B-Tech 4TH Semester Mid Term Examination

April 2021

Subject: Programming Paradigm [CS2203]

Date: 19/04/2021
Name: Abhiroop Mukherjee
Enrolment No.: 510519109
G-Suite ID: 510519109.abhirup@students.iiests.ac.in
No. of Sheets Uploaded: 8

Q1) b) → Initialization List is a way to initialize ~~values~~ values of a class

→ They can only be used in constructor and does it's work during compile time, removing ~~any~~ that work during execution.

~~→ This~~

→ It is the only way to initialized constant or reference data members of class.

Eg class ~~Pe~~ person {

```
    char * name;
    int age;
    const int death_time;
public:
    person ();
};
```

```
person :: person ( ) : name (NULL), age (-1), death-time (202)
{ }

int main ()
{
  :
}
```

→ The order of invoking of initialization list is not in the order of occourence, but in order with the declarations in class.

c) → Call by reference is a way to invoke functions, using ~~val~~ references of actual parameter, rather than using values of actual parameter.

Eg
```
void swap (int &a, int &b) {
    int c = a;
    a = b;
    b = c;
}
```

→ ~~This is~~ By this way we achive following things:

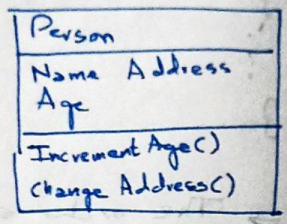
→ No copy of actual parameters: save spaces in stack

→ Avoid Shallow Copy Problems

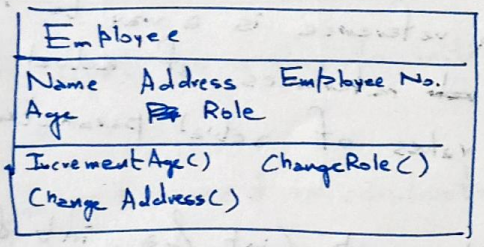
→ ~~Can~~ Editing of actual parameters possible [can also be stopped using const]

e) Inheritance is a concept by which we make classes upon existing classes, by which additional data can be added to the new class.

Eg consider ~~class~~ object Person →



Eg consider another ~~class~~ object Employee



→ we can make employee ~~a~~ class by following ways

i) ~~Build~~ Build from ~~Scratch~~. Scratch

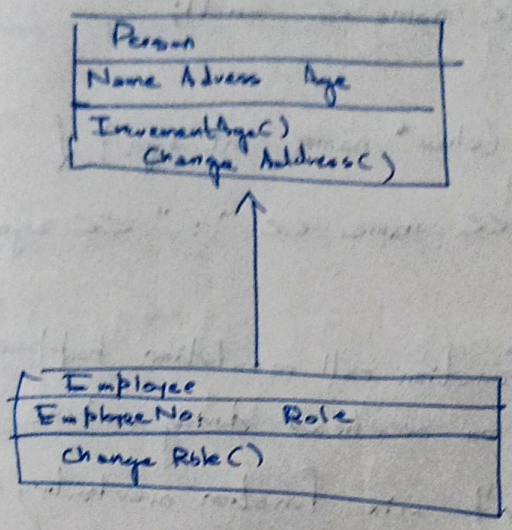
→ Code Reusability violated.

→ Code for IncrementAge() and ChangeAddress() already present

ii) Modify Person

→ Abstraction gets violated.

iii) Use Inheritance



→ ~~the~~ as "Employee is a Person"

→ Employee class can take all the stuffs of Person and add more stuffs ~~to~~ which are Employee specific.

f) → Function Overloading is a way by which we "overload" a function by ~~to~~ having ~~same name~~ multiple functions by same name, but with different parameters.

→ This is also called static ~~overloading~~ polymorphism as

i) Static as it happens during compile time and not runtime

ii) Polymorphism, as the function changes it action based on context [same name, multiple functionality]

Eg
```
void print (char* name ){
    cout << name << endl;
}
void print (char* name , int age) {
    cout << name << " : " << age << endl;
}
```

→ As ~~this~~ function call resolution happens in compile time, there is no overhead during runtime

→ We can't mix function overloading with default parameter

→ Resolution is based on number, type and order of ~~arg~~ arguments only, the return type doesn't play any role.

(12)

```
class JobLoad {
    unsigned int work_time;
        float salary_rate;

        JobLoad ();

public:
        JobLoad (float, int = 40);
    void addHrs (int);
    void print Salary ();
    ~~void Reset ();~~ friend void Reset (JobLoad &);
    bool subtract Hrs (int);
    int  compre Sal (JobLoad &);
        ~~JobLoad (const JobLoad &); JobLoad &~~
      JobLoad & operator = (const &JobLoad)
};
void Reset (JobLoad & obj) {
    obj. work_time = 0;
}

JobLoad:: JobLoad (): work_time (0), salary_rate (0) { }

JobLoad:: JobLoad (float rate, int work = 40)
    : work_time (work), salary_rate (rate) { }

void JobLoad:: addHrs (int increse_size) {
    work_time += increse_size;
}

void JobLoad:: print Salary () {
    cout << ~~work_time (int)~~ (work_time * salary_rate)
        << endl;
}
```

```
bool JobLoad::subtractHrs(int hour){
    if(work_time < hour)
        return false;
    work_time -= hour;
    return true;
}

int JobLoad::compareSal(const JobLoad &obj){
    float thisSal = work_time * salary_rate;
    float objSal = obj.work_time * obj.salary_rate;

    if(thisSal == objSal) //avoiding float errors
        return 0;
    else if(thisSal > objSal)
        return 1;
    else
        return -1;
}

~~JobLoad::JobLoad(const JobLoad &obj){~~
    ~~work_time = obj.work_time;~~
    ~~salary_rate = obj.salary_rate;~~
~~} //this is not needed, default copy constructor~~
~~//was enough but still...~~

JobLoad & JobLoad::operator=(const JobLoad &rhs){
    work_time = ~~obj~~rhs.work_time;
    salary_rate = rhs.salary_rate;
    return *this;
} //default was enough, but still...
```

```
int main () {

    JobLoad jl1 (25, 55);
    jl1.addHrs (5);
    jl1.printSalary ();

    Reset (jl1);

    bool okay = jl1.subtractHrs (10);

    JobLoad jl2 (200);
    int r= jl2.compreSal (jl1);

    jl1= jl2;

}
```