Online Shopping System

OOP Final Project

Members:

- Saad Bin Khalid: 20k-0161 **(F)**
- Muhammad Moaaz Bin Sajjad: 20k-0154 **(F)**

Brief Description:

Our project is an online shopping system which utilizes the abilities of C++ language while abiding by the rules and regulations of OOP. The project is a console application which primarily consists of two pieces of code:

1. The management layer.
2. The networking layer.

Management Layer

Features:

The features implemented are as follows:

- Customers sign up or login.
- Customer details are stored in “customer.txt”.
- Customers buy items from the online shop.
- Before confirming, the bill is shown to them.
- They then have the option to confirm, cancel, or change (add or remove anything).
- Customers can write complaints.
- Any employee (or admin) can answer the complain.
- Employees and admin will avail 20% discount on purchases.
- Employee details will be stored in “emp.txt”.
- Employee accounts are made by the admin.
- The chats will be stored in a binary file, “complaint.dat”.
- Customers and employees alike can view their profile.
- Balance will be automatically deducted whenever a purchase is made.
- Details about goods are stored in “goods.txt”.
- Details about cash (current amount, inflow, outflow) stored in “cash.txt”.
- Automatic reorder is placed if goods fall below a certain level.

- Admin can view record of customers.
- He/she can also view record of employees.
- He/she can fire any employees and ban any customers.
- He/she can view the inventory level and manually order goods.
- He/she can also manually update balance.
- He/she can search for customers and employees.
- Admin also sets the reorder level (level to which goods fall after which a reorder is placed) and amount (how many goods to be reordered) and unit price.
- Admin can also view cash (we will start with an initial amount. Every time a customer buys, cash increases. Every time reorder placed, cash deducted).

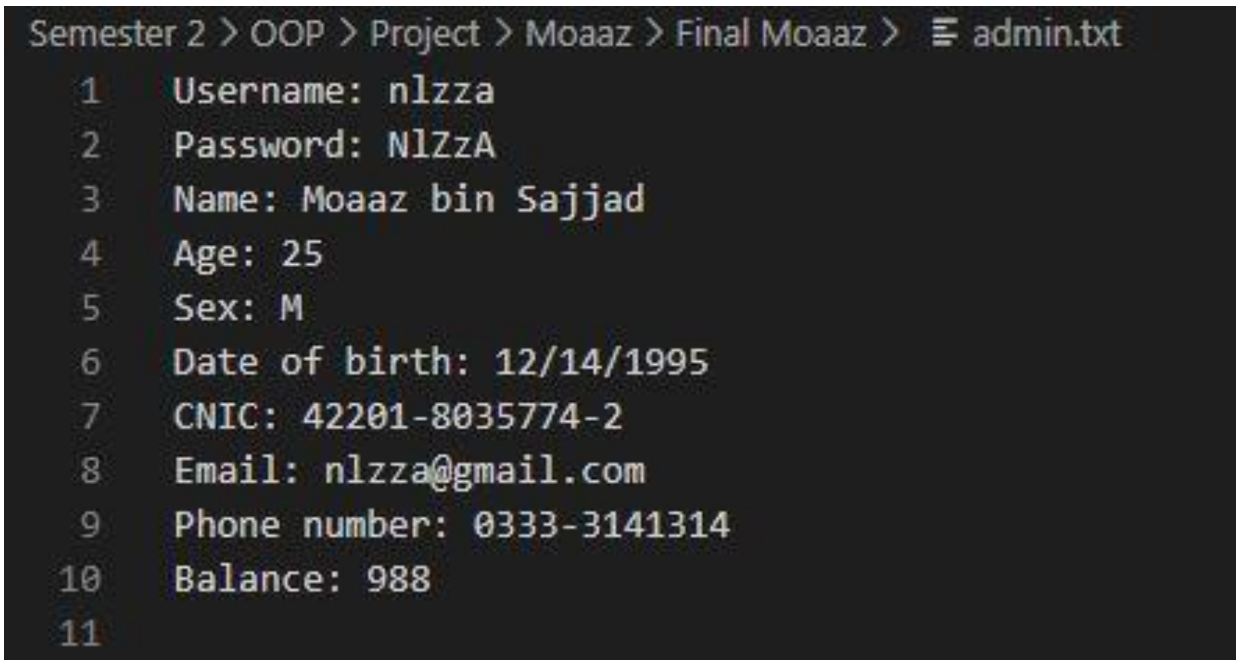
Files:

1. admin.txt:

It stores the following details about the admin:

- Username
- Password
- Name
- Age
- Sex (M for male, F for female, and O for other)
- Date of Birth (mm/dd/yyyy)
- CNIC
- Email
- Phone Number
- Balance

Below is a picture of “admin.txt”:


Semester 2 > OOP > Project > Moaaz > Final Moaaz > admin.txt

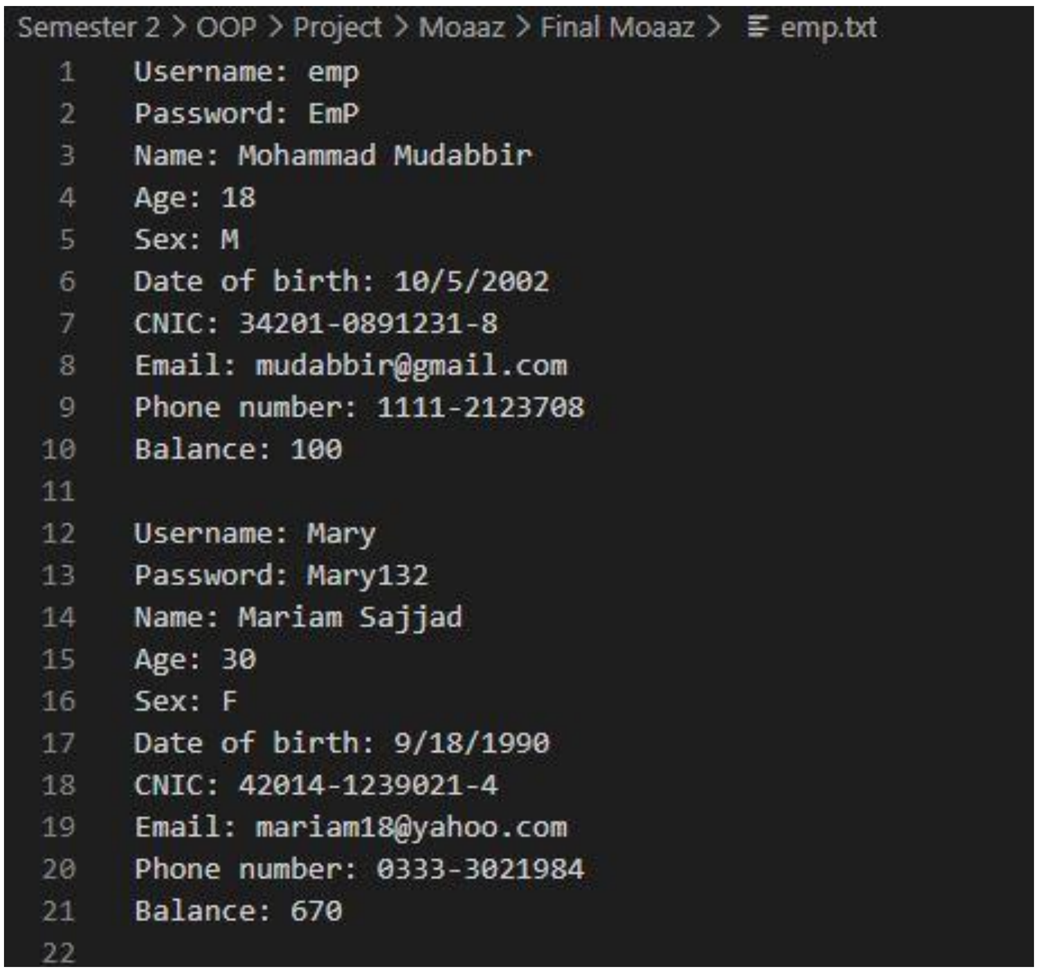
```
Username: nlzza
Password: NlZzA
Name: Moaaz bin Sajjad
Age: 25
Sex: M
Date of birth: 12/14/1995
CNIC: 42201-8035774-2
Email: nlzza@gmail.com
Phone number: 0333-3141314
Balance: 988
```

