[1]// 1

Lecture Review

In this lab, you will explore advanced object-oriented programming concepts including **Friend Functions**, **Friend Classes**, and **Casting Operators**. These features allow controlled access to private data and enable seamless type conversions.

1. Friend Functions

A **friend function** is a non-member function that has access to the private and protected members of a class. It is declared inside the class using the `friend` keyword.

Why use friend functions?

- To implement binary operators where the left operand is not an object of the class
- To allow external utility functions to access private data
- To implement I/O operators (« and »)

Example:

```
class Complex {
private:
    double real, imag;
public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // Friend function declaration
    friend Complex add(const Complex& c1, const Complex& c2);
    friend void display(const Complex& c);
};

// Friend function definition (outside the class)
Complex add(const Complex& c1, const Complex& c2) {
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}

void display(const Complex& c) {
    cout << c.real << " + " << c.imag << "i" << endl;
}
```

2. Friend Classes

A **friend class** is a class whose all member functions have access to the private and protected members of another class.

Example:

```
class BankAccount {
private:
    double balance;
    string accountNumber;
public:
    BankAccount(string acc, double bal)
        : accountNumber(acc), balance(bal) {}

    // Auditor can access all private members
    friend class Auditor;
};

class Auditor {
public:
    void checkAccount(const BankAccount& account) {
        cout << "Account: " << account.accountNumber << endl;
        cout << "Balance: " << account.balance << endl;
    }
};
```

Important Notes:

- Friendship is not mutual (if A is friend of B, B is not automatically friend of A)
- Friendship is not inherited
- Friendship is not transitive (if A is friend of B and B is friend of C, A is not friend of C)

3. Casting Operators (Type Conversion Operators)

Casting operators allow objects of a class to be converted to other types (built-in or user-defined).

3.1 Conversion to Built-in Types

Use `operator` keyword followed by the target type.

Example:

```
class Fraction {
private:
    int numerator, denominator;
public:
    Fraction(int n, int d) : numerator(n), denominator(d) {}

    // Conversion to double
    operator double() const {
```

```
        return (double)numerator / denominator;
    }

    // Conversion to int
    operator int() const {
        return numerator / denominator;
    }
};

int main() {
    Fraction f(3, 4);
    double d = f;     // Calls operator double()
    int i = f;        // Calls operator int()
    cout << d << endl; // 0.75
    cout << i << endl; // 0
}
```

3.2 Conversion from Other Types (Constructor Conversion)

A single-argument constructor can act as a conversion operator.

Example:

```
class Distance {
private:
    double meters;
public:
    // Conversion from double to Distance
    Distance(double m) : meters(m) {}

    // Conversion from Distance to double
    operator double() const {
        return meters;
    }
};

int main() {
    Distance d = 100.5;  // Calls constructor
    double m = d;        // Calls operator double()
}
```

Key Syntax Summary

```
// Friend function
friend ReturnType functionName(parameters);

// Friend class
friend class ClassName;
```

```
// Casting operator (no return type specified)
operator TargetType() const {
    // conversion logic
}
```

Lab Exercises

1. [15 marks] **Exercise 1: Complex Number Calculator**

   You are developing a mathematical software library that handles complex numbers. Complex numbers are widely used in engineering, physics, and signal processing. Each complex number has a real part and an imaginary part.

   Your task is to create a `Complex` class that supports:

   - Basic arithmetic operations using operator overloading
   - Conversion to built-in types using casting operators
   - A friend function to compute the magnitude of a complex number

   ### Class Specifications:

```
class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r = 0, double i = 0);

    // Arithmetic operators
    Complex operator+(const Complex& other) const;
    Complex operator-(const Complex& other) const;
    Complex operator*(const Complex& other) const;

    // Casting operators
    operator double() const;  // Returns magnitude
    operator int() const;     // Returns real part as integer

    // Friend function
    friend double magnitude(const Complex& c);
    friend void display(const Complex& c);
};
```

   ### Requirements:

   1. **Constructor**: Initialize real and imaginary parts
   2. **Operator +**: Add two complex numbers: $(a + bi) + (c + di) = (a + c) + (b + d)i$
   3. **Operator -**: Subtract two complex numbers
   4. **Operator ***: Multiply two complex numbers: $(a + bi) \times (c + di) = (ac - bd) + (ad + bc)i$
   5. **operator double()**: Return magnitude $\sqrt{real^2 + imag^2}$
   6. **operator int()**: Return real part as integer

