Hardik Soni
20CS30023

Name: Hardik Pravin Soni Semester 4
Roll Number: 20CS30023
Computer Science and Engineering
(CS29206/CS20002)

Software Engineering Theory

Assignment-2: Classes and Object-Oriented Programming

```
/*
* Name :- Hardik Pravin Soni
* Roll No :- 20CS30023
* Subject :- Software Engineering Theory (CS2920
*/
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;
class PrimalityTest
{
private:
    static PrimalityTest *_myTest;
    int nStored; // Number of Stored Prim
    int *primes; // Pointer to Buffer
    int bufsize; // Size of the Buffer
    // Constructor of class PrimalityTest
    PrimalityTest(int bufsize)
    {
        this->primes = new int[bufsize];
        this->bufsize = bufsize;
        // The first Prime is 2.
        this->nStored = 1;
```

```
    primes[0] = 2;
}
// Destructor of the Class PrimalityTest.
~PrimalityTest()
{
    delete this->_myTest;
    delete[] this->primes;
}
public:
static PrimalityTest &newTest(int bufsize=100)
{
    // If the _myTest is NULL pointer then, create a new instance.
    if(_myTest == NULL)
        _myTest = new PrimalityTest(bufsize);
    else{
    // If Bufsize is unequal then create a new instance
    if(_myTest->bufsize != bufsize)
    {
    PrimalityTest *t = new PrimalityTest(bufsize);
    for(int i=0; i < min(_myTest->nstored, bufsize)
                    ; i++)
    {
        // Populating a temporary buffer with initial instance's prime.
        t->primes[i] = _myTest->primes[i];
        t->nstored;
    }
    // Deleting the redundant initial buffer
    delete _myTest->primes;
    _myTest = t; }
}
```

```
// Returning the instance acc^n to parameter
   of this call
return *_myTest;
}
void test (int n)
{
// If it is 0 or 1 then neither prime
   nor composite
if (n==0 || n==1)
   cout << n << " is neither prime
               nor composite" << endl;
// If 2 is there, it is prime
else if (n==2)
   cout << n << " is prime." << endl;
else
{ // maxm factor is less than sqrt(n)+1
int limit = sqrt(n) + 1;
// if the number can be checked
   in the given buffer.
if (limit <= primes[nstored-1])
{
// Check in the buffer, if we found
   a factor, it is composite
for(int i=0; i< nstored; i++)
{
if (n % primes[i] == 0)
{
// Factor is Found
cout << n << " is composite." << endl
return;
}
}
```

Hardik Soni
2015300023

```
// If we didn't find a factor then
    it is late
cout << n << " is Prime." << endl;
else
{
  //Vector to store primes as Sieve
    of Eratosthenes
  vector<int> sieve (limit+1, 1);
  for(int p=2; p*p <= limit; p++)
  {
    // If prime[p] is not changed,
    // then it is prime.
    if (sieve[p])
    {
      // Update all multiples
      // of p greater than or
      // equal to the square of it
      // numbers which are multiple
      // of p and are less than p^2
      // are already been marked.
      for(int i = p*p; i <= limit; i+=p)
          sieve[i] = 0;
    }
  }
int flag = 1;
nstored = 0;
for(int i=2; i < sieve.size(); i++)
{
  if (sieve[i])
  {
    if( nstored >= bufsize)
    {
```

```
            flag = 0;
            break;
            // It is found that buffer is
            not sufficient at any point
            then it breaks and error
            messages is printed.
        }
        primes[++nsto nstored++] = i;
    }
    // If buffer memory was sufficient
    if (flag)
    {
        // Checking if the Number is
        Prime / composite
        for(int i = 0; i < nstored; i++)
        {
            if (n % i == 0)
            {
                // Checkin the buffer, if we
                found a factor it is composite
                cout << n << " is composite." << endl;
                return;
            }
        }
        // If we didn't find a factor
        then it is prime
        cout << n << " is Prime" << endl;
    }
    // Buffer Memory is not sufficient
    else
        cout << "ERROR: Buffer Size
                Overflow." << endl;
```

```
    }
  }
}
};
// The single instance of the class
PrimalityTest.
PrimalityTest * PrimalityTest::_myTest = NULL;
int main()
{
  PrimalityTest::newTest().test(2958);
  PrimalityTest::newTest().test(823);
  PrimalityTest::newTest().test(83479);
  return 0;
}
```

Hardik Soni
20CS30023

2.

