Name: ______________________________

Question	Points	Score
1	10	
2	10	
3	10	
4	10	
5	10	
6	60	
7	41	
Total:	151	

Name: ______________________________

2143 OOP Exam 1
Spring 2025

1. (10 points) Below is some usage dealing with a typical college student. Based on the usage, create the class definition as seen on the study guide and as discussed in class.

   Usage:

```
Student S1("John", "Doe", 20);
Student S2("Aarya", "Chakrabarti", 19);
S1.changeMajor("Computer Science");
S2.increaseTotalHours(12);
```

Solution:

Rubric

- **Basic Class Structure**
  - Full class declaration, correct syntax for class and member declaration
  - Data members correctly declared as private
  - Data members have correct data type
  - Public access specifier used for methods
- **Constructors**
  - Implemented base, and whatever usage showed you to impplement.
- **Setters and Getters**
  - All setters correctly declared with appropriate parameter types
  - All getters correctly declared, returning appropriate types
- **Additional Methods**
  - Correct implementation of `changeMajor`
  - Correct implementation of `increaseTotalHours`

```
class Student{
// default private
    string fname;
    string lname;
    int age;
    string major;
    int hours;
public:
    //setters
    void setFname(string);
    void setLname(string);
    void setAge(int);
    void setMajor(string);
    void setHours(int);
    // getters
    string getFname(string);
    string getLname(string);
    int getAge(string);
    string getMajor(string);
    int getHours(int);
    // additional methods from usage
    void changeMajor(string);
    void increaseTotalHours(int);
};
```

2. (10 points) Below is some usage dealing with popular books. Based on the usage, create the class definition as seen on the study guide and as discussed in class.

Usage:

```
Book B0;
Book B1("1984", "George Orwell", 328, 1949);
Book B2("To Kill a Mockingbird", "Harper Lee", 281);
B2.setIsbn("978-1-4028-9462-6");
B2.setPrice(23.99);
```

Solution:

Rubric

- **Basic Class Structure**
  - Full class declaration, correct syntax for class and member declaration
  - Data members correctly declared as private
  - Data members have correct data type
  - Public access specifier used for methods
- **Constructors**
  - Implemented base, and whatever usage showed you to impplement.
- **Setters and Getters**
  - All setters correctly declared with appropriate parameter types
  - All getters correctly declared, returning appropriate types
- **Additional Methods**
  - Correct implementation of `changeMajor`
  - Correct implementation of `increaseTotalHours`

```
class Book {
private:
    float price;
    int pages;
    int year;
    string author;
    string isbn;
    string title;
public:
    // Constructors
    // Usage showed multiple constructors with differing values, I'm implementing only two:
    Book();
    Book(string , string , int , int);
    Book(string , string , int);

    // Getters and Setters
    float getPrice();
    int getPages();
    int getYear();
    string getAuthor();
    string getISBN();
    string getTitle();
    void setPrice(float);
    void setPages(int);
    void setYear(int);
    void setAuthor(string);
    void setISBN(string);
    void setTitle(string);
};
```

3. (10 points) Assume there is a class called **Pixel** and it has three data members:

```
class Pixel{
    int Red;   //possible values are 0-255
    int Green; //possible values are 0-255
    int Blue;  //possible values are 0-255
public:
    // constructor(s)
    // methods
};
```

Where:

- If we want a **red** pixel we set Red=255 Green=0 and Blue=0.
- If we want a **purple** pixel we set Red=128 Green=0 and Blue=128.
- If we want a **pink** pixel we set Red=255,Green=192,Blue=203.
- If we want to **GrayScale** the pixel we set Red,Green, and Blue = the average of the three color channels.
  In this case: Red=255,Green=192,Blue=203 would now be Red=216,Green=216,Blue=216

I would like you to write a method for the Pixel class called **GrayScale** that turns a pixel into a shade of gray. To do this you can simply average the three color values together, then replace the Red, Green, and Blue data members with the average of all three color channels, so that each color channel has the same value.

