BLG252E-OBJECT ORIENTED PROGRAMMING

SECOND PRACTICE SESSION

Inheritance

While using different type of inheritance, following rules are applied:

•**Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and**protected** members of the base class.

•**Protected Inheritance:** When deriving from a **protected** base class, **public** and **protected**members of the base class become **protected** members of the derived class.

•**Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Question #1

Taken from Midterm #2 (2012 Spring)

A bank has 3 types of customers defined as:

1. a regular customer (**Customer**)
2. a VIP customer (**VIPCustomer**)
3. its own employees (**Employee**).

The bank applies different policies to its customers. When a customer comes into a bank office, he/she takes a queue number and waits to be served. The VIPCustomers have priority to regular customers and they are always served before them. The Employees never wait for other customers and they are directly served (unless there are other Employees in the queue). The Employees may be seen as a more featured VIPCustomer type.

Suppose that there is only one bank cashier in the office and design your classes according to the following constraints:

- Each customer should have its own queue number (**qn**) and an initial amount of money (**balance**) in their account.
- The balance information of an Employee could never be examined by anyone.
- The queue numbers will start from a different value (**count**) for each customer type and will be incremented by one for each new coming customer. For example: The regular customers will get qn values like 1,2,3,4… The VIPCustomers' queue numbers will start from 100 and continue like 101,102… The Employees' queue numbers will start from 200. Keep in mind that a new coming Employee will also increment the count for regular customers as well as VIPcustomers.
- A customer queue should be created in order to process the customers coming to the bank office. This queue may have multiple queues in it if needed. (For example you may create 3 different linkedlists for different customer types and process these lists in order.)

To Do

- Design these classes in C++ while **avoiding code repetition** as much as possible for all classes and providing **data hiding**.
- Test code is given to guide the design of your classes.
- All the dynamic data members should be declared as **private**. “friend” class declaration is not allowed.
- Write **all declarations and the bodies** of the required methods for the classes.
- (OPTIONAL!) You may use the linked list of the STL in your implementation:

(Ex: list<Customer>, some methods that you may need are: bool empty(), pop_front(), push_back(Customer&), front() returns a reference to the first element in the list.

Sample Test Code

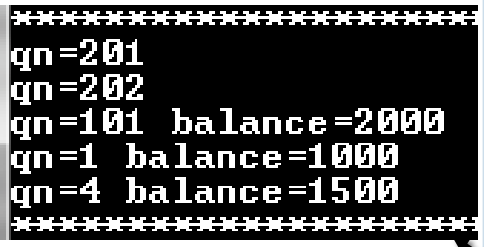
```
Customer c1(1000); VIPCustomer c2(2000); Employee c3(3000);
Customer c4(1500); Employee c5(5000);
    //c1's qn will be 1, c2->101, c3->201, c4->4, c5->202
cout<<"Total number of Customers="<< Customer::count<<endl; // 5
cout<<"Total number of VIPCustomers="<< VIPCustomer::count-100<<endl;//3
cout<<"Total number of Employees="<< Employee::count-200<<endl;//2



c1.getbalance(); c1.getqn(); c3.show(); c3.getqn();//ok
//c3.getbalance();//error at this line not possible

Customerqueue cq;
cq.add(c1);          cq.add(c2);          cq.add(c3);          cq.add(c4);          cq.add(c5);

cq.print();//will print all the customers in queue according to their priority
while(!cq.empty()){
  Customer *ref=cq.readnextCustomer();
  ref->show(); //will print only the qn values for employees, qn and balance for the others
  cq.popnextCustomer();
}
```



A sample code for using STL linked list:

```
list<int> mylist;
mylist.push_back(1); //the list is [1]
mylist.push_back(2); //the list is [1 2]
mylist.push_back(3);//the list is [1 2 3]
cout<<mylist.front()<<endl;//1 is printed to the screen
mylist.pop_front();//1 is removed, the list is [2 3]
//the loop iterates over the elements of the list
// 2 and 3 are printed to the screen
for(list<int>::iterator it=mylist.begin();it!=mylist.end();it++){
        cout<<*it<<endl;
}
while(mylist.empty()!=false)
        mylist.pop_front(); //this will empty the whole list
```

