

[1]// 1

## Lecture Review

In this lab, you will extend the `string` class from Lab 1 and work with the `Fraction` class taught in lectures. The focus will be on:

1. **Copy Constructor:** Creates a new object as a copy of an existing object. For classes with dynamic memory, it must perform a **deep copy** to avoid multiple objects pointing to the same memory.
2. **Destructor:** Automatically called when an object goes out of scope. It's responsible for cleaning up dynamically allocated memory to prevent memory leaks.
3. **Copy Assignment Operator (`operator=`):** Handles assignment between existing objects. It must:
  - Check for self-assignment
  - Free existing memory (if reallocating)
  - Perform deep copy of the source object
  - Return `*this`
4. **Operator Overloading:** Allows you to define custom behavior for operators (`+`, `[ ]`, `==`, etc.) when used with your class objects. This makes your classes more intuitive to use.
5. **Dynamic Memory Management:** Using `new` and `delete` to allocate and deallocate memory at runtime. Proper management is critical to avoid memory leaks and dangling pointers.

## Syntax Examples

### Copy Constructor

```
 MyClass(const MyClass& other) {
    m_data = new int[other.m_size];
    for (size_t i = 0; i < other.m_size; ++i) {
        m_data[i] = other.m_data[i];
    }
}
```

### Destructor

```
~MyClass() {
    delete[] m_arr; // Use delete[] for arrays
    delete ptr;     // Use delete for single objects
}
```

## Copy Assignment Operator

```
 MyClass& operator=(const MyClass& other) {
    if (this == &other) return *this;
    delete[] m_data;
    m_data = new int[other.m_size];
    for (size_t i = 0; i < other.m_size; ++i) {
        m_data[i] = other.m_data[i];
    }
    return *this;
}
```

## Operator Overloading

```
// Binary operator
MyClass operator+(const MyClass& other) const {
    MyClass result;
    // Perform addition logic
    return result;
}

// Subscript operator
char& operator[](size_t index) {
    return m_arr[index];
}

// Comparison operator
bool operator==(const MyClass& other) const {
    return /* comparison logic */;
}
```

## Lab Exercises

### 1. [35 marks] Exercise 1: Enhanced String Class

Extend the `string` class from Lab 1 with proper memory management and operator overloading. The class should dynamically allocate memory for character storage.

#### What You Already Have from Lab 1:

These methods are already implemented and should remain in your class:

- `string(const char* c)` – Parameterized constructor
- `size_t size()` – Returns the length of the string
- `void print()` – Prints the string to console
- `bool isPalindrome()` – Checks if string is a palindrome
- `char at(size_t index)` – Returns character at index
- `bool contains(char c)` – Checks if character exists in string
- `int find(char c)` – Returns index of first occurrence of character
- `void append(const string& other)` – Appends another string to current string

**Note:** Make sure to update these methods to be `const` where appropriate (methods that don't modify the object should be marked `const`).

## Required New Implementations for This Lab:

<code>~string()</code>	<b>Destructor:</b> Frees dynamically allocated memory using <code>delete[] m_arr</code> .
<code>string(const string&amp; other)</code>	<b>Copy constructor:</b> Creates a deep copy of another string object. Allocate new memory for <code>m_arr</code> and copy all characters from <code>other.m_arr</code> . Set <code>m_len</code> to <code>other.m_len</code> .
<code>string&amp; operator=(const string&amp; other)</code>	<b>Copy assignment operator:</b> Handles assignment between existing objects. Must: (1) Check for self-assignment, (2) Free old memory, (3) Allocate new memory and perform deep copy, (4) Return <code>*this</code> .
<code>char&amp; operator[](size_t index)</code>	<b>Subscript operator:</b> Returns a reference to the character at the given index. Should allow both reading and writing. Return <code>m_arr[index]</code> .
<code>string operator+(const string&amp; other) const</code>	<b>Concatenation operator:</b> Returns a new string that is the concatenation of the current string and <code>other</code> .
<code>bool operator==(const string&amp; other) const</code>	<b>Equality operator:</b> Returns <code>true</code> if both strings have the same length and same characters at each position.
<code>bool operator!=(const string&amp; other) const</code>	<b>Inequality operator:</b> Returns <code>true</code> if strings are not equal. Can be implemented as <code>return !(*this == other);</code>