Libraries

```
#include <iostream>
#include <list>
#include <utility>
#include <vector>
#include <algorithm>
using namespace std;
```

Here, in the above codeblock, I have included all the important/ requisite libraries (namely):

- iostream → provides basic input/ output services.
- list → are sequence containers that allow constant time insert/ erase operations anywhere within the sequence & iteration in both directions.
- utility → template which define default behaviour for the relational operator !=, >, <= and >= b/w objects of same type.
- vector → sequence containers representing arrays that can change the size during runtime.

```
class Customer {
private:
    string name; // Name is declared as string
    size_t id; // Id is declared as unsigned integer.
    static size_t NumCustomers;
                // Number of instances of Customer
                are unsigned int statically declared
public:
```

Hardik Soni
20CS30023

```
Customer (string name = "NA") {
    this→ name = name;
    this→ id = ++NumCustomer;
}
~ Customer () {
    this→ name = "NA";
}
friend ostream &operator << (ostream &os, const
                             Customer &cust){
    cout << "*Customer*" << endl;
    cout << "Name:" << endl;
    os << cust.name << "ID:" << cust.id << endl;
    cout << endl;
    return os;
}
friend istream &operator >> (istream &is, const
                             Customer & cust){
    cout << "*Customer*" << endl;
    cout << "Name: ";
    is >> cust.name;
    cout << "ID: ";
    is >> cust.id;
    return is;
}
}; // End of Class 'Customer'
```

* Customer (string name = "NA"): the constructor of the class Customer. Default value of parameter 'name' is "NA"
* ~ Customer (): destructor for class Customer.

Hardik Soni
20CS30023

* friend ostream &operator<<(ostream &os, const Customer &cust)

: The function is responsible for cust handling output stream of class Customer. In a given format it prints details of class Customer.

* friend istream &operator>>(istream &is, const Customer &cust)

: The function is responsible for handling input stream of class Customer. In the given format it takes details of class Customer.

```
class ProductItem{
private: string title; // title is stored as string
         size_t id; // id as unsigned int
         static size_t NumProductItem;
         float price; // Number of instances as
         int copies;  a static unsigned int.
public:
    ProductItem(string title="NA", float price=0){
    this->title = title;
    this->id = ++NumProductItem;
    this->price = price;
    this->copies = 0;
    }
    ~ProductItem(){
    this->title = "NA";
    this->copies = 0;
    this->price = 0;
    }
```

Hardik Soni
20CS30023


```
friend ostream &operator<<(ostream &os, const ProductItem &pi)
{
    cout << "\n ** Product Item ** " << endl;
    os << " Title:" << pi.title << " ID: " << pi.id << endl;
    os << "Price:" << pi.price << " Copies:" << pi.copies << endl;
    cout << std::endl;
    return os;
}
friend istream &operator>>(istream &is, const ProductItem &pi)
{
    cout << " ** Product Item ** " << endl;
    cout << " Title:";
    is >> pi.title;
    cout << "ID:";
    is >> pi.id;
    cout << "Price:";
    is >> pi.price;
    cout << " Copies:";
    is >> pi.copies;
    cout << std::endl;
    return is;
}
ProductItem &operator+(int a){
    this->copies += a; // Adding copies to the Product Item
    return *this;
}
ProductItem &operator=(ProductItem const &pi){
    if (this != &pi)
    {
        // overloading Assignment operator
```

Harsh Soni

2015300123

```
    this→title = pi.title;
    this→id = pi.id;
    this→copies = pi.copies;
    this→price = pi.price;
}
return *this;
}
```

* Product Item(string title = "NA", float price = 0): The constructor for the class ProductItem.
* ~ProductItem(): The destructor for the class ProductItem.
* friend ostream &operator<<(ostream &os, const ProductItem &pi);

The function is responsible for handling output stream of class ProductItem. In a given format it prints the contents of class ProductItem. namely: title, id, price and copies.

* friend istream &operator>>(istream &is, const ProductItem &pi);

The function is responsible for handling input stream of class ProductItem. In the given format it takes the input of class ProductItem. namely: title, id, price and copies.

* ProductItem &operator*(int a): this function overloads * for ProductItem, so that it helps us to add a certain number

Hardik Soni
20CS30023

of products 'a' to Order.

* ProductItem &operator=(ProductItem const &p); This function is responsible for overloading the '=' operator for ProductItem. It checks first if given reference is not equal to given instance if it not then assigns and then returns it / return it.

```
class Order{
  size_t id;
  static size_t NumOrders;
  Customer c;
  vector<ProductItem> prods;

public:
  Order(Customer &in_c)
  {
    this->id = ++NumOrders;
    this->c = in_c;
  }
  ~Order()
  {
    this->c.~Customer();
    this->prods.clear();
  }
  size_t getid() const // getter for id
  {
    return this->id;
  }
```

```
friend ostream &operator<<(ostream &os, const Order &O)
{
  cout << " ** Order ** " << endl;
   os << O.c;
  cout << " ID: " << endl;
   os << O.id;
  cout << "Products Size: " << O.prods.size() << endl;
   for (auto &prod : O.prods)
        os << cout << prod;
   cout << endl;
   return os;
}
Order &operator+(ProductItem &p)
{
  this->prods.push_back(p); // Adding a ProductItem to order
  return *this;
}
Order &operator=(Order const &O)
{                          // Overloading Assignment operator
  if(this != &O)
  {
    this->id = O.id;
    this->c = O.c;
    this->prods = O.prods;
  }
  return *this;
}
}; // End of class 'Order'
```

Harshik Soni
2015300025


```
friend istream &operator>> (istream &is, Order &o)
{
  cout << " ** Order ** " << std::endl;
  cout << " ID: " // The input stream used for Formatted Input to class Order.
  is >> o.id; // Input stream of 'id'
  is >> o.c; // Input stream of 'c'.
  return is;
}
```

Hardik Soni
20CS30023

* Order (customer &inc_c): The constructor for the class Order.
* ~Order(): the destructor for the class Order.
* friend ostream &operator<<(ostream &os, const Order &o): this function is responsible for output output stream of class Order. In the given format it prints entities of class Order: Customer, ID, Product Size and Products Information.