7. **Friend function magnitude()**: Calculate and return magnitude (can access private members)
8. **Friend function display()**: Display in format "a + bi" or "a - bi"

Example Usage:

```
int main() {
    Complex c1(3, 4);
    Complex c2(1, 2);

    Complex sum = c1 + c2;
    Complex diff = c1 - c2;
    Complex prod = c1 * c2;

    cout << "c1: "; display(c1);
    cout << "c2: "; display(c2);
    cout << "Sum: "; display(sum);
    cout << "Difference: "; display(diff);
    cout << "Product: "; display(prod);

    double mag = c1;  // Casting to double
    int r = c1;       // Casting to int

    cout << "Magnitude of c1: " << mag << endl;
    cout << "Real part of c1 (as int): " << r << endl;
    cout << "Magnitude using friend: " << magnitude(c1) << endl;

    return 0;
}
```

Marks Rubric (Total: 15 marks)

Constructor	2	Properly initializes real and imaginary parts
Arithmetic Operators (+, -, *)	4	Correct implementation of all three operators
Casting operator double()	2	Returns correct magnitude
Casting operator int()	2	Returns real part as integer
Friend function magnitude()	2	Accesses private members and calculates correctly
Friend function display()	2	Properly formats output (handles negative imaginary)
Testing	1	Comprehensive test in main()
Total	**15**	

2. [15 marks] **Exercise 2: Bank Account System with Auditor**

   You are designing a banking system where customer accounts must be kept secure. Account holders cannot directly access their balance or account numbers for security reasons. However, a trusted `Auditor` class needs full access to verify account information and detect fraud.

   Implement a `BankAccount` class with private data and an `Auditor` friend class that can inspect all account details.

Class Specifications:

```
class BankAccount {
private:
    string accountNumber;
    string holderName;
    double balance;
    bool isFrozen;

public:
    BankAccount(string accNum, string name, double initialBalance);

    void deposit(double amount);
    bool withdraw(double amount);
    void display() const;

    // Auditor can access all private members
    friend class Auditor;
};

class Auditor {
public:
    void inspectAccount(const BankAccount& account);
    bool detectSuspiciousActivity(const BankAccount& account);
    void freezeAccount(BankAccount& account);
    void unfreezeAccount(BankAccount& account);
};
```

Requirements:

1. **BankAccount constructor**: Initialize all member variables
2. **deposit()**: Add amount to balance (only if account is not frozen)
3. **withdraw()**: Deduct amount if sufficient balance and account not frozen. Return true if successful
4. **display()**: Show holder name and balance (not account number for security)
5. **Auditor::inspectAccount()**: Display all account details including private account number
6. **Auditor::detectSuspiciousActivity()**: Return true if balance is negative or unusually high ($>$ 1,000,000)

7. **Auditor::freezeAccount()**: Set `isFrozen` to true
8. **Auditor::unfreezeAccount()**: Set `isFrozen` to false

Example Usage:

```
int main() {
    BankAccount acc1("ACC001", "Ali Khan", 50000);
    BankAccount acc2("ACC002", "Sara Ahmed", 1500000);

    acc1.deposit(10000);
    acc1.withdraw(5000);
    acc1.display();

    Auditor auditor;

    cout << "\n--- Audit Report ---" << endl;
    auditor.inspectAccount(acc1);

    if (auditor.detectSuspiciousActivity(acc2)) {
        cout << "Suspicious activity detected on ACC002!" << endl;
        auditor.freezeAccount(acc2);
    }

    acc2.withdraw(100);  // Should fail (frozen)
    auditor.unfreezeAccount(acc2);
    acc2.withdraw(100);  // Should succeed

    return 0;
}
```

Marks Rubric (Total: 15 marks)

Constructor	2	Initializes all members correctly
deposit() and withdraw()	3	Proper validation and frozen account check
Friend class declaration	2	Correct use of friend keyword
Auditor::inspectAccount()	2	Accesses private members correctly
Auditor::detectSuspiciousActivity()	2	Proper logic for detection
Freeze/Unfreeze methods	2	Correctly modifies private isFrozen
Testing	2	Demonstrates all functionality
Total	**15**	

3. [10 marks] **Exercise 3: Temperature Converter**

   You are building a weather application that needs to convert temperatures between Celsius and Fahrenheit seamlessly. Create a `Temperature` class that uses casting operators to enable automatic conversion.

