y`.
- `c` is a **pointer** to `z`. `*c` dereferences to `z`.
- `a += 10` → `a=15` (x still 5)
- `b += 20` → `b=30`, so `y=30`
- `*c += 30` → `z=45`

Line 39 prints: `Inside modify: a=15, b=30, *c=45`

After returning: `x=5, y=30, z=45`

Line 51 prints: `After: x=5, y=30, z=45`

- `ptr = &x` → `ptr` points to `x`
- `ref = y` → `ref` is an alias for `y`
- `*ptr = ref + z` → `x = 30 + 45 = 75`
- `ref = *ptr - 10` → `y = 75 - 10 = 65`

Line 59 prints: `Final: x=75, y=65, z=45`

Complete output:

```
Before: x=5, y=10, z=15
Inside modify: a=15, b=30, *c=45
After: x=5, y=30, z=45
Final: x=75, y=65, z=45
```

Question 2: Tracing — Recursion + Vectors

What is the output of the following program? For each recursive call, show the values of `left` and `right` and the state of the vector.

```
#include <iostream>
#include <vector>

int mystery(std::vector<int>& v, int left, int right) {
    if (left >= right) {
        return v[left];
    }
    int mid = (left + right) / 2;
    int temp = v[left];
    v[left] = v[right];
    v[right] = temp;
    return mystery(v, left + 1, right - 1) + v[left] + v[right];
}

int main() {
    std::vector<int> nums = {1, 2, 3, 4, 5};

    int result = mystery(nums, 0, 4);

    std::cout << "Result: " << result << std::endl;
    std::cout << "Vector: ";
    for (int i = 0; i < nums.size(); i++) {
        std::cout << nums[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Solution

`mystery` swaps `v[left]` and `v[right]`, then recurses inward, returning the recursive result plus the (now-swapped) values at the outer positions.

Call 1: `mystery(v, 0, 4)`, `v = {1, 2, 3, 4, 5}`
- `left=0 < right=4`, so swap `v[0]` and `v[4]` → `v = {5, 2, 3, 4, 1}`
- return `mystery(v, 1, 3) + v[0] + v[4]` = `mystery(v, 1, 3) + 5 + 1`

Call 2: `mystery(v, 1, 3)`, `v = {5, 2, 3, 4, 1}`
- `left=1 < right=3`, so swap `v[1]` and `v[3]` → `v = {5, 4, 3, 2, 1}`
- return `mystery(v, 2, 2) + v[1] + v[3]` = `mystery(v, 2, 2) + 4 + 2`

Call 3: `mystery(v, 2, 2)`, `v = {5, 4, 3, 2, 1}`
- `left=2 >= right=2` → base case, return `v[2] = 3`

Unwinding:
- Call 2 returns: `3 + 4 + 2 = 9`
- Call 1 returns: `9 + 5 + 1 = 15`

Final vector: `{5, 4, 3, 2, 1}` (each pair was swapped)

Complete output:

```
Result: 15
Vector: 5 4 3 2 1
```

Question 3: Debugging — Linked List

The program below intends to delete a node by value from a linked list, print the result, then free all memory. It contains **4 bugs**. For each one, state the line, explain the problem, and write the fix.

```cpp
#include <iostream>

struct Node {
    int data;
    Node* next;
};

void deleteValue(Node* head, int value) {
    if (head == nullptr) return;

    if (head->data == value) {
        head = head->next;
        return;
    }

    Node* current = head;
    while (current != nullptr) {
        if (current->next->data == value) {
            Node* temp = current->next;
            current->next = temp->next;
            delete temp;
            return;
        }
        current = current->next;
    }
}

void printList(Node* head) {
    while (head != nullptr) {
        std::cout << head->data << " -> ";
        head = head->next;
    }
    std::cout << "nullptr" << std::endl;
}

void deleteList(Node* head) {
    while (head != nullptr) {
        delete head;
        head = head->next;
    }
}

int main() {
    Node* list = nullptr;
    for (int i = 5; i >= 1; i--) {
        Node* newNode = new Node;
        newNode->data = i;
        newNode->next = list;
        list = newNode;
    }

    std::cout << "Before: ";
    printList(list);

    deleteValue(list, 3);
    deleteValue(list, 1);

    std::cout << "After: ";
    printList(list);

    deleteList(list);
    return 0;
}
```

Solution

Bug 1 — Line 116: `head` passed by value, caller's pointer is never updated

`deleteValue` takes `Node* head` by value. When the head node is the one to delete, `head = head->next` only changes the local copy. `list` in `main` is unaffected.

Fix: pass by reference to pointer:

```
void deleteValue(Node*& head, int value)
```

Bug 2 — Line 120: Head node is not freed (memory leak)

Even after fixing Bug 1, the original head node is never deleted before `head` is advanced. This leaks memory.

Fix: save and delete the old head:

```
Node* old = head;
head = head->next;
delete old;
return;
```

Bug 3 — Line 125: Loop condition allows null dereference

`while (current != nullptr)` permits `current` to be the last node, then `current->next->data` on line 126 dereferences a null pointer and crashes.

Fix: stop when `current->next` is null:

```
while (current->next != nullptr)
```

Bug 4 — Lines 145–147: Use-after-free in `deleteList`

`delete head` frees the node, then `head = head->next` reads from freed memory.

Fix: save `next` before deleting:

```
void deleteList(Node* head) {
    while (head != nullptr) {
        Node* next = head->next;
        delete head;
        head = next;
    }
}
```

Question 4: Debugging — Vectors & Functions

The program below intends to collect even numbers from a vector and print them comma-separated. It contains **4 bugs**. For each one, state the line, explain the problem, and write the fix.

```cpp
#include <iostream>
#include <vector>

std::vector<int> getEvens(std::vector<int> v) {
    std::vector<int> result;
    for (int i = 0; i <= v.size(); i++) {
        if (v[i] % 2 != 0) {
            result.push_back(v[i]);
        }
    }
    return result;
}

void printVector(const std::vector<int>& v) {
    for (int i = 0; i < v.size(); i++) {
        std::cout << v[i];
        if (i < v.size()) std::cout << ", ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> nums = {1, 2, 3, 4, 5, 6};
    std::vector<int> evens = getEvens(nums);
    std::cout << "Even numbers: ";
    printVector(evens);
    return 0;
}
```

Solution

Bug 1 — Line 184: Vector passed by value instead of const reference

`getEvens` takes `std::vector<int> v` by value, making an unnecessary copy. For a function that only reads its input, use a const reference.

Fix:

```cpp
std::vector<int> getEvens(const std::vector<int>& v)
```

Bug 2 — Line 186: Off-by-one error in loop condition

`i <= v.size()` allows `i` to reach `v.size()`, causing an out-of-bounds access on line 187 (`v[v.size()]` is undefined behavior).

Fix:

```cpp
for (int i = 0; i < v.size(); i++)
```

Bug 3 — Line 187: Wrong condition selects odd numbers instead of even

`v[i] % 2 != 0` is true when the number is **odd**. To collect evens, the condition should be `== 0`.

Fix:

```cpp
if (v[i] % 2 == 0)
```

Bug 4 — Line 197: Trailing comma printed after the last element

`if (i < v.size())` is always true for every valid index, so a comma is printed after every element including the last. Fix: only print the comma when it is not the last element.

Fix:

```cpp
if (i < v.size() - 1) std::cout << ", ";
```

Question 5: Programming — Counting Characters Recursively

Write a recursive function with the following signature:

```
int countChar(const std::string& s, char c, int index)
```

Returns how many times `c` appears in `s` from position `index` to the end. No loops allowed.

Examples: `countChar("hello", 'l', 0)` → 2, `countChar("banana", 'a', 0)` → 3, `countChar("", 'a', 0)` → 0

Part A: Pseudocode

Identify your base case and your recursive case. Explain how you combine the result of the recursive call with the current character.

Solution

```
function countChar(s, c, index):
    // Base case: index is at or past the end of the string
    if index >= length of s:
        return 0

    // Check whether the current character matches
    match = 1 if s[index] == c, else 0

    // Recursive case: add match to the count from the rest of the string
    return match + countChar(s, c, index + 1)
```

Base case: When `index >= s.size()`, we have scanned the entire string — return 0.

Recursive case: Check whether `s[index]` equals `c` (contributing 0 or 1), then add the count from the remaining suffix starting at `index + 1`.

Part B: Implementation

Write the complete function and a `main()` with at least 4 test cases — include an empty string and a character that does not appear.

Solution

```cpp
#include <iostream>
#include <string>

int countChar(const std::string& s, char c, int index) {
    if (index >= (int)s.size()) return 0;
    int match = (s[index] == c) ? 1 : 0;
    return match + countChar(s, c, index + 1);
}

int main() {
    std::cout << countChar("hello", 'l', 0) << std::endl;    // 2
    std::cout << countChar("banana", 'a', 0) << std::endl;   // 3
    std::cout << countChar("", 'a', 0) << std::endl;          // 0
    std::cout << countChar("hello", 'z', 0) << std::endl;    // 0
    return 0;
}
```

Question 6: Programming — Removing Duplicates

Write a function with the following signature:

```
std::vector<int> removeDuplicates(const std::vector<int>& v)
```

Returns a new vector with only the unique elements from `v`, in the order they first appear. No sorting or standard library algorithms (no `std::set`, `std::unique`, etc.).

Examples: `{1, 2, 3, 2, 1, 4}` → `{1, 2, 3, 4}`, `{5, 5, 5}` → `{5}`, `{}` → `{}`

Part A: Pseudocode

Explain how you decide whether each element has already been seen earlier in the vector.

Solution

```
function removeDuplicates(v):
    result = empty vector

    for each element x in v:
        found = false
        for each element y already in result:
            if y == x:
                found = true
                break

        if not found:
            append x to result

    return result
```

For each element in `v`, we scan `result` from the beginning. If the element already appears in `result`, we skip it. Otherwise we add it. Since `result` only grows with first occurrences, order is preserved.

Part B: Implementation

Write the complete function and a `main()` with at least 3 test cases including an empty vector.

Solution

```cpp
#include <iostream>
#include <vector>

std::vector<int> removeDuplicates(const std::vector<int>& v) {
    std::vector<int> result;
    for (int i = 0; i < (int)v.size(); i++) {
        bool found = false;
        for (int j = 0; j < (int)result.size(); j++) {
            if (result[j] == v[i]) {
                found = true;
                break;
            }
        }
        if (!found) result.push_back(v[i]);
    }
    return result;
}

void printVector(const std::vector<int>& v) {
    std::cout << "{ ";
    for (int i = 0; i < (int)v.size(); i++) {
        std::cout << v[i];
        if (i < (int)v.size() - 1) std::cout << ", ";
    }
    std::cout << " }" << std::endl;
}

int main() {
    std::vector<int> a = {1, 2, 3, 2, 1, 4};
    printVector(removeDuplicates(a));   // { 1, 2, 3, 4 }

    std::vector<int> b = {5, 5, 5};
    printVector(removeDuplicates(b));   // { 5 }

    std::vector<int> c = {};
    printVector(removeDuplicates(c));   // {  }
```

```
    return 0;
}
```

Question 7: Programming — Sorted Linked List

Given the following struct:

```
struct Node {
    int data;
    Node* next;
};
```

Write a function with the following signature:

```
void insertSorted(Node*& head, int value)
```

Inserts a new node into a sorted singly linked list, maintaining ascending order. For example, inserting 30, 10, 50, 20, 40 into an empty list should produce `10 -> 20 -> 30 -> 40 -> 50 -> nullptr`.

Part A: Pseudocode

List each distinct case your function must handle (empty list, insert at beginning, insert in middle or end) and describe the pointer updates required for each.

Solution

```
function insertSorted(head, value):
    create newNode with newNode.data = value, newNode.next = nullptr

    // Case 1: empty list
    if head == nullptr:
        head = newNode
        return

    // Case 2: new value belongs before the current head
    if value <= head.data:
        newNode.next = head
        head = newNode
        return

    // Case 3: find the insertion point in the middle or at the end
    current = head
    while current.next != nullptr AND current.next.data < value:
        advance current

    // Insert between current and current.next
    newNode.next = current.next
    current.next = newNode
```

- **Empty list:** set `head` to the new node.
- **Insert at beginning:** new node's `next` points to old head; `head` updated to new node. Must pass `head` by reference so caller sees the change.
- **Insert in middle or end:** traverse until `current->next` is null or its value is >= the new value, then splice in the new node.

Part B: Implementation

Write `insertSorted`, a `printList` function, a `deleteList` function, and a `main()` that inserts at least 5 values in non-sorted order, prints the result, and frees all memory.

Solution

```
#include <iostream>

struct Node {
    int data;
    Node* next;
};

void insertSorted(Node*& head, int value) {
    Node* newNode = new Node;
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr || value <= head->data) {
        newNode->next = head;
        head = newNode;
        return;
    }

    Node* current = head;
    while (current->next != nullptr && current->next->data < value) {
```

```
        current = current->next;
    }
    newNode->next = current->next;
    current->next = newNode;
}

void printList(Node* head) {
    while (head != nullptr) {
        std::cout << head->data << " -> ";
        head = head->next;
    }
    std::cout << "nullptr" << std::endl;
}

void deleteList(Node* head) {
    while (head != nullptr) {
        Node* next = head->next;
        delete head;
        head = next;
    }
}

int main() {
    Node* list = nullptr;

    insertSorted(list, 30);
    insertSorted(list, 10);
    insertSorted(list, 50);
    insertSorted(list, 20);
    insertSorted(list, 40);

    printList(list);   // 10 -> 20 -> 30 -> 40 -> 50 -> nullptr

    deleteList(list);
    return 0;
}
```