2. emp.txt:

It stores employee records. A blank line separates the records. The following details are stored:

- Username
- Password
- Name
- Age
- Sex (M for male, F for female, and O for other)
- Date of Birth (mm/dd/yyyy)
- CNIC
- Email
- Phone Number
- Balance

Below is a picture of “emp.txt”:



```
Username: emp
Password: EmP
Name: Mohammad Mudabbir
Age: 18
Sex: M
Date of birth: 10/5/2002
CNIC: 34201-0891231-8
Email: mudabbir@gmail.com
Phone number: 1111-2123708
Balance: 100

Username: Mary
Password: Mary132
Name: Mariam Sajjad
Age: 30
Sex: F
Date of birth: 9/18/1990
CNIC: 42014-1239021-4
Email: mariam18@yahoo.com
Phone number: 0333-3021984
Balance: 670

```

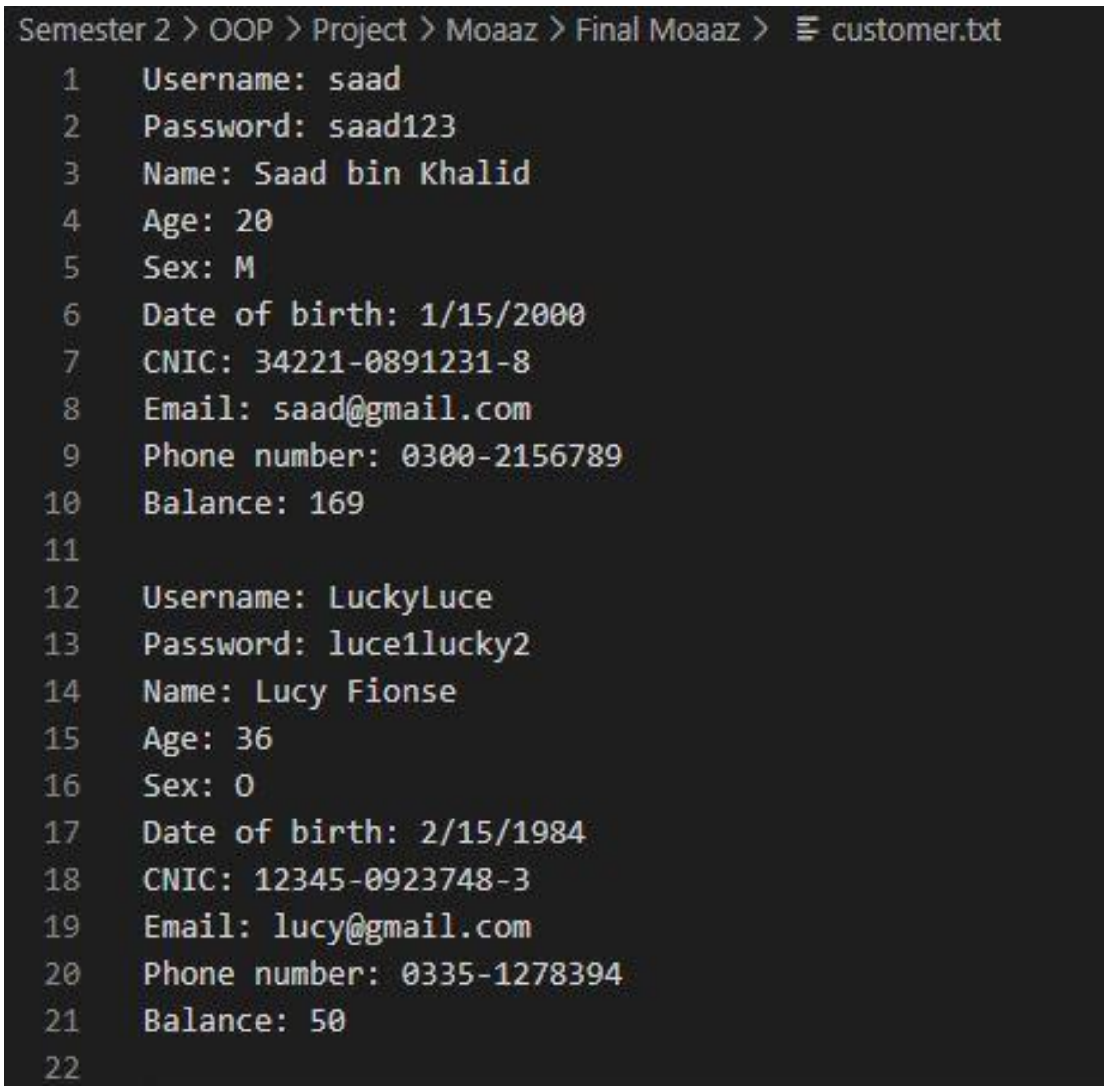
3. customer.txt:

It stores customer records. A blank line separates the records. The following details are stored:

- Username
- Password
- Name
- Age

- Sex (M for male, F for female, and O for other)
- Date of Birth (mm/dd/yyyy)
- CNIC
- Email
- Phone Number
- Balance

Below is a picture of “customer.txt”:



Semester 2 > OOP > Project > Moaaz > Final Moaaz > customer.txt

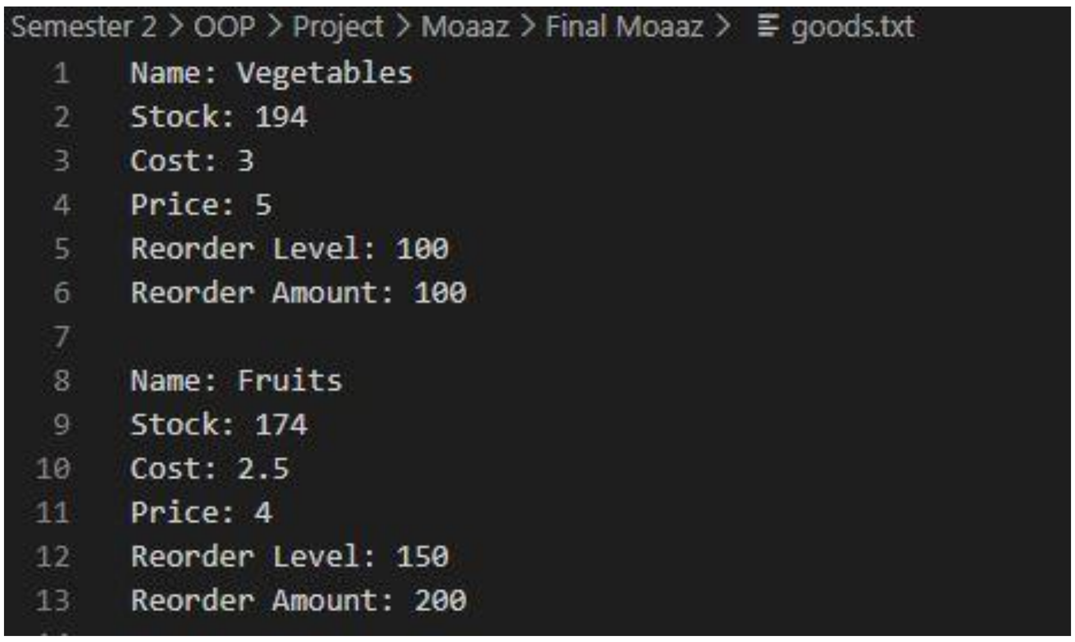
```
Username: saad
Password: saad123
Name: Saad bin Khalid
Age: 20
Sex: M
Date of birth: 1/15/2000
CNIC: 34221-0891231-8
Email: saad@gmail.com
Phone number: 0300-2156789
Balance: 169

Username: LuckyLuce
Password: luce1lucky2
Name: Lucy Fionse
Age: 36
Sex: O
Date of birth: 2/15/1984
CNIC: 12345-0923748-3
Email: lucy@gmail.com
Phone number: 0335-1278394
Balance: 50

```

4. goods.txt:

It stores the following details about goods:

- Name
- Stock (how many items are in inventory)
- Cost (per unit cost)
- Price (per unit selling price)
- Reorder level (level to which goods fall after which a reorder is placed)
- Reorder amount (how many goods to be reordered)

Below is a picture of “goods.txt”:


Semester 2 > OOP > Project > Moaaz > Final Moaaz > goods.txt

```
Name: Vegetables
Stock: 194
Cost: 3
Price: 5
Reorder Level: 100
Reorder Amount: 100

Name: Fruits
Stock: 174
Cost: 2.5
Price: 4
Reorder Level: 150
Reorder Amount: 200
```

5. complaint.dat:

It stores the complains made by customers, if any. The file is binary for privacy and security reasons. The following details are stored about every complaint:

- Complaint
- Its answer (if it has been given)
- Username of the customer who made the complaint.
- Whether it has been seen by an employee (or admin) or not.
- Whether it has been answered or not.

Functions:

void welcome(void):

This function, defined in person.h, prints “WELCOME TO MSM GROCERY CENTER” as a centered heading.

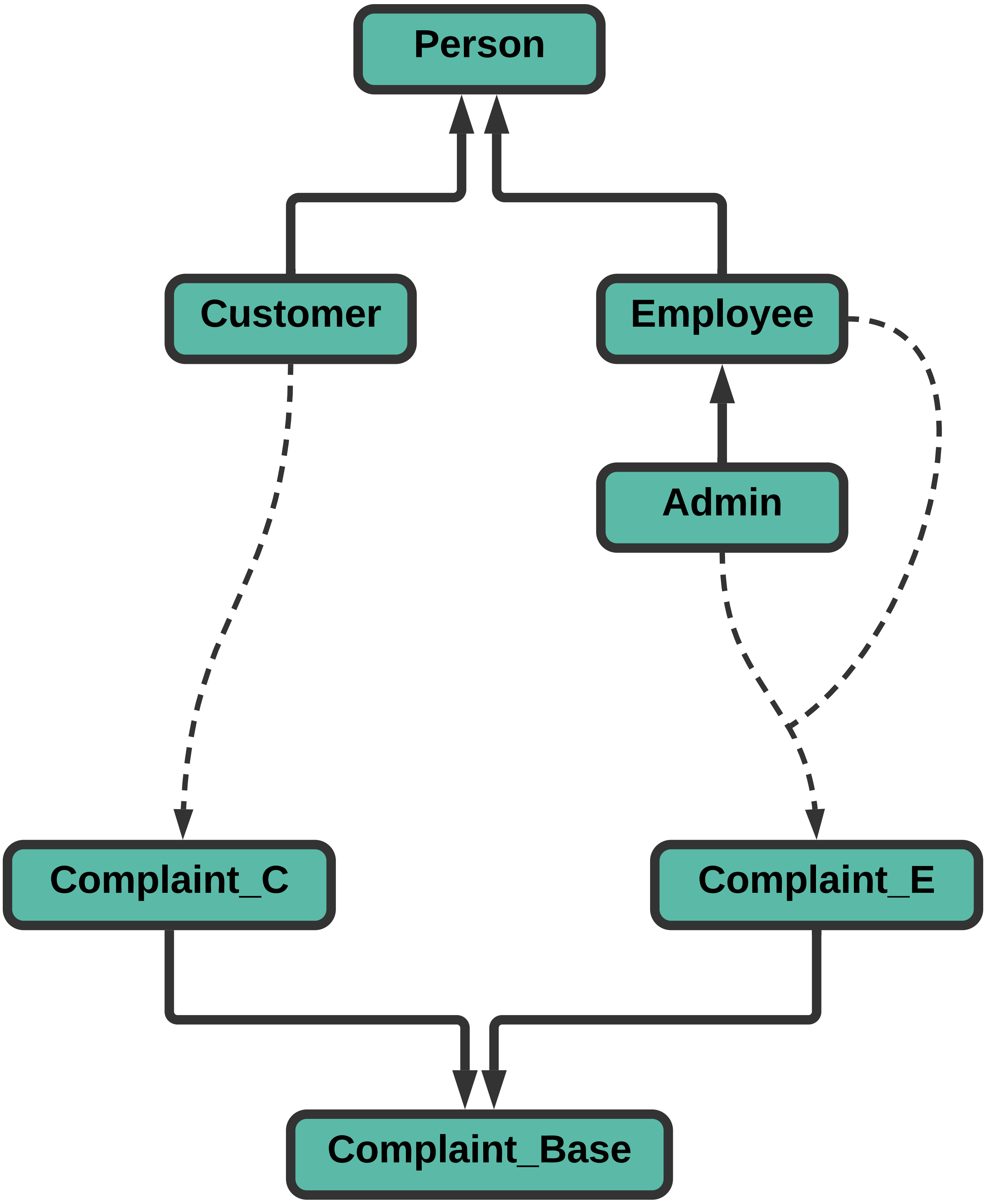
void heading(const string& head):