Class Specifications:

```
class Temperature {
private:
    double celsius;

public:
    // Constructor takes Celsius value
    Temperature(double c = 0);

    // Casting operators
    operator double() const;  // Returns Celsius value
    operator int() const;     // Returns Celsius as integer

    // Friend function to get Fahrenheit
    friend double toFahrenheit(const Temperature& t);

    void display() const;
};
```

Requirements:

1. **Constructor**: Initialize temperature in Celsius
2. **operator double()**: Return temperature in Celsius as double
3. **operator int()**: Return temperature in Celsius as integer (truncated)
4. **Friend function toFahrenheit()**: Convert Celsius to Fahrenheit using formula: $F = C \times \frac{9}{5} + 32$
5. **display()**: Show temperature in both Celsius and Fahrenheit

Example Usage:

```
int main() {
    Temperature t1(25.5);
    Temperature t2(100);
    Temperature t3(-40);

    t1.display();
    t2.display();
    t3.display();

    double c = t1;  // Casting to double
```

```
    int ci = t1;    // Casting to int

    cout << "Temperature in Celsius (double): " << c << endl;
    cout << "Temperature in Celsius (int): " << ci << endl;
    cout << "Temperature in Fahrenheit: " << toFahrenheit(t1) << endl;

    return 0;
}
```

Marks Rubric (Total: 10 marks)

Constructor	1	Initializes celsius correctly
operator double()	2	Returns celsius value
operator int()	2	Returns truncated integer
Friend function toFahrenheit()	3	Correct conversion formula
display()	1	Shows both formats
Testing	1	Tests all functionality
Total	**10**	

4. [15 marks] **Exercise 4: Library Book Management System**

   You are developing a digital library system. Books have titles, authors, ISBN numbers, and page counts. The system needs proper encapsulation to protect book data, and should support various operations through operator overloading.

Class Specifications:

```
class Book {
private:
    string title;
    string author;
    string isbn;
    int pages;
    bool isAvailable;

public:
    Book(string t, string a, string i, int p);

    // Getters
    string getTitle() const;
    string getAuthor() const;
    string getISBN() const;
    int getPages() const;
    bool getAvailability() const;

    // Setters
    void setAvailability(bool status);

    void borrowBook();
    void returnBook();
    void displayInfo() const;

    // Operator overloading
    bool operator<(const Book& other) const;   // Compare by pages
    bool operator>(const Book& other) const;   // Compare by pages
    bool operator==(const Book& other) const;  // Compare by ISBN

    Book operator+(const Book& other) const;   // Combine pages (anthology)
};
```

Requirements:

1. **Constructor**: Initialize all members, set `isAvailable` to true
2. **Getters**: Return respective private data members
3. **setAvailability()**: Modify availability status

4. **borrowBook()**: Set `isAvailable` to false if available, display message
5. **returnBook()**: Set `isAvailable` to true, display message
6. **displayInfo()**: Display all book information in formatted manner
7. **operator** $<$: Compare books by page count (return true if current book has fewer pages)
8. **operator** $>$: Compare books by page count (return true if current book has more pages)
9. **operator** $==$: Compare books by ISBN (return true if ISBNs match)
10. **operator** $+$: Create a new book combining page counts (useful for anthology collections)

Example Usage:

```
int main() {
    Book b1("C++ Programming", "Bjarne Stroustrup",
            "978-0321563840", 1376);
    Book b2("The C++ Language", "Bjarne Stroustrup",
            "978-0201889543", 1040);
    Book b3("Effective C++", "Scott Meyers",
            "978-0321334879", 320);

    cout << "--- Book Information ---" << endl;
    b1.displayInfo();
    b2.displayInfo();
    b3.displayInfo();

    // Test borrowing
    b1.borrowBook();
    cout << "\nAfter borrowing b1:" << endl;
    cout << "Is b1 available? " << (b1.getAvailability() ? "Yes" : "No")
         << endl;

    b1.returnBook();

    // Test comparison operators
    if (b3 < b1) {
        cout << "\n" << b3.getTitle() << " has fewer pages than "
             << b1.getTitle() << endl;
    }

    if (b1 > b2) {
        cout << b1.getTitle() << " has more pages than "
             << b2.getTitle() << endl;
    }

    Book b4("C++ Programming", "Author", "978-0321563840", 500);
    if (b1 == b4) {
        cout << "\nb1 and b4 have the same ISBN" << endl;
    }
```

```
    // Combine books (anthology)
    Book anthology = b1 + b2;
    cout << "\nCombined anthology:" << endl;
    anthology.displayInfo();

    return 0;
}
```

