ಭಾರತೀಯ ಮಾಹಿತಿ ತಂತ್ರಜ್ಞಾನ ಸಂಸ್ಥೆ ರಾಯಚೂರು
भारतीय सूचना प्रौद्योगिकी संस्थान रायचूर
Indian Institute of Information Technology Raichur

Course Title: Introduction to Programming	**Class**: CSB24
Instructor: Dr. Ramesh K. Jallu	**Semester**: 1
Course Code:	**Date**: March 23, 2025
Exam: Practice Problems Set-1	**email:** jallu@iiitr.ac.in

1 Implement the following problems to the best of your understanding

1. Create a `Bank_Account` class with attributes `Account_number` and `Balance`, and two methods `Deposit()` and `Withdraw()`. Create two derived classes, `Savings_Account` and `Current_Account`. For `Savings_Account`, add an attribute `Interest_Rate` and a method `Add_Interest()` to calculate and add interest to the balance. For `Current_Account`, add an attribute `Overdraft_Limit` and overload the `Withdraw()` method to allow withdrawing more than the available balance up to the `Overdraft_Limit`, displaying an appropriate message if the limit is exceeded. Demonstrate the functionality by creating objects of both derived classes and performing deposits, withdrawals, and interest calculations.

2. Create a `Book` class with attributes like `Title`, `Author`, and `ISBN`. Derive two classes, `Text_Book` and `Novel`, from the `Book` class. Add additional attributes like `Subject` for `Text_Book` and `Genre` for `Novel`. Overload the `Display()` method in both derived classes to show specific details. Implement a `Library` class to manage a collection of books, allowing users to add, search, and display books based on their type.

3. Create a `Employee` class with attributes like `Name`, `Employee_ID`, and `Salary`. Derive two classes, `Full_Time_Employee` and `Part_Time_Employee`, from the `Employee` class. Add additional attributes like `Bonus` for `Full_Time_Employee` and `Hours_Worked` for `Part_Time_Employee`. Overload the `Calculate_Salary()` method to compute the total salary differently for full-time and part-time employees. Implement a `Payroll` class to manage employee records and calculate monthly payroll.

4. Create a `Product` class with attributes like `Product_ID`, `Name`, and `Price`. Derive two classes, `Electronics` and `Clothing`, from the `Product` class. Add additional attributes like `Warranty_Period` for `Electronics` and `Size` for `Clothing`. Overload the `Display_Details()` method to show specific details for each product type. Implement a `Catalog` class to manage products, allowing users to add, search, and filter products by category.

5. Create a `Vehicle` class with attributes like `Vehicle_ID`, `Model`, and `Rental_Price`. Derive two classes, `Car` and `Bike`, from the `Vehicle` class. Add additional attributes like `Num_Seats` for `Car` and `Engine_Capacity` for `Bike`. Overload the `Calculate_Rent()` method to compute the rental cost differently based on the duration and type of vehicle. Implement a `Rental_System` class to manage vehicles and process rentals.

6. Create a `Patient` class with attributes like `Patient_ID`, `Name`, and `Age`. Derive two classes, `In_Patient` and `Out_Patient`, from the `Patient` class. Add additional attributes like `Room_Number` for `In_Patient` and `Appointment_Date` for `Out_Patient`. Overload the `Display_Details()` method to show specific details for each patient type. Implement a `Hospital` class to manage patient records and generate reports.

7. Create a `Food_Item` class with attributes like `Item_ID`, `Name`, and `Price`. Derive two classes, `Veg_Item` and `Non_Veg_Item`, from the `Food_Item` class. Add additional attributes like `Calories` for `Veg_Item` and `Protein_Content` for `Non_Veg_Item`. Overload the `Display_Details()` method to show specific details for each food type. Implement a `Restaurant` class to manage the menu and process orders.

8. Create a `Student` class with attributes like `Student_ID`, `Name`, and `Grade`. Derive two classes, `High_School_Student` and `College_Student`, from the `Student` class. Add additional attributes like `SAT_Score` for `High_School_Student` and `CGPA` for `College_Student`. Overload the `Calculate_Grade()` method to compute the grade differently for high school and college students. Implement a `School` class to manage student records and generate grade reports.

9. Create a `Product` class with attributes like `Product_ID`, `Name`, and `Quantity`. Derive two classes, `Perishable_Product` and `Non_Perishable_Product`, from the `Product` class. Add additional attributes like `Expiry_Date` for `Perishable_Product` and `Shelf_Life` for `Non_Perishable_Product`. Overload the `Check_Stock()` method to display alerts for perishable products nearing expiry. Implement an `Inventory` class to manage products and generate stock reports.

10. Create a `Flight` class with attributes like `Flight_Number`, `Destination`, and `Seats_Available`. Derive two classes, `Domestic_Flight` and `International_Flight`, from the `Flight` class. Add additional attributes like `Baggage_Allowance` for `Domestic_Flight` and `Visa_Required` for `International_Flight`. Overload the `Book_Ticket()` method to handle booking rules differently for domestic and international flights. Implement a `Booking_System` class to manage flights and process bookings.

2 Find out the error(s) in the following snippets based on function overloading: It is a very useful concept but it can lead to errors if it is not handled properly.

