PART B – OPERATOR OVERLOADING, FRIEND FUCNTIONS, INHERITANCE [20+30= 50 Marks]

Question: 1 [30 Marks]

Find the output of the following code snippets. In case you find the code to have some error, mention it in the output section.

1.	class Count {
	  private:
	  int value;
	  public:
	  Count() : value(5) {}
	  void operator ++ () {
	  ++value;
	  }
	  void operator ++ (int) {
	  value++;
	  }
	  void display() {
	  cout << "Count: " << value << endl;
	  }
	};
	
	int main() {
	  Count count1;
	  count1++;
	  count1.display();
	  ++count1;
	  count1.display();
	  return 0;
	}
		Count: 6
		Count: 7

2.	class Box
	{
	  private:
	  int length;
	  public:
	  Box (): length (0) {}
	  friend int printLength (Box); //friend function
	};
	int printLength (Box b)
	{
	  b. length +=10;
	  return b. length;
	}
	int main ()
	{
	  Box b;
		Length of box:10

	cout <<" Length of box:" <<printLength (b)<<endl; return 0; }	

3.	class B; //forward declaration.
	class A
	{
	int x;
	public:
	void setdata (int i)
	{
	x=i;
	}
	friend void max (A, B); //friend function.
	} ;
	class B
	{
	int y;
	public:
	void setdata (int i)
	{
	y=i;
	}
	friend void max (A, B);
	};
	void max (A a, B b)
	{
	if (a.x >= b.y)
	std:: cout<< a.x << std::endl;
	else
	std::cout<< b.y << std::endl;
	}
	int main ()
	{
	A a;
	B b;
	a. setdata (10);
	b. setdata (20);
	max (a, b);
	return 0;
	}
		20

4.	class A
	{
	int x=4;
	friend class B; //friend class
	};
		value of x is:4

	class B
	{
	public:
	void display (A &a)
	{
	cout<<"value of x is:" <<a.x;
	}
	};
	int main ()
	{
	A a;
	B b;
	b. display (a);
	return 0;
	}	

5.	class space
	{
	int x;
	int y;
	int z;
	public:
	void setdata (int a, int b, int c);
	void display(void);
	friend void operator- (space &s);
	};
	void space ::setdata (int a, int b, int c)
	{
	x=a; y=b; z=c;
	}
	void space::display(void)
	{
	cout<<x<<" "<<y<<" "<<z<<"\n";
	}
	void operator- (space &s)
	{
	s.x =- s.x;
	s.y =- s.y;
	s.z =- s.z;
	}
	int main ()
	{
	space s;
	s. setdata (5,2,9);
	cout<<"s:";
	s. display ();
	-s;
	cout<<"-s:";
		s:5 2 9
		
		-s: -5 -2 -9

	s. display (); return 0; }	

6.	class Distance {
	public:
	int feet, inch;
	Distance(int f, int i)
	{
	this->feet = f;
	this->inch = i;
	}
	
	void operator-()
	{
	feet--;
	inch--;
	cout << "\nFeet & Inches(Decrement): " <<
	feet << "'" << inch;
	}
	};
	
	// Driver Code
	int main()
	{
	Distance d1(8, 9);
	
	// Use (-) unary operator by
	// single operand
	-d1;
	return 0;
	}
		Feet & Inches(Decrement): 7'8

7.	class Cents
	{
	private:
	int m_cents {};
	
	public:
	Cents(int cents) : m_cents{ cents } { }
	friend Cents operator+(const Cents& c1,
	const Cents& c2);
	
	int getCents() const { return m_cents; }
	};
		I have 14 cents.

	Cents operator+(const Cents& c1, const Cents& c2)
	{
	return c1.m_cents + c2.m_cents;
	}
	
	int main()
	{
	Cents cents1{ 6 };
	Cents cents2{ 8 };
	Cents centsSum{ cents1 + cents2 };
	std::cout << "I have " << centsSum.getCents() << " cents.\n";
	
	return 0;
	}	

8.	class MinMax
	{
	private:
	int m_min {};
	int m_max {};
	
	public:
	MinMax(int min, int max)
	: m_min { min }, m_max { max }
	{ }
	
	int getMin() const { return m_min; }
	int getMax() const { return m_max; }
	
	friend MinMax operator+(const MinMax& m1, const MinMax& m2);
	friend MinMax operator+(const MinMax& m, int value);
	friend MinMax operator+(int value, const MinMax& m);
	};
	