Customerqueue Class

```
class Customerqueue{
        list<Customer> l1;
        list<VIPCustomer>l2;
        list<Employee>l3;
public:
        void add(Customer &c);
        void add(VIPCustomer &c);
        void add(Employee &c);
        void print();
        bool empty()const {return l1.empty()&&l2.empty()&&l3.empty();};
        ...       readnextCustomer();
        ... popnextCustomer();
};
```

```
void Customerqueue::add(const Customer &c){
       l1.push_back(c);}
void Customerqueue::add(const VIPCustomer &c){
       l2.push_back(c);}
void Customerqueue::add(const Employee &c){
       l3.push_back(c);}
void Customerqueue::print(){
       for(list<Employee>::iterator it=l3.begin();it!=l3.end();it++){
              it->show();
       }
    for(list<VIPCustomer>::iterator it=l2.begin();it!=l2.end();it++){
              it->show();
       }
       for(list<Customer>::iterator it=l1.begin();it!=l1.end();it++){
              it->show();
       }
}
...//please write the code of the remaining methods for Customerqueue
```

SOLUTION

```
class Customer{
private:
int qn;              //queue number
int balance;        // initial amount of money in their account

public:
static int count;
Customer(int amount):balance(amount){  //customer constructor
count++;
setqn(count);
};

Customer(const Customer &c){ //copy constructor
qn=c.qn;
balance=c.balance;
//cout<<"customer copy constructor="<<qn<<","<<balance<<endl;
};

~Customer(){ //customer destructor
}
```

```
void setqn(int x){qn=x;};
int getqn(){return qn;};
int getbalance(){return balance;};
virtual void show(){cout<<"qn="<<qn<<" balance="<<balance<<endl;};
};
int Customer::count=0;

class VIPCustomer:public Customer{
public:
static int count;
VIPCustomer(int amount):Customer(amount){  //VIPCustomer constructor
count++;
setqn(count);
};
~VIPCustomer(){                            // VIPCustomer destructor
};

};
```

```
int VIPCustomer::count=100;

class Employee:public VIPCustomer{                    // Employee derived from VIPCustomer
count++;

private:
int getbalance(){return Customer::getbalance();};

public:
static int count;
Employee(int amount):VIPCustomer(amount){      //Employee constructor
count++;
setqn(count);
};
~Employee(){                                  //Employee destructor
};
void show(){cout<<"qn="<<getqn()<<endl;};
};
int Employee::count=200;
```

```
class Customerqueue{
list<Customer> l1;
list<VIPCustomer>l2;
list<Employee>l3;
public:
void add(Customer &c);
void add(VIPCustomer &c);
void add(Employee &c);
void print()  ;
bool empty()const {return l1.empty()&&l2.empty()&&l3.empty();};
Customer* readnextCustomer();
void popnextCustomer();
};
void Customerqueue::popnextCustomer(){
if(!l3.empty()){
l3.pop_front();
}
else if(!l2.empty()){
l2.pop_front();
}
else if(!l1.empty()){
l1.pop_front();   }}
```

```
Customer *Customerqueue::readnextCustomer(){
if(!l3.empty()){
return &l3.front();
}
else if(!l2.empty()){
return &l2.front();
}
else if(!l1.empty()){
return &l1.front();
}
return 0;
}
void Customerqueue::add(Customer &c){
l1.push_back(c);
}
void Customerqueue::add(VIPCustomer &c){
l2.push_back(c);
}
void Customerqueue::add(Employee &c){
l3.push_back(c);
}
```

```
void Customerqueue::print() {
cout<<"***************************"<<endl;
for(list<Employee>::iterator it=l3.begin();it!=l3.end();it++){
it->show();
}
for(list<VIPCustomer>::iterator it=l2.begin();it!=l2.end();it++){
it->show();
}
for(list<Customer>::iterator it=l1.begin();it!=l1.end();it++){
it->show();
}
cout<<"***************************"<<endl;
}
```

Question #2

Taken from Midterm #2 (2012 Spring)

Many statements in the following program cause **compile-time errors**.

For the declarations and definitions of classes: please rewrite the methods (with errors) without changing the parameters, **comment out** (put // in front of the statements) incorrect assignments and **give the reasons** (next to the erroneous statements).