## Example Usage and Testing:

```

1 int main() {
2     string s1("Hello");
3     string s2(" World");
4
5     // Test 1: Copy constructor
6     cout << "Test 1: Copy Constructor" << endl;
7     string s3 = s1;
8     cout << "Original s1: ";
9     s1.print();
10    cout << endl;
11    cout << "Copy s3: ";
12    s3.print();
13    cout << endl;
14
15    // Modify the copy and verify original is unchanged
16    s3[0] = 'h';
17    cout << "After modifying s3[0] to 'h': " << endl;

```

```
18 cout << "Original s1: ";
19 s1.print();      // Should still print: Hello
20 cout << endl;
21 cout << "Modified s3: ";
22 s3.print();      // Should print: hello
23 cout << endl << endl;
24
25 // Test 2: Assignment operator
26 cout << "Test 2: Assignment Operator" << endl;
27 string s4("Initial");
28 cout << "Before assignment, s4: ";
29 s4.print();
30 cout << endl;
31 s4 = s2;
32 cout << "After s4 = s2, s4: ";
33 s4.print();      // Should print: World
34 cout << endl << endl;
35
36 // Test 3: Self-assignment
37 cout << "Test 3: Self-assignment" << endl;
38 s4 = s4;
39 cout << "After s4 = s4, s4: ";
40 s4.print();
41 cout << endl << endl;
42
43 // Test 4: [] operator
44 cout << "Test 4: Subscript Operator []" << endl;
45 cout << "s1[1] = " << s1[1] << endl;
46 s1[0] = 'J';
47 cout << "After s1[0] = 'J': ";
48 s1.print();      // Should print: Jello
49 cout << endl << endl;
50
51 // Test 5: + operator
52 cout << "Test 5: Concatenation Operator +" << endl;
53 string s5("Hello");
54 string s6(" World");
55 string s7 = s5 + s6;
56 cout << "s5: ";
57 s5.print();
58 cout << endl;
59 cout << "s6: ";
60 s6.print();
61 cout << endl;
62 cout << "s5 + s6 = ";
63 s7.print();      // Should print: Hello World
64 cout << endl << endl;
65
```

```

66 // Test 6: == and != operators
67 cout << "Test 6: Equality Operators == and !=" << endl;
68 string s8("Hello");
69 string s9("Hello");
70 string s10("World");
71
72 if (s8 == s9) {
73     cout << "s8 and s9 are equal" << endl;
74 } else {
75     cout << "s8 and s9 are NOT equal" << endl;
76 }
77
78 if (s8 != s10) {
79     cout << "s8 and s10 are NOT equal" << endl;
80 } else {
81     cout << "s8 and s10 are equal" << endl;
82 }
83 cout << endl;
84
85 // Test 7: Existing methods from Lab 1
86 cout << "Test 7: Lab 1 Methods Still Working" << endl;
87 cout << "Size of s7: " << s7.size() << endl;
88 cout << "s7 contains 'W': " << (s7.contains('W') ? "Yes" : "No") << endl;
89 cout << "Index of 'W' in s7: " << s7.find('W') << endl;
90 cout << "Is s8 palindrome? " << (s8.isPalindrome() ? "Yes" : "No") << endl;
91 cout << "Character at index 1 in s8: " << s8.at(1) << endl;
92
93 // Test 8: Append
94 string s11("Hello");
95 string s12(" There");
96 s11.append(s12);
97 cout << "After s11.append(s12): ";
98 s11.print();
99 cout << endl << endl;
100
101 // Test 9: Chain operations
102 cout << "Test 9: Chained Operations" << endl;
103 string first("C");
104 string second("++");
105 string third(" Programming");
106 string result = first + second + third;
107 cout << "Result of chaining: ";
108 result.print(); // Should print: C++ Programming
109 cout << endl;
110
111 return 0;
112 }
```

