**MTH 4300: Programming and Computer Science II**

**Spring 2026**

**Section: SMWA**

# Midterm 2 Practice Exam 2

**Instructions:**

- Questions 1 and 2 are **tracing** questions. Write the exact output or stack contents requested.
- Questions 3 and 4 are **debugging** questions. Identify each bug, explain why it is a bug, and write the corrected line(s) of code.
- Questions 5 and 6 are **programming** questions. Part A asks for pseudocode; Part B asks for a complete C++ implementation.
- Use `std::` prefix where appropriate. Do not use `using namespace std`.
- You may assume all necessary headers are included unless stated otherwise.

**Note:** Submit handwritten responses as a PDF.

## Question 1: Tracing Operator Overloading

Consider the following program. The `Counter` class overloads several operators and prints messages from them.

```
#include <iostream>

class Counter {
 private:
  int value;

 public:
  Counter(int v) : value(v) {
    std::cout << "make " << value << std::endl;
  }

  Counter operator+(const Counter& other) const {
    std::cout << "add " << value << "+" << other.value << std::endl;
    return Counter(value + other.value);
  }

  Counter& operator++() {
    ++value;
    std::cout << "pre " << value << std::endl;
    return *this;
  }

  bool operator==(const Counter& other) const {
    std::cout << "eq " << value << "?" << other.value << std::endl;
    return value == other.value;
  }

  friend std::ostream& operator<<(std::ostream& os, const Counter& c) {
    os << "C(" << c.value << ")";
    return os;
  }
};

int main() {
  Counter a(2);
  Counter b(3);
  Counter c = a + b;
  ++c;
  std::cout << c << std::endl;
  if (c == a) {
    std::cout << "same" << std::endl;
  } else {
    std::cout << "diff" << std::endl;
  }
  return 0;
}
```

Write the **exact** output produced by this program, in order.

### Question 2: Tracing Inheritance Constructor and Destructor Order

Consider the following program. Each class prints a message from its constructor and destructor.

```
#include <iostream>

class Animal {
 public:
  Animal() { std::cout << "Animal ctor" << std::endl; }
  ~Animal() { std::cout << "Animal dtor" << std::endl; }
};

class Mammal : public Animal {
 public:
  Mammal() { std::cout << "Mammal ctor" << std::endl; }
  ~Mammal() { std::cout << "Mammal dtor" << std::endl; }
};

class Dog : public Mammal {
 public:
  Dog() { std::cout << "Dog ctor" << std::endl; }
  ~Dog() { std::cout << "Dog dtor" << std::endl; }
};

class Cat : public Mammal {
 public:
  Cat() { std::cout << "Cat ctor" << std::endl; }
  ~Cat() { std::cout << "Cat dtor" << std::endl; }
};

int main() {
  std::cout << "--- start ---" << std::endl;
  Dog d;
  std::cout << "--- middle ---" << std::endl;
  {
    Cat c;
  }
  std::cout << "--- end ---" << std::endl;
  return 0;
}
```

Write the **exact** output produced by this program, in order. Be careful to remember the rule about the order in which base-class and derived-class constructors and destructors are called.

## Question 3: Debugging a `Buffer` Class

The following class wraps a dynamically allocated array of `int`. It contains **three** bugs.

```
#include <cstddef>

class Buffer {
 private:
  int* data;
  std::size_t size;

 public:
  Buffer(std::size_t n) {
    size = n;
    data = new int(n);
    for (std::size_t i = 0; i < size; ++i) {
      data[i] = 0;
    }
  }

  Buffer(const Buffer& other) {
    size = other.size;
    data = other.data;
  }

  Buffer& operator=(const Buffer& other) {
    delete[] data;
    size = other.size;
    data = new int[size];
    for (std::size_t i = 0; i < size; ++i) {
      data[i] = other.data[i];
    }
    return *this;
  }

  ~Buffer() {
    delete[] data;
  }
};
```

For **each** of the three bugs:

1. Name the member function that contains the bug.
2. Explain in one or two sentences why it is a bug (what goes wrong at runtime).
3. Write the corrected line or lines of code.

## Question 4: Debugging a First-Unique-Character Function

The following C++ function is **intended** to return the **index** of the first character in `s` that appears **exactly once**. If no such character exists, it should return `-1`. The intended approach is:

1. First pass: count how many times each character appears, using a `std::unordered_map<char, int>`.
2. Second pass: walk through `s` from left to right, returning the index of the first character whose count is `1`.