Solution:

Rubric (10 points)

- **Basic Method Declaration**
  - Correct function signature
  - Placement of method inside `Pixel` class
- **GrayScale Logic (5 points)**
  - Correctly summing the three color channels
  - Correctly calculating the average of the color channels
  - Assigning the average to `Red`, `Green`, and `Blue`
- **Readability and Clarity**
  - Clear, readable code
  - Use of comments to explain the process

```
/**
* Long version...
* The description implied that we are converting the pixel to gray scale
* internally by changing each channel to the avg. So we return void.
*/
void Pixel::GrayScale(){
    int sum = 0;    // sum of all three channels
    int avg = 0;    // not really needed but makes readable code
    sum = (Red + Green + Blue); // total up all color chanels
    avg = sum / 3;  // Calculate the avg;
    Red = Green = Blue = avg; // Assign same value to all 3 channels
}

/**
* Short version ...
*/
void Pixel::GrayScale(){
    // one line thats less readable (or not?!)
    Red = Green = Blue = (Red + Green + Blue) / 3;
}
```

4. (10 points) If you were to write a method to return all three color channels from the pixel class, what would you use as a return type, and why?

Note: There are a thousand ways to solve most problems in our field. Solutions can be based on many things: readability, speed, simplicity, self documenting, utilizing efficient algorithms and so on. S0, some are good, some are bad, and some are pretty much the same. This question is me showing you this idea. Plus I was interested in seeing how you handled it. I'm not sure if I will factor this in to your grade.

Solution: ***If your answer is either in the list below, or you came up with a novel, but practical solution, you WILL get points.***

Rubric (10 points)

- **Return Type Justification**
  - 4 points: Provides a strong, clear explanation of the return type choice, explaining why it is suitable for the task.
  - 3 points: Gives a reasonable justification with a good explanation, but lacks full detail or misses some considerations.
  - 2 points: Return type explanation is somewhat unclear, only briefly touches on benefits but lacks depth.
  - 1 point: Little to no explanation for the return type, or the explanation is largely incorrect.
- **Return Type Correctness**
  - 3 points: Chooses a valid return type that effectively returns all three color channels.
  - 2 points: Return type is valid but less ideal for the task.
  - 1 point: Chooses a return type that works but introduces significant complexity or is a poor choice.
- **Code Example and Clarity**
  - 3 points: Provides a complete, syntactically correct code example, demonstrating how the chosen return type is used.
  - 2 points: Provides a code example, but with minor issues or lacks clarity.
  - 1 point: Provides an incomplete or incorrect code example, or it doesn't demonstrate the return type well.

There are a many ways to return multiple values from a function simply by packaging them together in something that counts as ONE thing.

Possible Return Types:

1. Return a `struct`
Example:

```
 struct Color {
     int r;
     int g;
     int b;
     Color(int r,int g,int b): r{r},g{g},b{b}
 };

Color getColor(){
    return Color(r,g,b);
}
```

Explanation:

- A **struct** is probably the best solution since it keeps the data members names, only contains whats necessary, BUT it could add new data members if the need arises: like *alpha* for alpha channel or *grayColorVal* for gray scale value.

---

2. Return a `std::array<int, 3>`
Example:

```
std::array<int, 3> getColor() {
    return {r, g, b};
}
```

Explanation:

- A **std::array** is decent as well but there are no names involved so we lose readability. You can however iterate over each value easily which could be a requirement by the user by chance? Doubtful, but it gets the job done.

---

3. Return a `std::vector<int>`
Example:

```
std::vector<int> getColor() {
    return {r, g, b};
}
```

Explanation:

- A **std::vector** could work, but given the fact that a vector can dynamically grow and shrink we are adding ”overhead” (extra methods, storage size) to our return type which should be simple.

4. Return an `std::map<std::string, int>` or `std::unordered_map<std::string, int>`
Example:

```
std::map<std::string, int> getColor() {
    return { {"r", r}, {"g", g}, {"b", b} };
}
```

Explanation:

- A **std::map** stores what are called key:value pairs, in our case r,g,b would keys, and there ints would be the values. This also means you could access the map like: std::cout¡¡colorMap['r']¡¡to access the red channel. Pretty cool since most of you have only seen this syntax for arrays, and the keys are always indexes (integers). But like the vector, if simple is what we are going for, this may be overkill. But it is readable!
- The two map types shown above: ordered v unordered maps are similar when accessing the container. The difference is that an ordered map remembers the order in which you place items in the map. This might be important, but is does add overhead, which is unnecessary

5. Return a `std::tuple<int, int, int>`
Example:

```
std::tuple<int, int, int> getColor() {
    return std::make_tuple(r, g, b);
}
```

Explanation:

- A **tuple** is a container type like an array, or vector, but you can store different types in the same tuple (like python lets you do in lists). However we only need ints and accessing the tuple is clunky. For example you need to do the following just to access a single item (red in this instance) `std::get<0>(tuple)`

6. Return a `Pixel` object itself
Example:

```
Pixel getColor() {
    return *this;
}
```

Explanation:

- This also is a decent solution, however its only decent when the data type is small / lightweight (not a lot of stored data). So, for the ***Pixel*** class it works, but remember some classes can store thousands or more of items and take up many thousands or millions of bytes and to return one of those would be extremely wasteful.