**Marks Rubric (Total: 35 marks)**

Destructor	3	Correctly frees dynamically allocated memory using <code>delete[]</code>
Copy Constructor	6	<ul style="list-style-type: none"> <li>• Allocates new memory (2 marks)</li> <li>• Performs deep copy correctly (3 marks)</li> <li>• Handles null terminator (1 mark)</li> </ul>
Copy Assignment Operator	8	<ul style="list-style-type: none"> <li>• Self-assignment check (2 marks)</li> <li>• Frees old memory (2 marks)</li> <li>• Allocates new memory and deep copies (3 marks)</li> <li>• Returns <code>*this</code> correctly (1 mark)</li> </ul>
Subscript Operator [ ]	3	Returns reference allowing read/write access
Addition Operator +	5	<ul style="list-style-type: none"> <li>• Creates new string object (1 mark)</li> <li>• Correctly concatenates (3 marks)</li> <li>• Handles null terminator (1 mark)</li> </ul>
Equality Operator ==	3	Correctly compares length and characters
Inequality Operator !=	2	Correctly implements using == operator
Testing & Demonstration	5	<ul style="list-style-type: none"> <li>• Comprehensive test cases (3 marks)</li> <li>• Deep copy verification (1 mark)</li> <li>• Self-assignment test (1 mark)</li> </ul>
<b>Total</b>	<b>35</b>	

2. [35 marks] **Exercise 2: Enhanced Fraction Class with Dynamic Memory**

Modify the Fraction class (from `Fraction.cpp` taught in lectures) to use **dynamic memory allocation** for the numerator and denominator. Instead of storing `int num` and `int den` directly, you will store `int* num` and `int* den`.

## Required Implementations:

<code>Fraction()</code>	Default constructor: Dynamically allocate memory for <code>num</code> and <code>den</code> , initialize to 0/1.
<code>Fraction(int n, int d)</code>	Parameterized constructor: Dynamically allocate memory and initialize. Ensure denominator is never 0 (use ternary operator). Reduce the fraction to simplest form.
<code>Fraction(const Fraction&amp; other)</code>	<b>Copy constructor:</b> Perform deep copy by allocating new memory and copying values from <code>other</code> .
<code>~Fraction()</code>	<b>Destructor:</b> Free dynamically allocated memory using <code>delete num; delete den;</code>
<code>Fraction&amp; operator=(const Fraction&amp; other)</code>	<b>Copy assignment operator:</b> Handle self-assignment, copy values (memory already exists, just assign values), return <code>*this</code> .
<code>int get_num() const</code>	Return the numerator by dereferencing: <code>return *num;</code>
<code>int get_den() const</code>	Return the denominator by dereferencing: <code>return *den;</code>
<code>void set_num(int n)</code>	Set numerator: <code>*num = n</code> ; then reduce the fraction.
<code>void set_den(int d)</code>	Set denominator: <code>*den = (d == 0) ? 1 : d</code> ; then reduce the fraction.
<code>Fraction operator+(const Fraction&amp; other) const</code>	Return sum of two fractions. Use local variables (e.g., <code>int n, int d</code> ), NOT pointers to avoid variable shadowing.
<code>Fraction operator-(const Fraction&amp; other) const</code>	Return difference: <code>(num × other.den - other.num × den) / (den × other.den)</code> .
<code>Fraction operator*(const Fraction&amp; other) const</code>	Return product: <code>(num × other.num) / (den × other.den)</code> .
<code>Fraction operator/(const Fraction&amp; other) const</code>	Return quotient: multiply by reciprocal. <code>(num × other.den) / (den × other.num)</code> .
<code>bool operator==(const Fraction&amp; other) const</code>	Return <code>true</code> if fractions are equal (both reduced to simplest form).
<code>bool operator!=(const Fraction&amp; other) const</code>	Return <code>true</code> if fractions are not equal.
<code>bool operator&lt;(const Fraction&amp; other) const</code>	Return <code>true</code> if current fraction is less than <code>other</code> . Use cross-multiplication.
<code>bool operator&gt;(const Fraction&amp; other) const</code>	Return <code>true</code> if current fraction is greater than <code>other</code> . Use cross-multiplication.
<code>void print() const</code>	Display fraction in format "num/den".
<code>double to_decimal() const</code>	Return decimal representation: <code>return (double)(*num) / (*den);</code> Note: Cast BEFORE division.