This function, defined in person.h, prints *head,* taken as argument, as a centered heading.

int main(void):

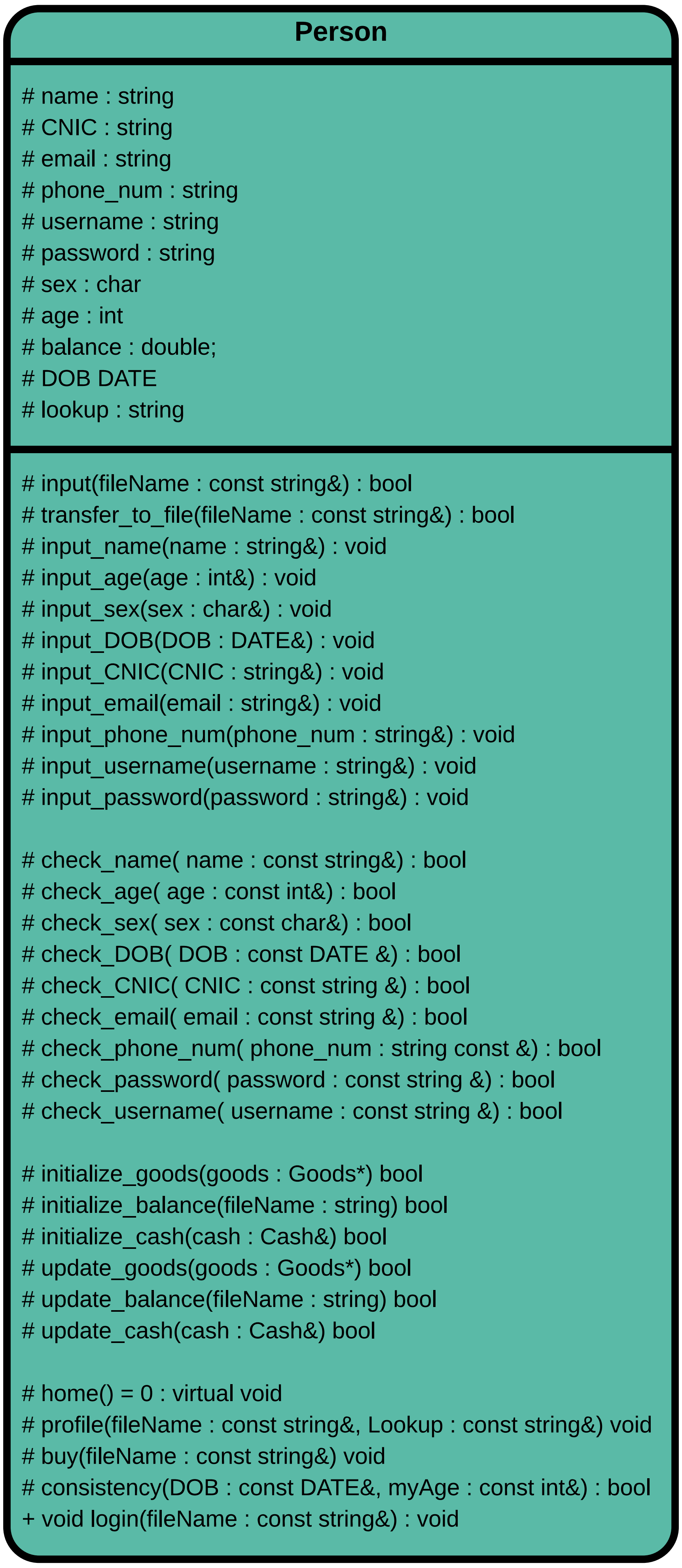
This is the function called at the start of the program. It presents users the choice to proceed as admin, employee, or customer. An invalid selection results in error message. Electing to exit ends the program. Selecting admin creates admin, an object of Admin, and calls admin.login(adminFile). Selecting employee creates employee, an object of Employee, and calls employee.login(empFile). Selecting customer creates customer, an object of Customer, and calls customer.start().

Classes:


Person
Customer
Employee
Admin
Complaint_C
Complaint_E
Complaint_Base

Person:

It is defined in Person.h, and its methods are defined in Person.cpp.



```
Person

# name : string
# CNIC : string
# email : string
# phone_num : string
# username : string
# password : string
# sex : char
# age : int
# balance : double;
# DOB DATE
# lookup : string

# input(fileName : const string&) : bool
# transfer_to_file(fileName : const string&) : bool
# input_name(name : string&) : void
# input_age(age : int&) : void
# input_sex(sex : char&) : void
# input_DOB(DOB : DATE&) : void
# input_CNIC(CNIC : string&) : void
# input_email(email : string&) : void
# input_phone_num(phone_num : string&) : void
# input_username(username : string&) : void
# input_password(password : string&) : void

# check_name( name : const string&) : bool
# check_age( age : const int&) : bool
# check_sex( sex : const char&) : bool
# check_DOB( DOB : const DATE &) : bool
# check_CNIC( CNIC : const string &) : bool
# check_email( email : const string &) : bool
# check_phone_num( phone_num : string const &) : bool
# check_password( password : const string &) : bool
# check_username( username : const string &) : bool

# initialize_goods(goods : Goods*) bool
# initialize_balance(fileName : string) bool
# initialize_cash(cash : Cash&) bool
# update_goods(goods : Goods*) bool
# update_balance(fileName : string) bool
# update_cash(cash : Cash&) bool

# home() = 0 : virtual void
# profile(fileName : const string&, Lookup : const string&) void
# buy(fileName : const string&) void
# consistency(DOB : const DATE&, myAge : const int&) : bool
+ void login(fileName : const string&) : void
```

Attributes:

- **string name:** It stores the name.
- **string CNIC:** It stores the CNIC.
- **string email:** It stores the email address.
- **string phone_num:** It stores the phone number.
- **string username:** It stores the username.
- **string password:** It stores the password.
- **char sex:** It stores M for male, F for female, and O for other.
- **int age:** It stores the age.
- **double balance:** It stores the balance. Initialized to 0 by default.
- **string lookup:** It stores the first line of the record in the file. That first line is then used to look up the record in the future.
- **DATE DOB:** DATE is a struct, defined in person.h, which stores the date in mm/dd/yyyy format. It has three attributes – day, month, and year to store the day, month, and year respectively. DOB stores the person's date of birth.