7. Return by `std::pair`
Example:

```
std::pair<int, std::pair<int, int>> getColor() {
    return std::make_pair(r, std::make_pair(g, b));
}
```

Explanation:

- Ok ... I got this one from Chat GPT (as well as the Tuple). Pairs are great for ..... ***pairs***. We will go into pairs later and use them for something that matches their use case. Remember what I said up top ... some are good some are bad, this one is bad.

8. Return a C-style array (not recommended) :)
Example:

```
int* getColor() {
    static int colors[3] = {r, g, b};
    return colors;
}
```

Explanation:

- So when answered this in class and was brainstorming, I came up with the same list you see with two exceptions: Pairs and Tuples. Tuples was an exciting discovery because I didn't know c++ had added that type to their arsenal! Pairs was just stupid. Having said all that, this return type (which was my goto, is a poor choice. I'll let Chat GPT explain and remind you guys .... it never stops (the learning).

Chat GPT:

- - A C-style array can be returned, but this method is **dangerous** because it exposes raw pointers, leading to potential issues with memory management and safety. Modern C++ favors safer containers like `std::array` or `std::vector`.
- - **When to use:** This method should be avoided unless you are working in a constrained environment where C++ containers aren't available.

Which Return Type is Best?

- **Overall Best Answer:** ***It depends.***
- **Best for Readability**: Returning a `struct` or `Pixel` is the most readable solution, especially if you want to add more properties later or group ***'r', 'g', 'b'*** logically.
- **Best for Simplicity:** `std::array<int, 3>` provides a clean, efficient way to return the values in a simple, fixed-size container.
- **Best for Flexibility:** If you expect to need future flexibility (e.g., to add more channels like alpha or others), `std::vector<int>` or `std::map<std:` allows for dynamic adjustments but comes at a performance cost.

For this particular scenario,

- `std::array<int, 3>` strikes the best balance between simplicity and performance,
- `struct` or `Pixel` is a close second for clarity and readability.
- However, one could also argue that a `std::tuple<int, int, int>` could be a simple yet expandable and robust solution for the fact that if you wanted to start adding return items, you could add any data type: string:ColorName, string:HexValue, etc..

5. (10 points) Overload the **ostream** operator for a movie class so that when a movie type is printed it looks similar to:

```
Trilogy: True
Duration: 136
Year: 1999
Budget: 63000000
Director: The Wachowskis
Title: The Matrix
```

You can assume the class has data members to match the output labels (trilogy,duration,year,etc.).

Solution:

Rubric (10 points)

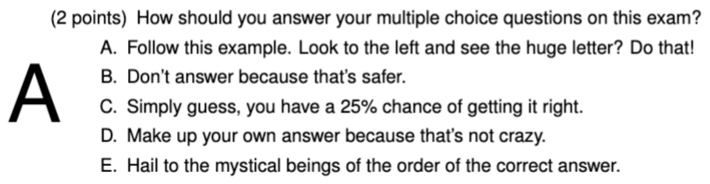
1. **Correct Declaration**
   - Properly declares `friend std::ostream& operator<<(std::ostream& os, const Movie& m);` inside the class.
   - If defined outside the class, signatures match exactly (return type, parameter types, const correctness).
2. **Definition Syntax & Return**
   - Correct function signature: `std::ostream& operator<<(std::ostream& os, const Movie& m)`.
   - Returns `os` at the end so it can be chained (`std::cout << movie << " additional text";`).
3. **Access & Use of Class Data Members**
   - Uses the correct data members in the correct order (i.e., trilogy, duration, year, budget, director, title).
   - Prints them in a way that matches the specified output format (labels, spacing, newlines).
   - Handles printing booleans or enumerations (like "Trilogy: True") in a sensible way.
4. **Formatting/Output Clarity**
   - Each data member is labeled clearly (e.g., `os << "Trilogy: " << m.trilogy << std::endl;`).
   - Includes line breaks, spacing, or punctuation as specified.
5. **Code Style & Readability**
   - Reasonable indentation, no obvious syntax errors, minimal confusion.
   - Everything is easy to follow for someone reading or maintaining the code.

```cpp
class Movie {
private:
   // stuff
public:

    // Declare the operator<< as a friend. The signature must match the definition if you
    // do define the method outside of the class.
    friend std::ostream& operator<<(std::ostream& os, const Movie& m);
};

std::ostream& operator<<(std::ostream& os, const Movie& m) {
    // write everything to the os instance we passed in:
    os << "Trilogy:"<<m.trilogy<<std::endl;
    os << "Duration:"<<m.duration<<std::endl;
    os << "Year:"<<m.year<<std::endl;
    os << "Budget:"<<m.budget<<std::endl;
    os << "Director:"<<m.director<<std::endl;
    os << "Title:"<<m.title<<std::endl;
    // then return os to be printed to stdout.
    return os;
}
```