## Helper Methods:

```
private:
    int gcd(int a, int b) const {
        a = abs(a);
        b = abs(b);
        if (b == 0) return a;
        return gcd(b, a % b);
    }

    void reduce() {
        int g = gcd(*num, *den);
        *num /= g;
        *den /= g;

        if (*den < 0) {
            *num = -(*num);
            *den = -(*den);
        }
    }
```

## Example Usage:

```
1 int main() {
2     Fraction f1(1, 2);      // 1/2
3     Fraction f2(1, 3);      // 1/3
4     Fraction f3(2, 4);      // 2/4 -> should reduce to 1/2
5
6     cout << "Test 1: Copy Constructor" << endl;
7     Fraction f4 = f1;
8     cout << "f4 = ";
9     f4.print();
10    cout << endl << endl;
11
12    cout << "Test 2: Assignment Operator" << endl;
13    Fraction f5;
14    f5 = f2;
15    cout << "f5 = ";
16    f5.print();
17    cout << endl << endl;
18
19    cout << "Test 3: Arithmetic Operators" << endl;
20    Fraction sum = f1 + f2;
21    cout << "1/2 + 1/3 = ";
22    sum.print();
```

```
23     cout << endl;
24
25     Fraction diff = f1 - f2;
26     cout << "1/2 - 1/3 = ";
27     diff.print();
28     cout << endl;
29
30     Fraction prod = f1 * f2;
31     cout << "1/2 * 1/3 = ";
32     prod.print();
33     cout << endl;
34
35     Fraction quot = f1 / f2;
36     cout << "1/2 / 1/3 = ";
37     quot.print();
38     cout << endl << endl;
39
40     cout << "Test 4: Comparison Operators" << endl;
41     if (f1 == f3) {
42         cout << "1/2 and 2/4 are equal (after reduction)" << endl;
43     }
44
45     if (f2 < f1) {
46         cout << "1/3 is less than 1/2" << endl;
47     }
48
49     if (f1 > f2) {
50         cout << "1/2 is greater than 1/3" << endl;
51     }
52     cout << endl;
53
54     cout << "Test 5: Complex Expressions" << endl;
55     Fraction f6(1, 4);
56     Fraction result1 = f1 + f2 * f6;
57     cout << "1/2 + (1/3 * 1/4) = ";
58     result1.print();
59     cout << endl;
60
61     Fraction result2 = f1 * f2 + f6;
62     cout << "(1/2 * 1/3) + 1/4 = ";
63     result2.print();
64     cout << endl << endl;
65
66     cout << "Test 6: Decimal Conversion" << endl;
67     cout << "1/2 as decimal: " << f1.to_decimal() << endl;
68     cout << "1/3 as decimal: " << f2.to_decimal() << endl;
69     cout << endl;
70
```

```
71 cout << "Test 7: Setters and Getters" << endl;
72 Fraction f7(6, 8);
73 cout << "f7 before set: ";
74 f7.print();
75 cout << endl;
76 f7.set_num(1);
77 cout << "After set_num(1): ";
78 f7.print();
79 cout << endl;
80
81 return 0;
82 }
```