Methods:

- **void input_name(string& name):** It inputs name and calls check_name(name) to validate the entered name. The function exits if the name is successfully validated; otherwise, it re-calls itself.
- **bool check_name(const string& name):** This function is used to validate name. A range-based for loop is used to read the characters one by one. Then an if statement is used to ensure that the character is a letter or a space. If it is anything else, an error message is printed to the screen, and false is returned, indicating that the name is incorrect. If the for loop exits successfully (i.e no character violates the rules), then true is returned.
- **void input_age(int& age):** It inputs age and calls check_age(age) to validate the entered age. The function exits if the age is successfully validated; otherwise, it re-calls itself.
- **bool check_age(const int& age):** This function is used to validate age. An if statement is used to ensure that age is within 10 and 99 inclusive. If it is not, an error message is printed, and false is returned, indicating that the age is incorrect; otherwise, true is returned.
- **void input_sex(char& sex):** It inputs sex and calls check_sex(sex) to validate the entered sex. The function exits if the sex is successfully validated; otherwise, it re-calls itself.
- **bool check_sex(const char& sex):** This function is used to validate sex. An if statement is used to ensure that only 'M' (for male), 'F' (for female), and 'O' (for other) get accepted as inputs. If sex is equal to

'M', 'F', or 'O', true is returned; otherwise, an error message is printed, and false is returned.

- **void input_DOB(DATE& DOB, const int& myAge):** It inputs date of birth in mm/dd/yyyy format. scanf() is used to input the date in the correct format. check_DOB(DOB) is then called to ensure that the year, month, and day values are correct. Then consistency(DOB, myAge) is called to check whether the date of birth is consistent with the entered age. The function exits if the date of birth passes both the validation checks; otherwise, it re-calls itself.
- **bool check_DOB(const DATE& DOB):** This function us used to validate the date of birth. First, an if statement is used to ensure that the year is between 1921 and 2011 inclusive. If it is not, error message is printed and false returned. Then another if statement is used to ensure that the month is between 1 and 12 inclusive. If it is not, error message is printed and false returned. Then finally, we use a switch-case to validate day. If month is 1, 3, 5, 7, 8, 10, or 12; we ensure that day is between 1 and 31 inclusive. If month is 4, 6, 9, or 11; we ensure that day is between 1 and 30 inclusive. Else, we ensure that day is between 1 and 28 inclusive (we have assumed that there is no leap year). If the validation for day is unsuccessful, error message is printed and false returned. If all these validation tests are passed successfully, true is returned.
- **bool consistency(const DATE& DOB, const int& myAge):** This function is used to ensure that the entered date of birth is consistent with the person's age. ctime library has been included for this purpose. now, a variable of time_t (a datatype defined in ctime) stores the number of seconds elapsed since 1 Jan 1970. localtime(&now) returns a pointer to the tm structure (defined in ctime) representing the current local time and date. We then calculate the age. If the age is same as entered age, true is returned; otherwise, error message is printed and false returned.
- **void input_CNIC(string& CNIC):** It inputs CNIC and calls check_CNIC(CNIC) to validate the entered CNIC. The function exits if the CNIC is successfully validated; otherwise, it re-calls itself.
- **bool check_CNIC(const string& CNIC):** This function is used to validate CNIC. We first use an if statement to ensure that CNIC has exactly 15 characters. If not, error message is printed and false returned. Then a range-based for loop is used to read the characters one by one. An if statement is used to ensure that the characters are numbers or -. If any other character is encountered, error message is printed and false returned. We then check the pattern. We do this by ensuring that there is a – at positions 5 and 13 and numbers at

every other position. The function returns true only if all these tests are passed successfully.

- **void input_email(string& email):** It inputs email and calls check_email(email) to validate the entered email. The function exits if the email is successfully validated; otherwise, it re-calls itself.
- **bool check_email(const string& email):** This function is used to validate the email. This is done by ensuring the presence of @. A bool called found is used for this purpose. It is initially false. A range based for loop is used to go through all the characters of the email one by one. If @ is found, found is changed to true, and break is used to terminate the loop. If found is true, the function returns true indicating successful validation; otherwise, error message is printed and false returned.
- **void input_phone_num(string& phone_num):** It inputs phone number and calls check_phone_num(phone_num) to validate the entered phone number. The function exits if the phone number is successfully validated; otherwise, it re-calls itself.
- **bool check_phone_num(const string& phone_num):** This function is used to validate the phone number. A range-based for loop is used to read the characters one be one. Then an if statement is used to ensure that character is a number, or – or # or *. If it is anything else, error message is printed and false returned. If the loop ends successfully (i.e no character violates the rules), true is returned.
- **void input_username(string& username):** This function inputs username. The function has been made inline due to its small size.
- **void input_password(string& password):** This function inputs password. getch(), defined in conio.h, is used to input characters. A * appears on the screen with the input of each character. The characters are appended to password via push_back() function. Once the password has been entered, check_password(password) is called for double-entry verification. The function exits if this verification is successful; otherwise, it re-calls itself.
- **bool check_password(const string& password):** This function verifies the password by having the user re-enter the password. This re-entered password is stored in a string called confirm. Once the password has been re-entered, confirm (storing the re-entered password) and password (storing the original password) are compared. If they match, true is returned, indicating successful verification; otherwise, error message is printed and false returned.
- **bool transfer_to_file(const string& filename):** It writes the entered details to the file passed as argument. The file is opened for output in append mode. A check is used to ensure successful opening of

the file. Then the details are written to the file, after which another check is used to ensure that the writing operation was carried out successfully. If everything is done successfully, the function returns true; otherwise, false is returned denoting error.