6. Answer the multiple choice questions below. Read them carefully, and answer them on this sheet like the example show below.



A

(2 points) How should you answer your multiple choice questions on this exam?

A. Follow this example. Look to the left and see the huge letter? Do that!
B. Don't answer because that's safer.
C. Simply guess, you have a 25% chance of getting it right.
D. Make up your own answer because that's not crazy.
E. Hail to the mystical beings of the order of the correct answer.

(1) (2 points) If you need to check to see if two linked lists are equal to each other, what approach would you take when overloading the comparison operator? (Assume the lists are of equal length).

A. Compare only the head nodes, since they may have the sam
B. Traverse one list, checking that the other list has all the first lists values.
C. Traverse both lists, checking each node for equality.
D. Convert the lists to arrays and compare them using a standard array comparison.

(2) (2 points) Which of the following operations is efficient in both array-based and list-based stacks?

A. Pushing an element to the front
B. Accessing a random element
C. Pushing an element to the top
D. Deleting an element from the middle

(3) (2 points) When do you need a copy constructor in a class?

A. When the class does not have dynamically allocated memory.
B. When the class manages resources like dynamically allocated memory.
C. When the class only contains primitive types.
D. When the default constructor is deleted.

(4) (2 points) Which type of copy is performed by the default copy constructor?

A. Deep copy
B. Shallow copy
C. Recursive copy
D. Lazy copy

(5) (2 points) What is a potential issue of using the default copy constructor in a class with pointers?

A. Pointer dereferencing becomes more complex.
B. It leads to self-assignment issues (pointing to itself).
C. It results in multiple objects sharing (pointing to) the same memory locations.
D. It automatically deep copies the memory.

(6) (2 points) What is the purpose of the `if(this == &other)` check in the overloaded assignment operator?

A. Checks for self copy which could lead to accidental deletion of the object's own memory.
B. To optimize memory allocation.
C. To ensure that the destructor is called properly.
D. To avoid performing a deep copy.

(7) (2 points) Which rule suggests that if a class requires a copy constructor, it likely also requires a destructor and an overloaded assignment operator?

A. Rule of Data Members
B. Rule of Constructors
C. Rule of Three
D. Copy Constructor Rule

(8) (2 points) In the case of a class managing dynamic memory, when should you overload the assignment operator?

A. When you need to perform a shallow copy.
B. When the class contains only primitive types.
C. When you need to implement a deep copy.
D. To manage multiple objects sharing the same memory.

(9) (2 points) What happens if a class with a pointer does not define a copy constructor and the default one is used?

A. A deep copy will be automatically performed because C++ is smart.
B. The default constructor will be called, but perform a deep copy.
C. Multiple objects will share the same pointer (point to same memory).

D. The pointer will be deleted immediately.

(10) (2 points) What is the main difference between public and private methods in object-oriented programming?

A. Public methods can only be accessed by objects of the same class.
B. Private methods can be accessed from anywhere in the program.
C. Public methods can be accessed from outside the class, while private methods cannot.
D. Private methods must be static, while public methods cannot be static.

(11) (2 points) Which of the following is an example of a situation where you would use a private method?

A. When only private data members need to be initialized.
B. When work needs done with method(s) that are not privy to public users.
C. A method that allows a user to add two fractions together.
D. A function to retrieve the numerator of a fraction.

(12) (2 points) What is the main purpose of making a method private in a class?

A. To increase the performance of the program.
B. To protect internal logic from being accessed or modified externally.
C. To allow other classes to modify internal data.
D. To make the method accessible to derived classes only.