1.
```
#include<iostream>
using namespace std;

class Base
{
protected:
    int value;
public:
    Base()
    {
        value = 100;
    }
    void show()
    {
        cout << "Value: " << value << endl;
    }
};

class Derived : private Base
{
public:
    void print()
    {
        cout << "Value in Derived: " << value << endl;
    }
};

int main()
{
    Derived obj;
    obj.print();
    obj.show();  // Will this compile?
    return 0;
}
```

2. How many times is A's constructor called? What happens if `virtual` is removed?

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

```
class A
{
public:
    A()
    {
        cout << "A's constructor" << endl;
    }
};

class B : virtual public A
{
public:
    B()
    {
        cout << "B's constructor" << endl;
    }
};

class C : virtual public A
{
public:
    C()
    {
        cout << "C's constructor" << endl;
    }
};

class D : public B, public C
{
public:
    D()
    {
        cout << "D's constructor" << endl;
    }
};

int main()
{
    D obj;
    return 0;
}
```

3. The following code has an error though it looks like everything seem in order.

```
#include<iostream>
```

```
using namespace std;

class Base
{
public:
    void show(int x)
    {
        cout << "Base: " << x << endl;
    }
};

class Derived : public Base
{
public:
    void show()
    {
        cout << "Derived's show()" << endl;
    }
};

int main()
{
    Derived obj;
    obj.show(10);  // What happens here?
    return 0;
}
```

When a derived class defines a function with the same name as a function in the base class, all overloaded versions of that function in the base class become hidden in the derived class. This scenario is called **function hiding**. Even if the parameter list differs, the base class function is hidden, unless explicitly brought back with `using`. To allow access to the base class function, use `using BaseClass::functionName;` in the derived class.

Fix:

Option 1: Explicitly Bring `show(int)` into `B`'s Scope

Use `using Base::show;` inside `Derived` to bring the base class function into scope:

```
class Derived : public Base
{
public:
    using Base::show;  // Bring Base's show(int) into Derived's scope
    void show()
```

```
    {
        cout << "Derived's␣show()" << endl;
    }
};
```

Option 2: Override the Function Correctly

If the intent was to override `show(int)`, ensure `Derived`'s `show()` takes an integer:

```
class Derived : public Base
{
public:
    void show(int x)
    {
        cout << "Derived's␣show()" << x << endl;
        // Overriding Base's show(int)
    }
};
```

4.
```
#include<iostream>
using namespace std;

int add(int a, int b)
{
    return a + b;
}

double add(int a, int b)
{
    return a + b;
}

int main()
{
    cout << add(5, 10) << endl;
    return 0;
}
```

5.
```
#include<iostream>
using namespace std;

class Animal
{
public:
```

```
    void eat()
    {
        cout << "Animal eats" << endl;
    }
};

class Dog : public Animal
{
public:
    void eat()
    {
        cout << "Dog eats bones" << endl;
    }
};

int main()
{
    Animal* a = new Dog;
    a.eat();
    delete a;
    return 0;
}
```

6.
```
#include<iostream>
using namespace std;

void print(int a, int b = 10)
{
    cout << "a: " << a << ", b: " << b << endl;
}

void print(int a) // Error here
{
    cout << "a: " << a << endl;
}

int main()
{
    print(5);
    return 0;
}
```

Error Explanation

The error occurs because the two `print` functions are ambiguous when called with a single argument:

- The first function, `print(int a, int b = 10)`, can be called with one argument (since `b` has a default value).
- The second function, `print(int a)`, also takes a single argument.

This ambiguity results in a **compilation error**.

How to Fix the Error

To resolve the ambiguity, you can:

1. Remove one of the overloaded functions.
2. Change the parameters of one of the functions to make them distinct.

7. Identify the error related to the `const` keyword in function overloading and fix it.

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

void print(int a)
{
    cout << "Non-const: " << a << endl;
}

void print(const int a) // Error here
{
    cout << "Const: " << a << endl;
}

int main()
{
    int x = 10;
    print(x);
    return 0;
}
```

Error explanation

The error occurs because the two `print` functions are not distinguishable by the compiler. The `const` qualifier for pass-by-value parameters does not affect the function's

signature. As a result, the compiler cannot differentiate between the two functions, leading to a compilation error.

8. Identify the destructor-related error in inheritance and fix it.

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

class Base
{
public:
    ~Base()
    {
        cout << "Base Destructor" << endl;
    }
};

class Derived : public Base
{
public:
    ~Derived()
    {
        cout << "Derived Destructor" << endl;
    }
};

int main()
{
    Base* b = new Derived();
    //new Derived; also works, but there is slight difference
    delete b; // Error here
    return 0;
}
```

Error Explanation

The error occurs because the destructor of the `Base` class is not declared as `virtual`. When you delete an object of the `Derived` class through a pointer to the `Base` class:

- Only the `Base` class destructor is called.
- The `Derived` class destructor is not called, leading to **undefined behavior** (e.g., memory leaks if the `Derived` class allocates resources).

This happens because the destructor of the `Base` class is `non-virtual`, so the compiler does not know to call the `Derived` class destructor.

To fix the error, declare the destructor of the `Base` class as `virtual`. This ensures that the destructor of the `Derived` class is also called when deleting through a `Base` pointer. Always declare the destructor of a `base` class as `virtual` if you intend to delete derived objects through a `base` class pointer.

9. Identify the error related to reference parameters in function overloading and fix it.

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

void print(int& a)
{
    cout << "Reference: " << a << endl;
}

void print(int a) // Error here
{
    cout << "Value: " << a << endl;
}

int main()
{
    int x = 10;
    print(x);
    return 0;
}
```

Error Explanation

The error in the given code occurs due to function overloading ambiguity. When `print(x);` is called in `main()`, the compiler finds two matching overloads: `print(int&)` (pass-by-reference) and `print(int)` (pass-by-value). Since `x` is an `int` value, it can be bound to both functions equally well, leading to a compilation error due to ambiguity. This happens because references and values with the same type do not provide enough distinction for overload resolution. To resolve this, we can either change one function's parameter type (e.g., `double` instead of `int`), use a pointer for differentiation (`int*` instead of `int&`), or remove one of the conflicting overloads to avoid ambiguity.