* friend istream &operator>>(istream &is, const Order &o) this function is responsible for input stream of class Order. In the given format it takes input for Customer, ID.

* Order &operator+(ProductItem &p): this method overloads the operator '+' for class Order. It helps you add a certain productItem to a given instance of class Order.

* Order &operator=(Order const &o): This function is responsible for overloading the operator '=' for class 'Order'. It checks if given reference is not equal to given reference is not equal to given instance if it is not then assigns it and then returns it/assigns.

Hardik Soni
20CS30023


```
class ShoppingBasket
{
  size_t id;
  static size_t NumBasket;
  Customer c;
  list<Order> orders;
public:
  ShoppingBasket(Customer const &in_c)
  {
    this->id = ++NumBasket;
    this->c = in_c;
  }
  ~ShoppingBasket()
  {
    this->id--;
    this->c.~Customer();
    this->orders.clear();
  }
  friend ostream &operator<<(ostream &os,
                             const ShoppingBasket
                             &sb)
{
  cout << "** Shopping Basket **" << endl;
  os << sb.c;
  cout << "ID:";
  os << sb.id;
  cout << " orders size: " << orders.size()
       << endl;
  for (auto it = sb.orders.begin()
       ; it != sb.orders.end(); it++)
    cout << it.operator*();
  cout << endl;
```

Hardik Soni
2021530023


```
return os;
}
friend istream &operator>>(istream &is,
                           ShoppingBasket &sb)
{
  cout << " Shopping Basket " << endl;
  cout << " ID: ";
  is >> sb.id;
  is >> sb.c;
  return is;
}
ShoppingBasket &operator+(Order &p)
{                          // Adding Order
  this->orders.push_back(p); // to SB
  return *this;
}
ShoppingBasket &operator-(int orderid)
{
  this->orders.remove_if([&orderid]
                         (const Order &n)
                         { return n.getid() ==
                           orderid; });
                         // Removing Order from
                         ShoppingBasket
  return *this;
}
ShoppingBasket &operator=(ShoppingBasket const &sb)
{
  if (this != sb)  // overloading the
                   Assignment operator
  {
    this->id = sb.id;
    this->c = sb.c;
    this->orders.clear();
```

Hardik Soni
2015130023


```
        this->orders.assign(sb.orders.begin(),
                            sb.orders.end());
    }
    return *this;
}; // End of class ShoppingBasket.
```

* ShoppingBasket(Customer& const in_c): The constructor for the class ShoppingBasket. This Customer for the given ShoppingBasket is assigned to the given Customer Reference as Input.
* ~ShoppingBasket(): The destructor for the Class ShoppingBasket
* friend ostream& operator<<(ostream& os, const ShoppingBasket& sb)

this function is responsible for output stream of class ShoppingBasket. In the given format it prints Customer, ID, Orders.

* friend istream& operator>>(istream& is, const ShoppingBasket& sb)

this function is responsible for Input stream of Class ShoppingBasket. In the given format

Hardik Soni

201530023

it takes input for ID and customer.

* ShoppingBasket operator+(Order &p): Here, we are overloading the operator '+' for class ShoppingBasket. In:

```
this → orders.push_back(p);
```

we are adding Order p to orders entity of class ShoppingBasket.

* ShoppingBasket &operator-(int orderid): Here we are overloading the operator '-' for class ShoppingBasket. In:

```
this→orders.remove_if([&orderid](order n)
    { return n.getid() == orderid; });
```

we are checking if the orderid of Order n is equal to the given orderid to remove.

* ShoppingBasket &operator=(ShoppingBasket const &sb): This function is responsible for overloading the operator '=' for class ShoppingBasket. It checks if given reference is equal to given instance of class 'ShoppingBasket'. If it is not then assigns all corresponding entities namely: customer, ID and orders and then returns it.

Hardik Soni
20CS30023


```
// Initialising the static variables of the
      Respective classes.
size_t Customer::NumCustomer = 0;
size_t ProductItem::NumProductItem = 0;
size_t Order::NumOrder = 0;
size_t ShoppingBasket::NumBasket = 0;

// All the unsigned variables
```

* Main *

```
int main(){ // create a customer
Customer *c = new Customer("Nikhil");
ProductItem *p = new ProductItem("Something");
// created a product item
Order* o = new Order(*c);
// add 10 copies of p to o
Order &oref = *o;
oref = oref + *p * 10;
ShoppingBasket *s = new ShoppingBasket
                        (*c);
ShoppingBasket &shop = *s;
shop = shop + oref;
shop = shop - oref.getid();
return 0;
}
```