Total Points: 30
Point Distribution:

- **Problem 1:** 15 points
- **Problem 2:** 15 points

---

1. Problem 1

The code for the `complexNumber` class is provided below.
You are required to **implement operator overloading** functions to facilitate the following operations on any complex numbers `A`, `B`, and `C`:

1. `A == B`
   → Overload the **Boolean operator** `==` to test if `A` and `B` are equal.
2. `C = A * B`
   → Overload the **multiplication operator** `*`.
3. `A = B`
   → Overload the **assignment operator** `=`.
4. `cin >> A >> B`
   → Overload the **stream operator** `>>` to take user input for the real and imaginary parts of a complex number.
   (Hint: You might want to create a friend function to facilitate this.)

Also, write a **static function** for the `complexNumber` class which initializes the `numCount` variable. Your function should take an **integer argument provided by the user** and assign this value to `numCount`.

1.1. Provided Code

```cpp
#include <iostream>
using namespace std;

class complexNumber {
private:
    static int numCount;
    int Re, Imag;

public:
    complexNumber() { numCount++; } // constructor with no argument

    complexNumber(int Real, int Imaginary) {
        numCount++;
        cout << "constructor with argument is called" << endl;
        Re = Real;
        Imag = Imaginary;
    }

    ~complexNumber() {
        cout << "destructor is called" << endl;
        numCount--;
    }

    complexNumber(complexNumber &anycomplex) {
        numCount++;
        cout << endl;
        cout << "Copy constructor called" << endl;
        Re = anycomplex.Re;
        Imag = anycomplex.Imag;
    }

    void printComplex() {
        cout << "number of active complex numbers = " << numCount
             << " and present number = " << Re << " + j" << Imag << endl;
    }

    complexNumber operator+(complexNumber &b) {
        complexNumber temp;
        cout << "overloaded + is called" << endl;
        temp.Re = Re + b.Re;
        temp.Imag = Imag + b.Imag;
        return temp;
    }

    complexNumber operator++() {
        complexNumber temp;
        cout << "overloaded ++ (pre) called" << endl;
        temp.Re = ++Re;
        temp.Imag = ++Imag;
        return temp;
```

```
    }

    complexNumber operator++(int) {
        complexNumber temp;
        cout << "overloaded ++ (post) called" << endl;
        temp.Re = Re++;
        temp.Imag = Imag++;
        return temp;
    }

    friend ostream &operator<<(ostream &out, complexNumber &somecomplex) {
        cout << "overloaded << called" << endl;
        out << somecomplex.Re << "+ j" << somecomplex.Imag;
        return out;
    }
};

int complexNumber::numCount = 0;

void printC(complexNumber a) { a.printComplex(); }

int main() {
    complexNumber A, B(1,2), C(2,3);
    B.printComplex();
    ++B;
    printC(C);

    A = B + C;
    A = B++;

    A.printComplex();
    B.printComplex();

    cout << A << B;

    cin.get();
    return 0;
}
```

---

2. Problem 2

You are to design an **abstract class** called `Employee` with the following **protected members**:

2.1. Data Members

- `char *name`

- `long int ID`

2.2. Constructors

- **Default constructor** – initializes data members to default values.
- **Copy constructor**

2.3. Methods

- `setPerson(char *n, long int id)` – allows user to set information for each person.
- `virtual void print()` – virtual function that prints the data attributes of the class.
- **Destructor**

2.4. Derived Classes

Define two classes derived from `Employee`:

- **Manager**
- **Secretary**

Each class:

- Inherits all members from `Employee`.
- Has its own data members and member functions.

2.4.1. Additional Members

- **Manager:** `degree` (string such as *Diploma, Bachelor, Master, Doctor*).
- **Secretary:** `contract` (Boolean – 1 for permanent, 0 for temporary).

All member functions of the derived classes should **override** their base class functions.

2.5. Testing Code

```
int main() {
    Employee *p = new Manager("Bruce Lee", 234567, "Dr.");
    p->print();

    Secretary p2;
    p2.setPerson("Wilma Jones", 341256, "permanent");

    delete p;

    p = &p2;
    p->print();

    return 0;
}
```

2.6. Questions

- Do you observe any **polymorphic behavior**?
- What do you **learn** from this example?