For the main function: just **comment out** the erroneous lines. After commenting out the incorrect statements in the main program, write the **outputs** generated by each line indicating the line number

```
class Aclass{
        int x;
        void funcA();
protected:
        int y;
        void protectedFuncA(){}
public:
        Aclass(int new_x):x(new_x){cout<<"Aclass::constructor x = "<< x << endl;}

        void publicFuncA(){cout << "publicFuncA()"<<endl;}

        ~Aclass(){cout<<"Aclass::destructor"<<endl;}
};
```

```
class Bclass:public Aclass{
public:
        Bclass(int new_y){y=new_y};
        void funcB(){
                cout<<"Bclass::funcB"<<endl;
                x=5;
                y=7;
                protectedFuncA();
        };
 Bclass(const Bclass& in_b):Aclass(in_b){cout<<"Bclass::Copy constructor"<< endl;}

        void publicFuncA(int){cout << "publicFuncA(int)"<<endl;}

        ~Bclass(){cout<<"Bclass::destructor y = "<< y <<endl;}

};
```

```
class Cclass{
        int z;
public:
        Cclass(){z=0; cout<<"Cclass::constructor z="<<z<<endl;}
        Cclass(const Cclass& in_c){cout<<"Cclass:: Copy constructor"<<endl;}


        ~Cclass(){cout<<"Cclass::destructor"<<endl;}
        void funcC(){cout<<"Cclass::funcC"<<endl;}

};
```

```
class Dclass:private Cclass{
        Bclass obj_b;
public:
        Dclass(const Bclass &newb){}
        Dclass(const Dclass &newd){}
        ~Dclass(){cout<<"Dclass::destructor"<<endl;}
        void funcB();
};

void Dclass::funcB(){
        cout<<"Dclass::funcB"<<endl;
        Bclass::funcB();
        obj_b.y=5;
}
```

```
int main (){
1:      Bclass obj_b1(3);
2:      Bclass obj_b2(obj_b1);
3:      Bclass obj_b3(8);
4:      obj_b1.funcA();
5:      obj_b1.protectedFuncA();
6:      obj_b1.publicFuncA();
7:      obj_b1.publicFuncA(5);
8:      Dclass obj_d1(obj_b2);
9:      Dclass obj_d2(obj_d1);
10:     obj_d1.funcC();
11:     obj_d1.z = 3;
12:     obj_d1.funcB();
13:     return 0;
}
```

SOLUTION

```
class Bclass:public Aclass{
public:
        //Bclass(int new_y){y=new_y};
Bclass(int new_y):Aclass(new_y){y = new_y;}  //Aclass has no default constructor.
        void funcB(){
                cout<<"Bclass::funcB"<<endl;
                //x=5;        //private data of Aclass may not be reached directly

                y=7;
                protectedFuncA();
        };
 Bclass(const Bclass& in_b):Aclass(in_b){cout<<"Bclass::Copy constructor"<< endl;}


        void publicFuncA(int){cout << "publicFuncA(int)"<<endl;}

        ~Bclass(){cout<<"Bclass::destructor y = "<< y <<endl;}

};
```

```
class Dclass:private Cclass{
        Bclass obj_b;
public:
        //Dclass(const Bclass &newb){}
Dclass(const Bclass &newb):obj_b(newb){}

        //Dclass(const Dclass &newd){}
Dclass(const Dclass &newd):Cclass(newd), obj_b(newd.obj_b){}

        ~Dclass(){cout<<"Dclass::destructor"<<endl;}

        void funcB(); };
void Dclass::funcB(){
        cout<<"Dclass::funcB"<<endl;
        //Bclass::funcB();
obj_b.funcB();

        obj_b.y=5; }
```

```
class Cclass{
        int z;
public:
        Cclass(){z=0; cout<<"Cclass::constructor z="<<z<<endl;}

        Cclass(const Cclass& in_c){cout<<"Cclass:: Copy
constructor"<<endl;}

        ~Cclass(){cout<<"Cclass::destructor"<<endl;}

        void funcC(){cout<<"Cclass::funcC"<<endl;}

};
```

```
int main (){

Bclass obj_b1(3);
Bclass obj_b2(obj_b1)
Bclass obj_b3(8);
//obj_b1.funcA(); // private function of Aclass may not be reached
//obj_b1.protectedfuncA();// protected function of Aclass
//obj_b1.publicfuncA();
obj_b1.publicFuncA(5);
Dclass obj_d1(obj_b2);
Dclass obj_d2(obj_d1);
//obj_d1.funcC();
//obj_d1.z = 3; //the default access for classes is private
obj_d1.funcB();
getchar();
return 0;
}
```

OUTPUT

Aclass::constructor x = 3

Bclass::Copy constructor

Aclass::constructor x = 8

publicFuncA(int)

Cclass::constructor z = 0

Bclass::Copy constructor

Cclass::Copy constructor

Bclass::Copy constructor

Dclass::funcB

Bclass::funcB