- **bool input(const string& filename):** This function is used to input all the details. It calls all the input functions one by one, and then finally calls transfer_to_file(fileName) to write the contents to the file passed as argument. The bool returned by transfer_to_file(fileName) is also returned by this function.
- **void login(const string& fileName):** This function carries out the login operation. It prompts the user to enter the username and password. Password is entered via getch() and esterisks are printed to the screen in place of actual characters. Username is stored in username and password stored in password. Then the file passed as argument is opened in input mode. Two bools, userFound and passFound, initialized to false, are used to determine whether the username and password have been found. Lines are read one by one until the end of file. If statement is used to check whether the line contains entered username. If it does, userFound is changed to true, and we match the next line’s password with the entered password. If password also matches, passFound is changed to true, the first line of the record which matched (the line containing username) is stored in lookup, file is closed, and home() called. Otherwise, we continue our search. If the file ends without a successful result, we print error message and re-call login(fileName) to have the user re-try.
- **virtual void home(void):** This is a pure virtual function which has been defined inside the child classes of Person. This function makes Person an abstract class.
- **void profile(const string& fileName, const string& Lookup):** This function displays the profile. Since it is a feature common to Admin, Employee and Customer; therefore, we have defined it in Person which is the base class. This function opens the file passed as argument in input mode. If file fails to open correctly, error message is printed, and the function ends bringing the user back to home page. The file is read line-by-line until the end of file. The lines are compared with Lookup (which is same as lookup, the attribute of Person). If the lines match, it means we have found our record. Upon this, we print out the record. The printing of lines read from the file stops once we encounter a blank line which indicates the end of the record.
- **void buy(const string& fileName):** This function enables customers, admin, and employees to buy goods from the store. The feature is common to Admin, Employee, and Customer, so it has

been defined in the base class – Person. First, an array of N goods is declared. N is a macro defined to 10 in Person.h. N indicates the number of goods the store sells. After this, we declare cash, an object of Cash. Since, Person is a friend class of Goods and Cash, therefore, this function has access to the private attributes of Goods and Cash. The goods, balance, and cash are initialized via initialize_goods(goods), initialize_balance(fileName) and initialize_cash(cash) respectively. If any of these functions fails (i.e returns false), error message is printed and buy() ends bringing the user back to home page. After this, the entire menu is printed, and the user is given choice to add items to order, confirm order, or go back. Switch-case is used to determine what to do next. If an invalid selection is made, error message is printed, and the user has to re-select. If the user selects to go back, go (a bool which is true) is changed to false which means we exit the loop, upon which the buy() function ends, and the user returns to home page.

If the user elects to add items to order, the user is first asked the item number which must be between 1 and 10; this is validated via a do-while enclosed inside an if statement. Then the quantity is enquired. The quantity must be positive, but it must not exceed the quantity of that item in stock; this is validated via a do-while enclosed inside an if statement. Once this is done, the item is added to stock. Its bought attribute (which was initialized to false in initialize_goods(goods)) is changed to true, indicating that it has now been purchased. Meanwhile, the quantity attribute is set to the quantity entered by the user.

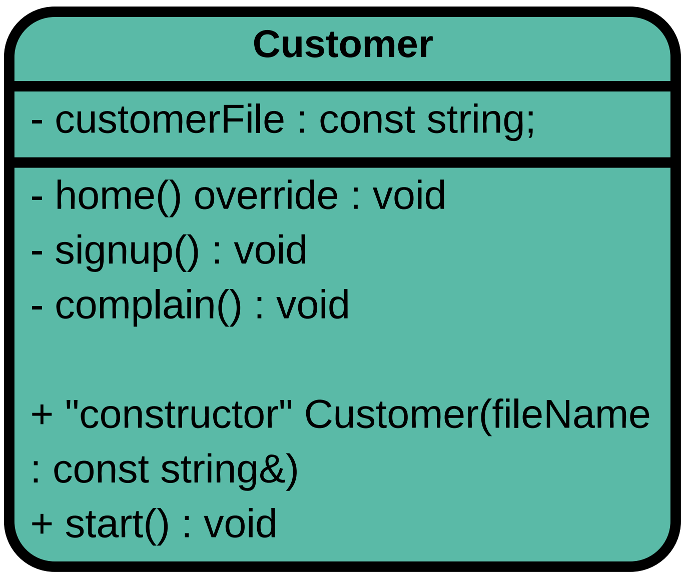
If the user elects to confirm order, we first declare two variables – order, a bool set to false which checks whether an order has actually been placed, and ordercost, a double initialized to zero which stores the cost of the order. We run a for loop from 0 to N, going through all of the goods in the goods array. If a good’s bought attribute is true (i.e it has been bought), we change order to true, and we add its cost to ordercost. The cost is calculated by multiplying price with quantity. In case of employees and admin, a 20% discount is availed. Once we come out of the loop, we check the status of order. If order is false (i.e no order has been placed), an error message is printed, and the user has to re-select. Otherwise, we proceed to check whether ordercost exceeds user’s balance. If it does, error message is printed, and the user has to re-select. If it does not, we proceed to deduct balance and update cash. We then start another for loop going through all of the goods. If the good has been bought (i.e its bought attribute is true), its stock is deducted. Then we check whether the stock is lower than re-order level and whether there is enough cash to be able to re-order. If both the conditions are true, a re-order is placed. The stock is increased and cash deducted. After that, the files are updated by calling update_balance(fileName), update_goods(goods), and update_cash(cash). If any of these functions fails (i.e returns false), error message is printed and buy() ends bringing the user back to home page. Otherwise, message is printed indicating that the order has been successfully placed.

- **bool initialize_goods(Goods* goods):** The purpose of this function is to initialize the goods array which is passed as an argument. First, “goods.txt” is opened in input mode. If it fails to open, false is returned to indicate erroneous termination. Then we start a loop from 0 to N. We go through the array and initialize every item by reading from the file. We then close the file and return true to indicate successful termination.
- **bool initialize_balance(string fileName):** The purpose of this function is to initialize the balance. The file passed as argument is opened in input mode. If it fails to open, false is returned to indicate erroneous termination. We read the file until we reach the requisite record. We then reach the balance line and extract it. After that, we close the file and return true, indicating successful termination.
- **bool initialize_cash(Cash& cash):** The purpose of this function is to initialize cash. First, “cash.txt” is opened in input mode. If it fails to open, false is returned to indicate erroneous termination. We read the file and initialize all the attributes of cash. We then close the file and return true to indicate successful termination.
- **bool update_goods(Goods* goods):** The purpose of this function is to update “goods.txt”. “goods.txt” is opened in output mode. If file fails to open successfully, false is returned to denote erroneous termination. We then run a for loop from 0 to N, writing the details of each item to the file. Once the loop ends, the file is closed and true is returned to indicate successful termination.
- **bool update_balance(string fileName):** The purpose of this function is to update balance in the file passed as argument. Two files are opened for this purpose, the original file in input mode and a new file in output mode. If files fail to open successfully, false is returned to denote erroneous termination. We then copy contents to the new file and replace the old balance with the new one. The files are then closed. The old file is deleted, and the new one is renamed to the name of the old file. True is returned to indicate successful termination.
- **bool update_cash(Cash& cash):** The purpose of this function is to update “cash.txt”. “cash.txt” is opened in output mode. If file fails to open successfully, false is returned to denote erroneous termination.

We write the attributes of cash to the file. Finally, the file is closed, and true is returned to indicate successful termination.

Customer:

It is defined in Customer.h, and its methods are defined in Customer.cpp. It is a derived class of Person.