**Marks Rubric (Total: 35 marks)**

Default Constructor	2	Correctly allocates memory and initializes to 0/1
Parameterized Constructor	4	<ul style="list-style-type: none"> <li>• Allocates memory (1 mark)</li> <li>• Handles zero denominator (1 mark)</li> <li>• Calls reduce() (2 marks)</li> </ul>
Copy Constructor	4	<ul style="list-style-type: none"> <li>• Allocates new memory (2 marks)</li> <li>• Performs deep copy (2 marks)</li> </ul>
Destructor	3	Correctly deletes both pointers separately
Copy Assignment Operator	5	<ul style="list-style-type: none"> <li>• Self-assignment check (1 mark)</li> <li>• Correct value assignment without deleting (3 marks)</li> <li>• Returns <code>*this</code> (1 mark)</li> </ul>
Helper Methods (gcd, reduce)	3	<ul style="list-style-type: none"> <li>• gcd correctly implemented (1 mark)</li> <li>• reduce() works properly (2 marks)</li> </ul>
Arithmetic Operators (+, -, *, /)	8	<ul style="list-style-type: none"> <li>• Each operator correct (2 marks each)</li> <li>• No variable shadowing</li> <li>• Division uses reciprocal</li> </ul>
Comparison Operators (<, >, ==, !=)	3	All comparison operators work correctly
Getters and Setters	2	get_num, get_den, set_num, set_den implemented
to_decimal()	1	Correctly casts before division
print()	1	Displays fraction correctly
Testing & Demonstration	4	<ul style="list-style-type: none"> <li>• Tests all operators (2 marks)</li> <li>• Tests complex expressions (1 mark)</li> <li>• Tests reduction (1 mark)</li> </ul>
<b>Total</b>	<b>35</b>	

### 3. [30 marks] Exercise 3: Dynamic Array Class

Create a `DynamicArray` class that manages a dynamically allocated integer array with automatic resizing capabilities.

#### Class Specifications:

The class should maintain:

- `int* m_data`: Pointer to dynamically allocated array
- `size_t m_size`: Current number of elements
- `size_t m_capacity`: Total allocated capacity

#### Required Implementations:

<code>DynamicArray()</code>	Default constructor: Initialize with capacity 10, size 0, allocate memory.
<code>DynamicArray(size_t initial_capacity)</code>	Parameterized constructor: Initialize with given capacity, size 0, allocate memory.
<code>DynamicArray(const DynamicArray&amp; other)</code>	<b>Copy constructor:</b> Deep copy of array data (allocate new memory, copy elements).
<code>~DynamicArray()</code>	<b>Destructor:</b> Free allocated memory using <code>delete[]</code> .
<code>DynamicArray&amp; operator=(const DynamicArray&amp; other)</code>	<b>Copy assignment operator:</b> Handle assignment with deep copy. Must check self-assignment, free old memory, allocate new memory, and copy data.
<code>void push_back(int value)</code>	Add element to end. If size == capacity, double the capacity first by calling <code>resize()</code> .
<code>void pop_back()</code>	Remove last element (decrease size). Check if array is not empty first.
<code>int&amp; operator[](size_t index)</code>	Access element at index (allows modification). Return <code>m_data[index]</code> .
<code>size_t size() const</code>	Return current number of elements.
<code>size_t capacity() const</code>	Return current capacity.
<code>bool empty() const</code>	Return <code>true</code> if size is 0.
<code>void clear()</code>	Remove all elements (set size to 0, but keep capacity and allocated memory).
<code>void print() const</code>	Print all elements in format: [1, 2, 3, 4]

## Helper Method:

```
private:
void resize(size_t new_capacity) {
    int* new_data = new int[new_capacity];

    for (size_t i = 0; i < m_size; ++i) {
        new_data[i] = m_data[i];
    }

    delete[] m_data;

    m_data = new_data;
    m_capacity = new_capacity;
}
```