	MinMax operator+(const MinMax& m1, const MinMax& m2)
	{
	int min{ m1.m_min < m2.m_min ? m1.m_min : m2.m_min };
		Result: (3, 16)

	int max{ m1.m_max > m2.m_max ? m1.m_max : m2.m_max };
	
	return { min, max };
	}
	
	MinMax operator+(const MinMax& m, int value)
	{
	int min{ m.m_min < value ? m.m_min : value };
	
	int max{ m.m_max > value ? m.m_max : value };
	
	return { min, max };
	}
	
	MinMax operator+(int value, const MinMax& m)
	{
	return m + value;
	}
	
	int main()
	{
	MinMax m1{ 10, 15 };
	MinMax m2{ 8, 11 };
	MinMax m3{ 3, 12 };
	
	MinMax mFinal{ m1 + m2 + 5 + 8 + m3 + 16 };
	
	std::cout << "Result: (" << mFinal.getMin() << ", " <<
	mFinal.getMax() << ")\n";
	
	return 0;
	}	

9.	class Shape {
	protected:
	int width;
	int height;
	public:
	void setDimensions(int w, int h) {
	width = w;
	height = h;
	}
		Area of Square: 25
		Perimeter of Square: 20

	};
	
	class Rectangle : public Shape {
	public:
	  int getArea() {
	  return width * height;
	  }
	};
	
	class Square : public Rectangle {
	public:
	  int getPerimeter() {
	  return 4 * width;
	  }
	};
	
	int main() {
	  Square square;
	  square.setDimensions(5, 5);
	  cout << "Area of Square: " << square.getArea()
	<< endl;
	  cout << "Perimeter of Square: " <<
	square.getPerimeter() << endl;
	  return 0;
	}	

10.	lass Shape {
	public:
	  virtual void displayArea() {
	  cout << "Shape Area" << endl;
	  }
	};
	
	class Circle : public Shape {
	private:
	  double radius;
	public:
	  Circle(double r) {
	  radius = r;
	  }
	  void displayArea() {
	  double area = 3.14 * radius * radius;
	  cout << "Circle Area: " << area << endl;
	  }
	};
	
	class Rectangle : public Shape {
	private:
	  double length;
		Shape Area
		Circle Area: 78.5
		Rectangle Area: 24

	double width;
	public:
	Rectangle(double l, double w) {
	length = l;
	width = w;
	}
	void displayArea() {
	double area = length * width;
	cout << "Rectangle Area: " << area << endl;
	}
	};
	
	int main() {
	Shape* shapes[3];
	shapes[0] = new Shape();
	shapes[1] = new Circle(5.0);
	shapes[2] = new Rectangle(4.0, 6.0);
	
	for (int i = 0; i < 3; i++) {
	shapes[i]->displayArea();
	}
	
	return 0;
	}	

11.	class Animal {
	public:
	virtual void makeSound() const = 0;
	};
	
	class Dog : public Animal {
	public:
	void makeSound() const {
	cout << "Woof! Woof!" << endl;
	}
	};
	
	class Cat : public Animal {
	public:
	void makeSound() const {
	cout << "Meow! Meow!" << endl;
	}
	};
	
	class Lion : public Cat {
	public:
	void makeSound() const {
	cout << "Roar!" << endl;
	}
		Compilation error: cannot instantiate an abstract class

	};
	
	int main() {
	  Animal* animals[4];
	  animals[0] = new Dog();
	  animals[1] = new Cat();
	  animals[2] = new Lion();
	  animals[3] = new Animal();
	
	  for (int i = 0; i < 4; i++) {
	  animals[i]->makeSound();
	  }
	
	  return 0;
	}	

12	class Animal {
	public:
	  virtual void makeSound() const = 0;
	};
	
	class Dog : public Animal {
	public:
	  void makeSound() const {
	  cout << "Woof! Woof!" << endl;
	  }
	};
	
	class Cat : public Animal {
	public:
	  void makeSound() const {
	  cout << "Meow! Meow!" << endl;
	  }
	};
	
	class Lion : public Cat {
	public:
	  void makeSound() const {
	  cout << "Roar!" << endl;
	  }
	};
	
	int main() {
	  Animal* animals[3];
	  animals[0] = new Dog();
	  animals[1] = new Cat();
	  animals[2] = new Lion();
	
	  for (int i = 0; i < 3; i++) {
		Woof! Woof!
		Meow! Meow!
		Roar!