Attributes:

- **const string customerFile:** It stores the name of the customer file, i.e customer.txt. It is initialized by a parameterized constructor.

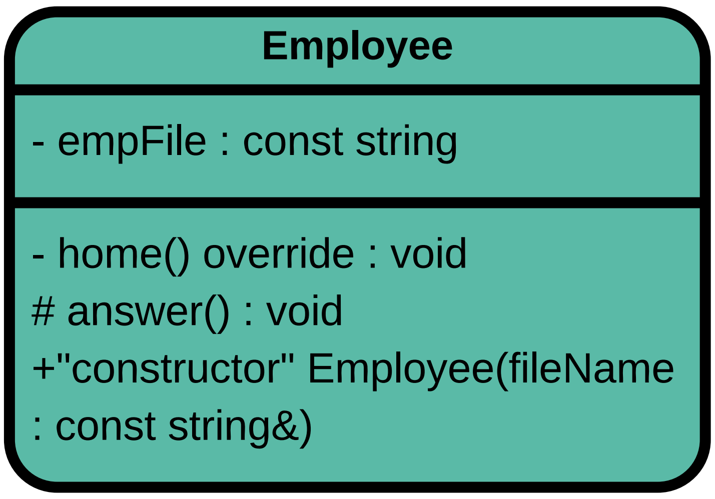
Methods:

- **void start(void):** This function gives customers a choice to login or signup or go back. An error message is displayed in case of invalid selection. In case of back, go (a bool set to true) is changed to false; the while loop breaks and the function ends, taking the user back to main(). In case of signup, signup() is called, and if customer elects to login, login(customerFile) is called.
- **Customer(const string& fileName):** It is a parameterized constructor which initializes customerFile to fileName.
- **void signup(void):** This function calls input(customerFile) which then does the rest of the work. If the input() function works correctly, it returns true, and message is displayed indicating that the account has been successfully created. If input() returns false, error message is displayed.
- **void home(void):** This function is the home page for customers. It gives customers choice to buy, complain, view profile or log out. A switch-case is used to call the appropriate function as per the customer’s selection.
- **void complain(void):** This function enables users to write complaints. First, complaint, an object of Complaint_C, is made. The customers are given option to write complain, review complain,

change complain, or go back. In case of first option, they are aske to write the complaint. Then complaint.write(text) is called to write the complain to complaint.dat. If complaint is successfully written, a message is displayed indicating so to the customer; otherwise, an error is displayed.
If customer elects to review the complaint, complaint.reView() is called. If customer wants to update complaint, he/she is asked to write the new complaint. Then complaint.update(newText, AnsStatus) is called. If it is successful, a message is displayed indicating so; otherwise, an error message is displayed. In case of invalid choice, customer is asked to re-select. If customer elects to go back, the function ends, and customer returns to home page.

Employee

It is defined in Employee.h, and its methods are defined in Employee.cpp. It is a derived class of Person.



Attributes

- **const string empFile:** It stores the name of the employee file, i.e emp.txt. It is initialized by a parameterized constructor.

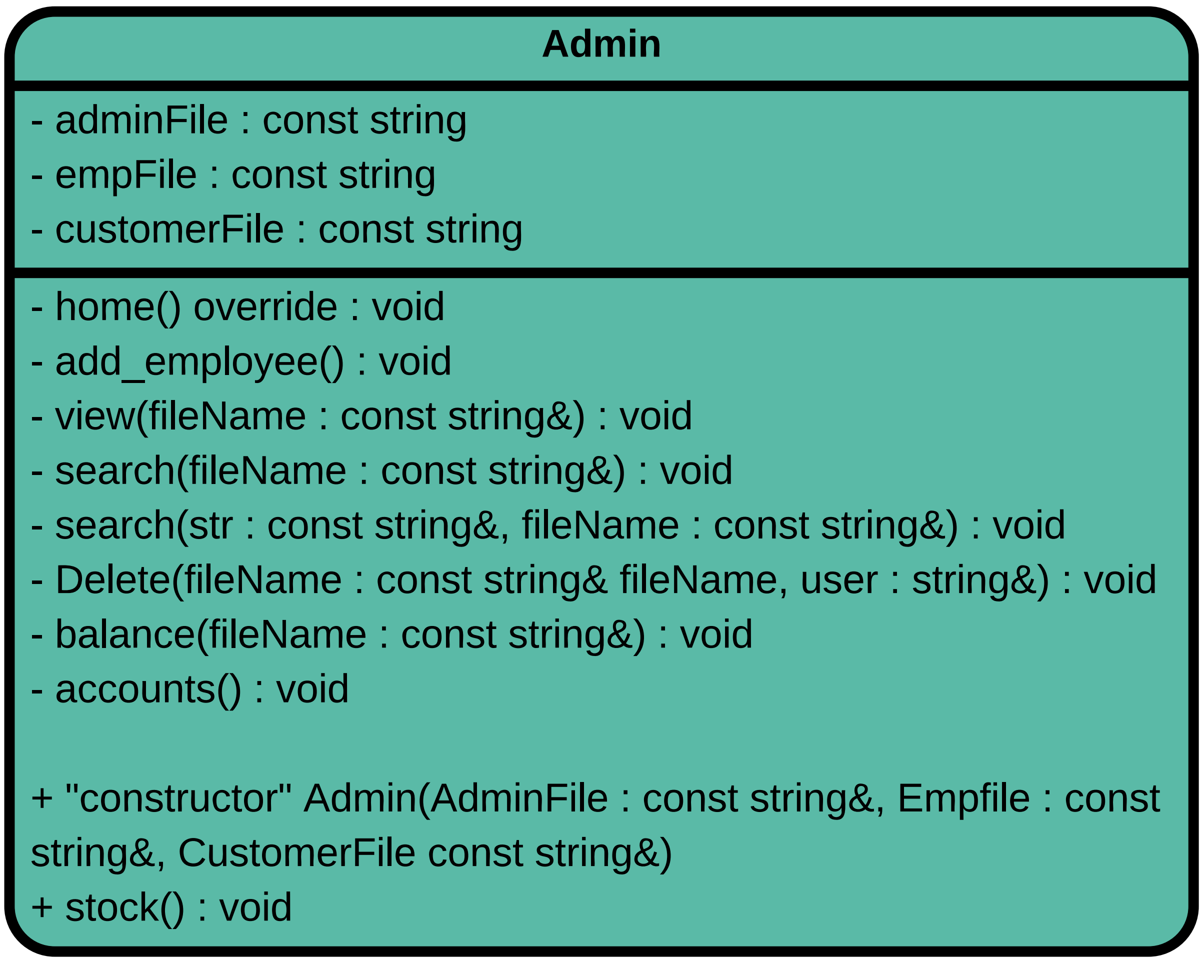
Methods

- **Employee(const string& fileName):** It is a parameterized constructor which initializes empFile to fileName.
- **void home(void):** This function is the home page for employees. It gives employees choice to buy, answer, view profile or log out. A switch-case is used to call the appropriate function as per the employee's selection.
- **void answer(void):** This function enables employees (and admin) to reply to customer complaints. First complaint, an object of Complaint_E, is created. Then the employees are given option to see complaint (anonymously or not), answer complaint, view complaintees, view all complaints, clear all complaints or go back.