## Example Usage:

```
1 int main() {
2     cout << "Test 1: Default Constructor and push_back" << endl;
3     DynamicArray arr;
4
5     for (int i = 1; i <= 15; ++i) {
6         arr.push_back(i * 10);
7         cout << "Added " << i * 10 << " | Size: " << arr.size()
8             << ", Capacity: " << arr.capacity() << endl;
9     }
10
11    cout << "Array contents: ";
12    arr.print();
13    cout << endl << endl;
14
15    cout << "Test 2: Copy Constructor (Deep Copy)" << endl;
16    DynamicArray arr2 = arr;
17    arr2[0] = 999;
18
19    cout << "Original array: ";
20    arr.print();
21    cout << endl;
22    cout << "Copied array (modified): ";
23    arr2.print();
24    cout << endl << endl;
25
26    cout << "Test 3: Assignment Operator" << endl;
27    DynamicArray arr3;
28    arr3.push_back(1);
```

```
29     arr3.push_back(2);
30     cout << "arr3 before assignment: ";
31     arr3.print();
32     cout << endl;
33
34     arr3 = arr;
35     cout << "arr3 after assignment: ";
36     arr3.print();
37     cout << endl << endl;
38
39     cout << "Test 4: Subscript Operator []" << endl;
40     cout << "arr[0] = " << arr[0] << endl;
41     cout << "arr[5] = " << arr[5] << endl;
42     arr[5] = 777;
43     cout << "After arr[5] = 777: ";
44     arr.print();
45     cout << endl << endl;
46
47     cout << "Test 5: pop_back" << endl;
48     arr.pop_back();
49     arr.pop_back();
50     cout << "After two pop_back() calls: ";
51     arr.print();
52     cout << " | Size: " << arr.size() << endl << endl;
53
54     cout << "Test 6: empty() and clear()" << endl;
55     cout << "Is arr empty? " << (arr.empty() ? "Yes" : "No") << endl;
56     arr.clear();
57     cout << "After clear(), size: " << arr.size()
58         << ", capacity: " << arr.capacity() << endl;
59     cout << "Is arr empty now? " << (arr.empty() ? "Yes" : "No") << endl << endl;
60
61     cout << "Test 7: Parameterized Constructor" << endl;
62     DynamicArray arr4(5);
63     cout << "arr4 initial capacity: " << arr4.capacity() << endl;
64     for (int i = 1; i <= 8; ++i) {
65         arr4.push_back(i);
66     }
67     cout << "After adding 8 elements, capacity: " << arr4.capacity() << endl;
68     cout << "arr4: ";
69     arr4.print();
70     cout << endl << endl;
71
72     cout << "Test 8: Self-assignment" << endl;
73     arr4 = arr4;
74     cout << "After self-assignment, arr4: ";
75     arr4.print();
76     cout << endl;
```

```
77
78     return 0;
79 }
```

**Marks Rubric (Total: 30 marks)**

Default Constructor	2	Allocates memory and initializes members correctly
Parameterized Constructor	2	Accepts capacity and initializes correctly
Copy Constructor	5	<ul style="list-style-type: none"> <li>• Allocates new memory (2 marks)</li> <li>• Deep copies all elements (2 marks)</li> <li>• Copies size and capacity (1 mark)</li> </ul>
Destructor	2	Correctly frees memory using <code>delete[]</code>
Copy Assignment Operator	6	<ul style="list-style-type: none"> <li>• Self-assignment check (1 mark)</li> <li>• Frees old memory (2 marks)</li> <li>• Allocates new memory (1 mark)</li> <li>• Deep copies data (1 mark)</li> <li>• Returns <code>*this</code> (1 mark)</li> </ul>
<code>resize()</code> Helper	3	<ul style="list-style-type: none"> <li>• Allocates new array (1 mark)</li> <li>• Copies old data (1 mark)</li> <li>• Frees old memory and updates pointer (1 mark)</li> </ul>
<code>push_back()</code>	3	<ul style="list-style-type: none"> <li>• Checks capacity and resizes if needed (2 marks)</li> <li>• Adds element correctly (1 mark)</li> </ul>
<code>pop_back()</code>	1	Decreases size (checks if not empty)
Subscript Operator <code>[]</code>	2	Returns reference allowing read/write
<code>size()</code> , <code>capacity()</code> , <code>empty()</code>	1	All three methods implemented correctly
<code>clear()</code>	1	Sets size to 0, keeps capacity
<code>print()</code>	1	Displays array in proper format
Testing & Demonstration	4	<ul style="list-style-type: none"> <li>• Tests resizing behavior (1 mark)</li> <li>• Tests deep copy (1 mark)</li> <li>• Tests all operations (2 marks)</li> </ul>
<b>Total</b>	<b>30</b>	

## Total Marks Distribution

Exercise	Marks
Exercise 1: Enhanced String Class	35
Exercise 2: Enhanced Fraction Class	35
Exercise 3: Dynamic Array Class	30
<b>Total</b>	<b>100</b>

Good luck with your lab! Remember to test thoroughly and ensure no memory leaks.