Marks Rubric (Total: 15 marks)

Constructor	2	Initializes all members correctly
Getters and Setters	2	Proper encapsulation
borrowBook() and returnBook()	2	Correct state management
displayInfo()	1	Formatted output
operator < and operator >	3	Both comparison operators work correctly
operator ==	2	Compares by ISBN
operator +	2	Creates combined book correctly
Testing	1	Comprehensive tests
Total	**15**	

5. [15 marks] **Exercise 5: Distance Measurement System**

   You are developing a measurement utility for a construction application. The system needs to handle distances in meters but also support automatic conversion to centimeters, feet, and inches through casting operators. The class should support arithmetic and comparison operations.

Class Specifications:

```
class Distance {
private:
    double meters;

public:
    Distance(double m = 0);

    // Arithmetic operators
    Distance operator+(const Distance& other) const;
    Distance operator-(const Distance& other) const;
    Distance operator*(double factor) const;  // Scale distance
    Distance operator/(double divisor) const; // Scale distance

    // Comparison operators
    bool operator>(const Distance& other) const;
    bool operator<(const Distance& other) const;
    bool operator==(const Distance& other) const;

    // Casting operators
    operator double() const;  // Returns meters
    operator int() const;     // Returns centimeters (as int)

    // Utility methods
    double toFeet() const;
    double toInches() const;
    void display() const;
};
```

Requirements:

1. **Constructor**: Initialize distance in meters
2. **operator +**: Add two distances
3. **operator -**: Subtract two distances (result cannot be negative)
4. **operator ***: Multiply distance by a factor
5. **operator /**: Divide distance by a divisor (check for division by zero)
6. **operator >, <, ==**: Compare distances
7. **operator double()**: Return distance in meters

8. **operator int()**: Return distance in centimeters (1 meter = 100 cm)
9. **toFeet()**: Convert meters to feet (1 meter = 3.28084 feet)
10. **toInches()**: Convert meters to inches (1 meter = 39.3701 inches)
11. **display()**: Show distance in all units (meters, cm, feet, inches)

Example Usage:

```
int main() {
    Distance d1(5.5);
    Distance d2(3.2);

    Distance sum = d1 + d2;
    Distance diff = d1 - d2;
    Distance scaled = d1 * 2;
    Distance halved = d1 / 2;

    cout << "Distance 1: "; d1.display();
    cout << "Distance 2: "; d2.display();
    cout << "Sum: "; sum.display();
    cout << "Difference: "; diff.display();
    cout << "Scaled (x2): "; scaled.display();
    cout << "Halved (/2): "; halved.display();

    // Casting operators
    double m = d1;   // Casting to double (meters)
    int cm = d1;     // Casting to int (centimeters)

    cout << "\nDistance in meters: " << m << endl;
    cout << "Distance in centimeters: " << cm << endl;
    cout << "Distance in feet: " << d1.toFeet() << endl;
    cout << "Distance in inches: " << d1.toInches() << endl;

    // Comparison
    if (d1 > d2) {
        cout << "\nd1 is greater than d2" << endl;
    }

    if (d1 == Distance(5.5)) {
        cout << "d1 equals 5.5 meters" << endl;
    }

    return 0;
}
```

Marks Rubric (Total: 15 marks)

Constructor	1	Initializes meters correctly
Arithmetic operators (+, -, *, /)	4	All four operators work correctly
Comparison operators (>, <, ==)	3	All three operators work correctly
operator double()	2	Returns meters
operator int()	2	Returns centimeters
toFeet() and toInches()	2	Correct conversion formulas
display()	1	Shows all units
Testing	1	Tests all functionality
Total	**15**	

6. [15 marks] **Exercise 6: Student Grade Management System**

   You are developing a student information system for a university. Each student has personal information and academic records stored in a dynamic array. This exercise focuses on the **Rule of Three**: proper implementation of copy constructor, destructor, and copy assignment operator to manage dynamic memory correctly.