(13) (2 points) What is a good reason to make the `operator+` method in a `Fraction` class public?

A. To hide the implementation details from the user.
B. To allow users to easily add fractions by using `fraction1 + fraction2`.
C. To prevent users from accessing the `+` operator outside of the class.
D. To ensure the method can only be used inside the class.

(14) (2 points) In a class, which of the following actions would likely be performed by a public method?

A. Calculating the greatest common divisor of two numbers.
B. Displaying the result of an operation to the user.
C. Sorting an internal array used only for temporary calculations.
D. Accessing a file that the user should not interact with directly.

(15) (2 points) Which data structure has faster access times?

A. Linked List
B. Array
C. Doubly Linked List
D. Scippy Doo List

(16) (2 points) Which data structure works better with many insertions and deletions?

A. Linked List
B. Array
C. Tree List
D. Scrappy Doo Tree

(17) (7 points) Given:

```
class Sample {
    int* ptr;
public:
    Sample(int val) { ptr = new int(val); }
    ~Sample() { delete ptr; }
    Sample(const Sample& other) { ptr = new int(*other.ptr); }
    void display() { std::cout << *ptr << std::endl; }
};
int main() {
    Sample obj1(10);
    Sample obj2 = obj1; // Copy constructor
    obj2.display();
    obj1.display();
}
```

What will be the output of the code above?

A. 10, 10

B. Garbage values due to double-free error

C. The program crashes

D. 0, 0

(18) (7 points) Given:

```
class A {
    int* ptr;
public:
    A(int val) { ptr = new int(val); }
    ~A() { delete ptr; }
    A& operator=(const A& other) {
        if (this != &other) {
            delete ptr;
            ptr = new int(*other.ptr);
        }
        return *this;
    }
    void display() { std::cout << *ptr << std::endl; }
};
int main() {
    A obj1(5);
    A obj2(10);
    obj2 = obj1;  // Assignment operator
    obj2.display();
    obj1.display();
}
```

What will be the output?

A. 5, 5

B. Garbage values due to unhandled self-assignment

C. The program crashes

D. 10, 10

(19) (7 points) Given:

```
#include <iostream>
#include <stack>
int main() {
    std::stack<int> s;
    s.push(1);
    s.push(2);
    s.push(3);
    s.pop();
    std::cout << s.top() << std::endl;
    return 0;
}
```

What will the output be when this code is run?

A. 1
B. 2
C. 3
D. The program crashes

(20) (7 points) Given:

```
#include <iostream>
#include <queue>
int main() {
    std::queue<int> q;
    q.push(10);
    q.push(20);
    q.pop();
    std::cout << q.front() << std::endl;
    return 0;
}
```

What will the output be when this code is run?

A. 10
B. 20
C. The program crashes
D. 0

7. Choose appropriate word for each blank.

Method	Constructor	Overload	Instance	Object
Shallow Copy	Deep Copy	Private	Public	Destructor
LIFO	FIFO	Inheritance	Polymorphism	Encapsulation
Copy Constructor	Assignment Operator	Definition	Pointers	Self Assignment
Terminates	Friend Keyword	New	Delete	Shallow Copy

(1) (5 points) When assigning one object to another, a **shallow copy** only duplicates the memory addresses of dynamically allocated resources, while a **deep copy** allocates new memory and duplicates the actual data.

(2) (5 points) In a class, members that should only be accessible by methods of the same class are declared as **private** , while members that should be accessible from outside the class are declared as **public** .

(3) (5 points) Stacks use the **LIFO** principle, while a queues use the **FIFO** principle.

(4) (3 points) The ability of a function or operator to behave differently based on the types or number of arguments is known as **polymorphism** .

(5) (7 points) The **"Rule of Three"** states that if a class requires a **destructor** , a **copy constructor** , or an **assignment** operator, it likely needs all three due to resource management needs such as dynamic memory.

(6) (3 points) The assignment operator should check for **self-assignment** to avoid issues like overwriting oneself, which can lead to unexpected behavior or errors such as freeing memory that's still in use.

(7) (3 points) When defining a class in C++, a method with the same name as the class is called a **constructor** .

(8) (3 points) In C++, the **friend** keyword allows a function or class to access the private and protected members of another class, thus bypassing the concept of **encapsulation** , which normally restricts direct access to an object's internal state.

(9) (7 points) When you create a class the code that blueprints the class is the class's **Definition** and when you create an **Instance** of that class we call it an **Object** .