	animals[i]->makeSound();
	}
	
	return 0;
	}	

13	lass Base {
	public:
	void print() {
	cout << "Base Class" << endl;
	}
	
	void print(int x) {
	cout << "Base Class: " << x << endl;
	}
	
	virtual void display() {
	cout << "Base Display" << endl;
	}
	};
	
	class Derived : public Base {
	public:
	void print() {
	cout << "Derived Class" << endl;
	}
	
	void print(int x, int y) {
	cout << "Derived Class: " << x << ", " << y <<
	endl;
	}
	
	void display() {
	cout << "Derived Display" << endl;
	}
	};
	
	int main() {
	Base* basePtr;
	Derived derivedObj;
	
	basePtr = &derivedObj;
	
	basePtr->print();
	basePtr->print(5);
	basePtr->print(3, 7);
	basePtr->display();
	
	return 0;
		Base Class
		Base Class: 5
		Base Class: 3, 7
		Derived Display

	}	

14	class Shape {
	public:
	  void display() {
	  cout << "Shape" << endl;
	  }
	
	  virtual void print() {
	  cout << "Shape Print" << endl;
	  }
	
	  void print(int x) {
	  cout << "Shape Print: " << x << endl;
	  }
	};
	
	class Circle : public Shape {
	public:
	  void display() {
	  cout << "Circle" << endl;
	  }
	
	  void print() {
	  cout << "Circle Print" << endl;
	  }
	
	  void print(int x, int y) {
	  cout << "Circle Print: " << x << ", " << y <<
	endl;
	  }
	};
	
	class Square : public Shape {
	public:
	  void display() {
	  cout << "Square" << endl;
	  }
	
	  void print() {
	  cout << "Square Print" << endl;
	  }
	
	  void print(double x) {
	  cout << "Square Print: " << x << endl;
	  }
	};
	
	int main() {
		Circle
		Circle Print
		Circle Print: 5, 10
		Square
		Square Print
		Square Print: 3.14

	Shape* shapePtr;
	Circle circleObj;
	Square squareObj;
	
	shapePtr = &circleObj;
	shapePtr->display();
	shapePtr->print();
	shapePtr->print(5, 10);
	
	shapePtr = &squareObj;
	shapePtr->display();
	shapePtr->print();
	shapePtr->print(3.14);
	
	return 0;
	}	

15	class A {
	public:
	virtual void print() {
	cout << "A" << endl;
	}
	
	void print(int x) {
	cout << "A: " << x << endl;
	}
	};
	
	class B : public A {
	public:
	void print() {
	cout << "B" << endl;
	}
	
	void print(double x) {
	cout << "B: " << x << endl;
	}
	};
	
	class C : public B {
	public:
	void print() {
	cout << "C" << endl;
	}
	
	void print(string s) {
	cout << "C: " << s << endl;
		B
		A: 5
		C
		B: 3.14
		C: Hello

	}
	};
	
	int main() {
	A* aPtr;
	B bObj;
	C cObj;
	
	aPtr = &bObj;
	aPtr->print();
	aPtr->print(5);
	
	aPtr = &cObj;
	aPtr->print();
	aPtr->print(3.14);
	aPtr->print("Hello");
	
	return 0;
	}	

Question 2: [20 Marks]

Fill in the boxes with True or False.

1	The assignment operator (**=**) cannot be overloaded in C++.	False
2	Operator overloading can change the precedence and associativity of operators in C++.	False
3	In C++, the derived class can access the protected members of the base class.	False
4	Operator overloading allows us to redefine the behavior of existing operators in C++.	True
5	Friend classes and functions can access private members of other classes.	True
6	A virtual class is a class that can be instantiated and used directly.	False
7	Pure virtual functions have no implementation in the base class and must be overridden by derived classes.	True
8	Method overloading allows a class to have multiple functions with the same name but different parameters.	True
9	Method overriding allows a derived class to provide a different implementation of a function that is already defined in the base class.	True
10	A class can be both the base class and derived class in C++.	True
11	A private member of a class can be accessed by a friend function.	True
12	Inheritance is a way to achieve code reusability in object-oriented programming.	True
13	Virtual functions are resolved at compile-time based on the type of the pointer.	False

14	Multiple inheritance can lead to the diamond problem where ambiguity arises due to two base classes having a common base class.	True
15	A pure virtual function can have a definition in the base class.	False
16	Friend functions are members of a class and can access private members of other classes.	False
17	A derived class can call the constructor of its base class explicitly.	True
18	Overriding a function in a derived class requires the virtual keyword in the base class.	True
19	In method overloading, functions must have the same name and the same number of parameters.	False
20	Virtual functions cannot be defined in the base class and overridden in the derived class.	False