Class Specifications:

```
class Student {
private:
    string name;
    string erpId;
    double* grades;     // Dynamic array
    int numCourses;
    double cgpa;

public:
    Student(string n, string id, int courses);
    Student(const Student& other);      // Copy constructor (Deep copy)
    ~Student();                         // Destructor
    Student& operator=(const Student& other);  // Assignment operator

    // Utility methods
    void setGrade(int courseIndex, double grade);
    void calculateCGPA();
    void display() const;

    // Getters
    string getName() const;
    string getErpId() const;
    double getCGPA() const;
    int getNumCourses() const;
    double getGrade(int index) const;

    // Analysis methods
    double getHighestGrade() const;
    double getLowestGrade() const;
    int countFailingGrades() const;  // Grades < 50
};
```

Requirements:

1. **Constructor**: Dynamically allocate array for grades, initialize all grades to 0
2. **Copy Constructor**: Perform **deep copy** of grades array (allocate new memory and copy values)
3. **Destructor**: Free dynamically allocated memory using `delete[]`

4. **Assignment Operator**: Implement with:
   - Self-assignment check
   - Free existing memory
   - Deep copy from source object
   - Return `*this`
5. **setGrade()**: Set grade at specific index with validation (0-100)
6. **calculateCGPA()**: Calculate average of all grades and store in cgpa
7. **display()**: Show name, ERP ID, CGPA, and all grades
8. **getHighestGrade()**: Return the highest grade from all courses
9. **getLowestGrade()**: Return the lowest grade from all courses
10. **countFailingGrades()**: Return count of grades below 50

Example Usage:

```
int main() {
    Student s1("Ali Ahmed", "26184", 5);

    s1.setGrade(0, 85);
    s1.setGrade(1, 90);
    s1.setGrade(2, 78);
    s1.setGrade(3, 92);
    s1.setGrade(4, 88);
    s1.calculateCGPA();

    cout << "--- Student Information ---" << endl;
    s1.display();

    cout << "\n--- Grade Analysis ---" << endl;
    cout << "Highest Grade: " << s1.getHighestGrade() << endl;
    cout << "Lowest Grade: " << s1.getLowestGrade() << endl;
    cout << "Failing Courses: " << s1.countFailingGrades() << endl;

    // Test deep copy (Copy Constructor)
    cout << "\n--- Testing Deep Copy ---" << endl;
    Student s2 = s1;  // Calls copy constructor
    s2.setGrade(0, 95);
    s2.calculateCGPA();

    cout << "\nOriginal Student (s1):" << endl;
    s1.display();
    cout << "\nCopied Student (s2) after modification:" << endl;
    s2.display();

    // Test assignment operator
```

```
    cout << "\n--- Testing Assignment Operator ---" << endl;
    Student s3("Sara Khan", "26185", 5);
    s3 = s1;  // Calls assignment operator
    s3.setGrade(1, 100);
    s3.calculateCGPA();

    cout << "\nOriginal (s1):" << endl;
    s1.display();
    cout << "\nAssigned (s3) after modification:" << endl;
    s3.display();

    // Test self-assignment
    s3 = s3;  // Should handle gracefully
    cout << "\nAfter self-assignment, s3:" << endl;
    s3.display();

    return 0;
}
```

Marks Rubric (Total: 15 marks)

Constructor	2	Dynamic allocation and initialization
Copy Constructor	3	Deep copy of grades array
Destructor	2	Properly frees memory
Assignment Operator	4	Self-check + deep copy + return *this
setGrade() and calculateCGPA()	2	Correct logic and validation
Analysis methods	1	getHighest, getLowest, countFailing
Getters and display()	1	All work correctly
Testing	2	Tests deep copy, assignment, self-assignment
Total	**15**	

7. [15 marks]

8. [15 marks] **Exercise 7: Time Duration System**

   You are building a time tracking application for a project management tool. The system needs to handle time durations (hours and minutes), support arithmetic operations, provide comparisons, and implement the Rule of Three for proper object management.

