B-Tech 4th Semester
End-Term Examination

abcdefghij

MAY 2021

Subject: Programming Paradigm [CS-2203]

Date: 24th May 2021
Name: Abhiroop Mukherjee
Exam Roll No.: 510519109
G-Suite ID: 510519109.abhiroop@students.iiests.ac.in
No. of Sheets Uploaded: 10

---

2) a) Consider a C++ Function

```
int fn1 (int param1, int param2)
{
  ⋮
}
```

→ given only this much information, we can't confirm that output of the function will be same everytime for same values of param 1 and param 2

→ This is because the fn1() can use global variable, randomness, etc.

→ This uncertainity is not present in functional programming, and this lack of uncertainity is called Referential Transperancy

→ Definition: A Function's Result depends only upon the values of it's argument and not any previous computation or order of evaluation of it's argument

→ This makes sure that ~~if we give same~~ for a set of paremeter values (param 1, param 2), output of function will always be same

→ This Property bring following advantages:

i) Memoization
ii) Modulaization
iii) Ease of Debugging
iv) Concurrence
v) Parallelization
vi) Idempotence

b) Given $F(m,n) = n+1$ if $m=0$

$= F(m-1, 1)$ if $m>0, n=0$

$= F(m-1, A(m,n-1))$ if $m>0, n>0$

→ Assuming $A(m,n)$ is already defined previously

→ In Scheme LISP Code:

```
(define (F m n)
    (cond ((= m 0) (+ n 1))
          ((and (> m 0) (= n 0)) (F (- m 1) 1))
          ((and (> m 0) (> n 0)) (F (- m 1)
                                    (A m (- n 1))))
    )
)
```

d) I) State of an Object

→ State of an object is one of the possible condition the state may have, due to the values of the objects attribute

Eg consider a cup [object is cup]

→ cup can be half full
→ cup can be 25% full
→ cup can be 68% full

→ all these are one of possible state of the object cup

II) Behavior of an object

→ Behavior determines how an object should act/react to request from other object, and are often represented by functions/methods in of the objects class

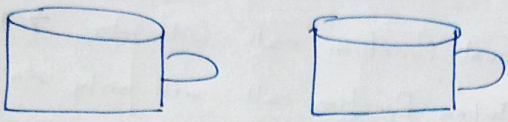
Eg: Consider a cup as an object

→ It's Behavior is to hold liquid

III) Identity

→ This means that objects can be distinguished by their invariant existence, and not by their state or ~~~~ looks or description

Eg consider two cups, ~~with~~ with same shape, color, state, etc, i.e identical object



→ Their invariant existence creates their identity.

~~Eg: we can de~~

e) "Inline Function speeds up execution and decreses the executable code size".

→ Inline Function is a request to compiler ~~to~~ that ~~the~~ inline ~~code~~ substitution should be preferred rather than usual call implementation.

Eg consider a function

```
void f (int a, int b)
{
}
```

and function call f(c, d)

→ following hidden steps take place during function call

a = c
b = d

→ using inline [and considering the request is accepted], the two hidden steps are skipped, and function body, ^ processing is directly placed in place of function call

~~Eg 2: now consider a local~~

→ So making inline skips parameter copy [if used] and also skips function ressolution. so first part of the given statement "speeds up execution is true".

Eg 2 consider a ~~size of~~ fn. with body size 10 lines and no. of steps to call fn. be 5 lines

→ So executable size will increase with each function call, if function is made inline

Eg 3: consider fn. with body size 2 lines and no. of steps to call function be 5 lines

→ So usual function call will do 7 lines of execution

→ But Inline function call will only do 2 lines of execution

→ So we see that using inline can also decrease executable size.

∴ The given statement is False.

h) Consider Following Example

```
class Circle {
    float radius;
public:
    float area();
    float ret_rad();
};

class Cylinder : public Circle {
    float height;
public:
    float area();
    float Volume();
};
```

→ now if we do

```
Circle *oB = new Cylinder;
oB -> area();
```

→ This will execute area() of Circle class, purely because oB is of type Circle

→ This happens because function calls are resolved at ~~compile time~~ compile time.

→ now consider we wanted to use area() based on the type of object it is, i.e in ~~this case~~ previous example as we assigned the pointer a Cylinder object, we want to execute area() of Cylinder.

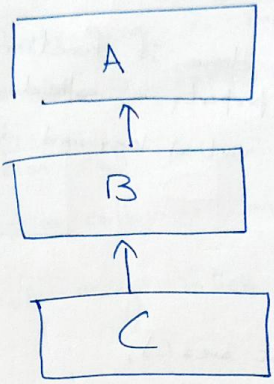
→ This can only be done if function resolution happens at runtime. and this property is called dynamic polymorphism

→ This is done using virtual keyword in C++

Eg

```
class Circle {
    float radius;
public:
    virtual float area();
    float ret_rad();
};
class Cylinder : public Circle {
    float height;
public:
    float area();
    float Volume();
};
int main() {
    Circle *oB = new Cylinder;
    oB -> area(); // this will now invoke Cylinder class area().
}
```

i) Multi Level Inheritance is a type of Inheritance where one object inherits from another object, which in turn inheritance from another different object, ~~it's UML diag~~ it will have this type of diagram:



Eg
```
class A {
  public:
    void fn() {
      cout << "In A" << endl;
    }
  void fn2(){ cout<<"In A 2" <<endl; }
};
class B : protected A {
  public:
    void fn() {
      cout << "In B" << endl;
    }
};
class C: public B {
  public:
    void fn() {
      cout << "In C" << endl;
    }
};
```

→ now if we do

```
int main(){
    C oC;    //executes fn
    oC.fn();  //executes fn of C
    oC.B::fn(); //executes fn of B
    //oC.A::fn(); // will give error
    oC.A::fn2(); // executes fn2 of A
}
```

→ class A is not direct base class of C, hence we can't do [same name]

```
oC.A::fn();
```

→ but other functions can be called

```
oC.A::fn2();
```

j) → Use case contains different ways a system can be used by the viewers

→ This is very diffevent from all other UML views as it focus only on interface and not in internal workings of a system

→ All other views must confirm to this view.