For example, `first_unique("leetcode")` should return `0` (the first `'l'` is unique), and `first_unique("aabb")` should return `-1`.

The function below contains **three** bugs.

```cpp
#include <string>
#include <unordered_map>

int first_unique(const std::string& s) {
  std::unordered_map<char, int> counts;
  for (std::size_t i = 0; i < s.length(); ++i) {
    counts[s[i]] = 1;
  }

  for (std::size_t i = 0; i <= s.length(); ++i) {
    if (counts[s[i]] == 1) {
      return i;
    }
  }
}
```

For **each** of the three bugs:

1. Identify the line (or lines) where the bug occurs.
2. Explain in one or two sentences why it is a bug (what goes wrong at runtime, or why the result is incorrect).
3. Write the corrected line or lines of code.

**Hint:** think carefully about (a) how the counting step accumulates counts, (b) the loop bound in the second pass, and (c) what value the function returns when no unique character exists.

## Question 5: Programming — The `Stack` Class

Implement a class `Stack` that represents a stack of `int` values backed by a **dynamically allocated** array. When the array becomes full, it must **double** its capacity.

Your class must have the following private members:

- `int* data` — pointer to the dynamically allocated array
- `int size` — the number of elements currently in the stack
- `int capacity` — the size of the underlying array

Your class must implement **all five** special member functions **and** the constructor listed below:

1. Constructor `Stack()` that allocates an initial buffer of capacity `4`, with `size = 0`.
2. Destructor that releases the dynamically allocated memory.
3. Copy constructor (deep copy).
4. Copy assignment operator (deep copy, handling self-assignment).
5. Move constructor (leaves the moved-from object in a valid empty state with `data == nullptr`, `size == 0`, `capacity == 0`).
6. Move assignment operator (same guarantees as move constructor, handling self-assignment).

Additionally, implement the following public methods:

- `void push(int value)` — appends a value to the top of the stack, doubling capacity if `size == capacity`.
- `int pop()` — removes and returns the top value. You may assume the stack is non-empty when `pop` is called.
- `int top() const` — returns (without removing) the top value. You may assume the stack is non-empty.
- `int get_size() const` — returns the current number of elements.
- `bool empty() const` — returns `true` if the stack has no elements.

### Part A: Pseudocode

Write pseudocode describing the constructor, destructor, copy constructor, copy assignment, move constructor, move assignment, `push` (including the doubling step), and the remaining accessors.

### Part B: C++ Implementation

Write the complete class definition including all six special member functions and the five public methods listed above.

## Question 6: Programming — Intersection of Two Arrays

Write a C++ function

```
std::vector<int> intersection(const std::vector<int>& a, const std::vector<int>& b);
```

that returns a vector containing the **unique** values that appear in **both** `a` and `b`. Each value should appear **at most once** in the returned vector, regardless of how many times it appears in either input. The order of elements in the returned vector does not matter.

**Examples:**

- `intersection({1, 2, 2, 1}, {2, 2})` returns a vector equivalent to `{2}`.
- `intersection({4, 9, 5}, {9, 4, 9, 8, 4})` returns a vector equivalent to `{4, 9}`.
- `intersection({1, 2, 3}, {4, 5, 6})` returns `{}`.
- `intersection({}, {1, 2, 3})` returns `{}`.
- `intersection({-1, 0, 1}, {1, 0, -1, 2})` returns a vector equivalent to `{-1, 0, 1}`.

**Requirement:** your solution must run in $O(m + n)$ expected time, where `m` and `n` are the sizes of `a` and `b`. You must use `std::unordered_set<int>` from the `<unordered_set>` header. The idea is:

- Insert every value of `a` into a set `set_a`.
- Iterate through `b`. For each value `x` in `b`, if `x` is in `set_a`, add `x` to the result **and** remove `x` from `set_a` so it cannot be added a second time.

A nested-loop $O(m \cdot n)$ solution is **not acceptable**.

Then, write a `main` function that tests your implementation on the five examples above. Print each output vector in the format `[a, b, c, ...]` (or `[]` if empty).

### Part A: Pseudocode

Describe the algorithm above, and explain why it runs in $O(m + n)$ expected time. Be explicit about **why** removing `x` from `set_a` after adding it to the result is necessary to avoid duplicates in the output.

### Part B: C++ Implementation

Write the complete program, including all necessary `#include` directives.