Class Specifications:

```
class Time {
private:
    int* hours;
    int* minutes;

    void normalize();  // Ensure minutes < 60, adjust hours

public:
    Time(int h = 0, int m = 0);
    Time(const Time& other);         // Copy constructor
    ~Time();                         // Destructor
    Time& operator=(const Time& other);  // Assignment operator

    // Arithmetic operators
    Time operator+(const Time& other) const;
    Time operator-(const Time& other) const;
    Time operator*(int factor) const;    // Multiply time
    Time operator/(int divisor) const;   // Divide time

    // Comparison operators
    bool operator==(const Time& other) const;
    bool operator>(const Time& other) const;
    bool operator<(const Time& other) const;

    // Utility methods
    int getTotalMinutes() const;
    double getTotalHours() const;  // Decimal hours
    void display() const;
};
```

Requirements:

1. **Constructor**: Dynamically allocate memory for hours and minutes, call normalize()
2. **Copy Constructor**: Deep copy both hours and minutes pointers
3. **Destructor**: Free both dynamically allocated pointers
4. **Assignment Operator**: Check self-assignment, copy values (memory already exists)

5. **normalize()**: If minutes >= 60, convert to hours (e.g., 90 min = 1h 30m)
6. **operator +**: Add two times and normalize result
7. **operator -**: Subtract times (result cannot be negative, return 0:0 if negative)
8. **operator ***: Multiply time by integer factor
9. **operator /**: Divide time by integer divisor
10. **operator ==, >, <**: Compare times
11. **getTotalMinutes()**: Return total time in minutes
12. **getTotalHours()**: Return total time in decimal hours (e.g., 2h 30m = 2.5)
13. **display()**: Show time in format "Xh Ym"

Example Usage:

```
int main() {
    Time t1(2, 45);
    Time t2(1, 30);
    Time t3(0, 150);  // Should normalize to 2h 30m

    cout << "Time 1: "; t1.display();
    cout << "Time 2: "; t2.display();
    cout << "Time 3 (after normalization): "; t3.display();

    // Arithmetic operations
    Time sum = t1 + t2;
    Time diff = t1 - t2;
    Time doubled = t1 * 2;
    Time halved = t1 / 2;

    cout << "\nSum: "; sum.display();
    cout << "Difference: "; diff.display();
    cout << "Doubled: "; doubled.display();
    cout << "Halved: "; halved.display();

    // Utility methods
    cout << "\nTotal minutes in t1: " << t1.getTotalMinutes() << endl;
    cout << "Total hours in t1 (decimal): " << t1.getTotalHours() << endl;

    // Comparison operators
    if (t1 > t2) {
        cout << "\nt1 is greater than t2" << endl;
    }

    if (t1 == Time(2, 45)) {
        cout << "t1 equals 2h 45m" << endl;
    }
```

```
    // Test copy constructor
    cout << "\n--- Testing Copy Constructor ---" << endl;
    Time t4 = t1;
    t4 = t4 + Time(1, 0);  // Add 1 hour

    cout << "Original t1: "; t1.display();
    cout << "Modified copy t4: "; t4.display();

    // Test assignment operator
    cout << "\n--- Testing Assignment Operator ---" << endl;
    Time t5(0, 0);
    t5 = t2;
    cout << "Assigned t5: "; t5.display();

    // Self-assignment
    t5 = t5;
    cout << "After self-assignment: "; t5.display();

    return 0;
}
```

Marks Rubric (Total: 15 marks)

Constructor with dynamic allocation	2	Allocates hours and minutes pointers
Copy Constructor	2	Deep copy of both pointers
Destructor	2	Deletes both pointers correctly
Assignment Operator	2	Self-check + assigns values
normalize()	1	Converts minutes to hours correctly
Arithmetic operators (+, -, *, /)	3	All four work with normalization
Comparison operators (==, >, <)	2	All three work correctly
Utility methods	1	getTotalMinutes, getTotalHours, display
Testing	1	Tests Rule of Three thoroughly
Total	**15**	

Total Marks Distribution

Exercise	Marks
Exercise 1: Complex Number Calculator	15
Exercise 2: Bank Account with Auditor	15
Exercise 3: Temperature Converter	10
Exercise 4: Library Book Management	15
Exercise 5: Distance Measurement System	15
Exercise 6: Student Grade Management	15
Exercise 7: Time Duration System	15
Total	**100**

Good luck with your lab! Remember to test all friend functions and casting operators thoroughly.