→ Use Case view is used to define "what" we need, not about "how" we do it.

→ Making a Use case view helps us to determine the Requirements or Specifications we need to cater to our cause.

Eg ~~Zomato~~

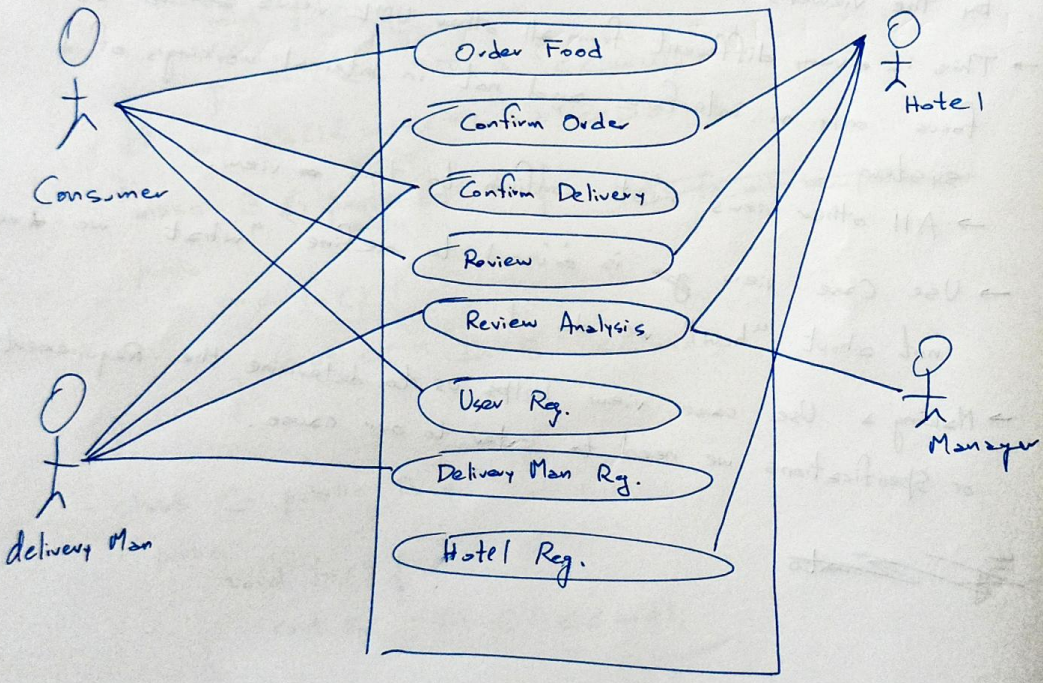
Eg: Zomato

Types of Users:

i) Consumer
ii) Delivery Man
iii) Hotel
iv) Zomato Manager

Functionality

i) User order food ; Hotel takes input
ii) ~~Hotel take~~ Hotel confirms food made, delivery man picks up
iii) Delivery man delivers, consumer takes
iv) Consumer Review of Food & Delivery & Hotel
v) ~~Kick~~ Demote / Promote Delivery Man based on Review
vi) User Registration
vii) Delivery Man Registration
viii) Hotel Registration.



Q2)

```
class ThermostaticHeater {
    unsigned int tempLimit; // temp from 0 to 30 celcius
    bool heaterState;       // on or off (on = true)

public:
    void evaluateState();          //switch heater on/off
    int getCurrentTemp();          // returns temp from sensor
    bool getCurrentState();        // report on or off (on = true)
    unsigned int getTempLimit();   // returns tempLimit
    void changeTempLimit();        // changes tempLimit to user
                                   // defined value
    ThermostaticHeater(): tempLimit(25), heaterState(false) { }
    ThermostaticHeater(unsigned int t_lim, bool h_sta)
        : tempLimit(t_lim), heaterState(h_sta) { }
};

unsigned int ThermostaticHeater::getTempLimit(){
    return tempLimit;
}

void ThermostaticHeater::changeTempLimit(){
    cout << "Enter new tempLimit: ";
    unsigned int temp;
    cin >> temp;
    if (temp >= 0 && temp <= 30)
        tempLimit = temp;
    else
        cout << "Error: Out of Range Value" << endl;
    evaluateState();
}
```

```
void ThermostaticHeater::evaluateState() {
    int curr_temp = getCurrentTemp();

    if (curr_temp >= tempLimit)
        heaterState = false;
    else
        heaterState = true;
}

int main() {
    ThermostaticHeater h(27, false);
    cout << "Temp Limit : " << h.getTempLimit() << endl;

    h.evaluate();
    h.evaluateState();
    h.changeTempLimit();
    cout << "Temp Limit: " << h.getTempLimit() << endl;
}
```