MTH 4300: Programming and Computer Science II

Fall 2025

Section: STRA

Midterm 2 (November 18th, 2025)

Grading Section (For Instructor Use)

Question	Points	Score
Question 1	30	/ 30
Question 2	10	/ 10
Question 3	10	/ 10
Total	**50**	/ 50

Comments:

Question 1

You are implementing a `Polynomial` class to perform mathematical operations on polynomials like $P(x) = 3x^2 + 2x + 1$.

Part 1 (20 points)

Design and implement a `Polynomial` class with the following specifications:

Private members:

- `double* coefficients` - dynamically allocated array storing coefficients
- `int degree` - highest power of x (degree of polynomial)

Public interface:

- `Polynomial(int deg)` - constructor creating polynomial of given degree, all coefficients initialized to 0.0
- Complete **Rule of Five**: copy constructor, copy assignment operator, move constructor, move assignment operator, destructor
- `void set_coefficient(int power, double coeff)` - sets coefficient for x^{power}
- `double get_coefficient(int power) const` - returns coefficient for x^{power}
- `int get_degree() const` - returns degree of polynomial

Storage format: Store coefficients where index represents power:

- `coefficients[0]` = constant term
- `coefficients[1]` = coefficient of x^1
- `coefficients[2]` = coefficient of x^2

Requirements:

- Copy operations create independent deep copies
- Move operations efficiently transfer ownership

Write the complete class definition and all member function implementations.

```
class Polynomial {
private:
  double *coefficients;
  int degree;

public:
  Polynomial(int degree) : degree(degree) {
    coefficients = new double[degree + 1];
  }

  ~Polynomial() {
    delete[] coefficients;
    coefficients = nullptr;
  }

  Polynomial(const Polynomial &other) {
    degree = other.degree;
    coefficients = new double[degree + 1];
    for (int i = 0; i < degree + 1; i++) {
      coefficients[i] = other.coefficients[i];
    }
  }

  Polynomial &operator=(const Polynomial &other) {
    if (this == &other) {
      return *this;
    }

    delete[] coefficients;

    degree = other.degree;
    coefficients = new double[degree + 1];
    for (int i = 0; i < degree + 1; i++) {
      coefficients[i] = other.coefficients[i];
    }

    return *this;
  }

  Polynomial(Polynomial &&other) noexcept
      : degree(other.degree), coefficients(other.coefficients) {
    other.coefficients = nullptr;
    other.degree = 0;
  }

  Polynomial &operator=(Polynomial &&other) noexcept {
```

```
    if (this == &other) {
      return *this;
    }

    delete[] coefficients;
    degree = other.degree;
    coefficients = other.coefficients;

    other.coefficients = nullptr;
    other.degree = 0;

    return *this;
  }

  int get_degree() const { return degree; }

  double get_coefficient(int power) const { return coefficients[power]; }

  void set_coefficient(int power, double coeff) {
    if (power < 0 || power > degree) {
      throw std::runtime_error("invalid power provided");
    }

    coefficients[power] = coeff;
  }

  double evaluate(double x) const;
  Polynomial derivate() const;
};
```

Part 2 (10 points)

Implement the following methods for your class as well.

- `double evaluate(double x) const` - evaluates polynomial at given x value
- `Polynomial derivative() const` - returns derivative as new Polynomial object

Requirements:

- `evaluate(x)` computes: $c_0 + c_1x + c_2x^2 + ... + c_nx^n$
- `derivative()` applies the following rule: $\frac{d}{dx}(c_nx^n) = n \cdot c_nx^{n-1}$

Solution

```
double Polynomial::evaluate(double x) const {
  double result = 0;

  double curr = 1;
  for (int i = 0; i < degree + 1; i++) {
    result += coefficients[i] * curr;
    curr *= x;
  }

  return result;
}

Polynomial Polynomial::derivate() const {
  if (degree == 0) {
    Polynomial dx(degree);
    return dx;
  }

  Polynomial dx(degree - 1);
  for (int i = 1; i < degree + 1; i++) {
    dx.coefficients[i - 1] = coefficients[i] * i;
  }

  return dx;
}
```

Question 2

Given the following `Node` structure for a singly linked list:

```
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};
```

Write a function `insert_sorted` that inserts a new value into a sorted linked list while maintaining the sorted order.

Function signature:

```
Node* insert_sorted(Node* head, int value);
```

Examples:

- Input: List = [1, 3, 5, 7], value = 4
- Output: [1, 3, 4, 5, 7]

- Input: List = [2, 4, 6], value = 1
- Output: [1, 2, 4, 6]

- Input: List = [1, 2, 3], value = 5
- Output: [1, 2, 3, 5]

Solution

```
Node *insert_sorted(Node *head, int value) {
  Node *new_node = new Node(value);

  if (head == nullptr) {
    return new_node;
  }

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

  Node *current = head;
  Node *prev = nullptr;

  while (current != nullptr && current->data < value) {
    prev = current;
    current = current->next;
  }

  if (prev != nullptr) {
    prev->next = new_node;
    new_node->next = current;
  }

  return head;
}
```

Question 3

You are given a string containing lowercase letters and asterisks (*). Each asterisk represents a "backspace" operation that removes the closest non-asterisk character to its left.

Write a function `remove_stars` that processes the string and returns the final result after all asterisk operations.

Function signature:

```
std::string remove_stars(const std::string& s);
```

```
- Input: "abc*de*f"
- Output: "abdf"

- Input: "a*b*c*"
- Output: ""

- Input: "hello*world"
- Output: "hellworld"
```

Solution

```
#include <stack>
#include <string>

std::string remove_stars(const std::string &s) {
  std::stack<char> stack;
  std::string res;

  for (const auto &ch : s) {
    if (ch == '*') { stack.pop(); }
    else { stack.push(ch); }
  }

  while (!stack.empty()) {
    res = stack.top() + res;
    stack.pop();
  }

  return res;
}
```