Going back takes employees back to home page. Invalid selection results in error message.
If employee wants to see complaint, he or she has to enter username first. complaint.see(readIn, name, true) is then called. If a person with that username has complained, then the complain will be displayed; otherwise error will be displayed. complaint.see(readIn, name, false) is called if employee wants to see the complaint anonymously.
If employee wants to answer complaint, he or she is asked to enter username followed by response. Then complaint.answer(ans, name) is called. If that username ahs no complaint, error is displayed. complaint.view(false) is called to display all complaintees, and complaint.view(true) is called to display all complaints. complaint.clear() is called to clear all existing complaints.

Admin:

It is defined in Admin.h, and its methods are defined in Admin.cpp. It is child class of Employee.


Admin
- adminFile : const string
- empFile : const string
- customerFile : const string
- home() override : void
- add_employee() : void
- view(fileName : const string&) : void
- search(fileName : const string&) : void
- search(str : const string&, fileName : const string&) : void
- Delete(fileName : const string& fileName, user : string&) : void
- balance(fileName : const string&) : void
- accounts() : void
+ "constructor" Admin(AdminFile : const string&, Empfile : const string&, CustomerFile const string&)
+ stock() : void

Attributes

- **const string adminFile:** It stores the name of the admin file, i.e admin.txt. It is initialized by a parameterized constructor.
- **const string empFile:** It stores the name of the employee file, i.e emp.txt. It is initialized by a parameterized constructor.

- **const string customerFile:** It stores the name of the customer file, i.e customer.txt. It is initialized by a parameterized constructor.

Methods

- **Admin(const string& AdminFile, const string& EmployeeFile, const string& CustomerFile):** It is a parameterized constructor which initializes adminFIle to AdminFile, empFile to EmployeeFile, and customerFile to CustomerFile.
- **void home(void):** This function is the home page for admin. It gives admin a myriad of choices. A switch-case is used to call the appropriate function as per the employee's selection.
- **void add_employee(void):** The purpose of this function is to make employee accounts. It calls input(empFile) which does the rest of the work. If the input() function works correctly, it returns true, and message is displayed indicating that the account has been successfully created. If input() returns false, error message is displayed.
- **void view(const string& fileName):** This function displays records of customers or employees, depending upon which file has been passed as an argument to the function. The file is opened in input mode and read. If the file fails to open correctly, an error message is displayed.
- **void search(const string& fileName):** It gives admin the choice to search the file passed as argument on the basis of CNIC, email, phone number, username, or password. It calls the search(const string& key, const string& fileName) function within it which does the actual searching. The file to be searched is passed as second argument and the basis to search on is passed as first argument.
- **void search(const string& key, const string& fileName):** The admin is asked to enter the CNIC/email/phone number/username/password first. Then the file passed as argument is opened in input mode. If the file fails to open correctly, an error message is displayed, and the function exits returning admin to the home page. We check all the records. If a record is found which has the matching CNIC/email/phone number/username/password, then found, a bool set to false, is changed to true, indicating that the record has been found. The first line of the record is stored in lookup. If found is true, profile(fileName, lookup) is called to display the record; otherwise, an error message is displayed.
- **void Delete(const string& fileName, string& user):** This function deletes the record passed as argument. fileName is the file in which the user is searched for, and user is the name of the user whose record is to be deleted. This effectively allows admin to ban

customers and fire employees. Two files are opened for this purpose, the original file in input mode and a new file in output mode. If files fail to open successfully, error message is displayed, and the function ends returning admin to home page. We then copy contents to the new file except for the record which is to be deleted. If no such record exists, error message is displayed; otherwise, a message denoting successful operation is printed to screen. The files are then closed. The old file is deleted, and the new one is renamed to the name of the old file.

- **void accounts(void):** This function displays financial details to admin. “cash.txt” is opened in input mode. Error is displayed if it fails to open; the function then ends, taking admin back to home page. If file opens successfully, the contents of file are displayed on the screen.
- **void balance(const string& fileName):** This function allows admin to manually update balance of customers and employees, depending upon the file passed as argument. The admin is asked to enter the username of the customer/employee whose balance is to be changed, along with the new balance. A do-while inside if statement is used to ensure that the new balance is not negative. Then two files are opened, the original file in input mode and a new file in output mode. If files fail to open successfully, error message is displayed, and the function ends returning admin to home page. We then copy contents to the new file and replace the old balance with the new one. If no such record exists, error message is displayed; otherwise, a message denoting successful operation is printed to screen. The files are then closed. The old file is deleted, and the new one is renamed to the name of the old file.
- **void stock(void):** This functions displays details of goods to admin and allows him/her to make modifications as well as re-order. It is a friend of Goods, so it has access to private attributes of Goods. First, an array of N goods, g, is declared where N is the number of goods defined to be 10. Then, initialize_goods(g) is called to initialize g. If it fails (returns false), error message is displayed, and function ends, returning admin to home page. If it does not, a for loop is run which prints out the details of all the goods. The admin is then given three options – order, modify, and go back. Error message is displayed if admin makes incorrect choice. Going back simply returns admin to home page.

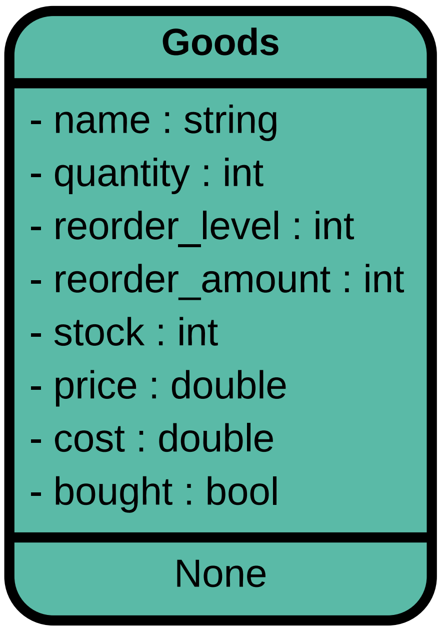
  If admin decides to order or modify, he/she is asked the name of the good he/she wants to order or modify. A for loop is run from 0 to N and the names of goods compared with entered name. If a good with the same name is found, b (a bool set to false) is changed to true, the index of the good is stored for future reference, and the loop breaks. If loop ends with no good found, error message is printed and function ends.

If a matching good is found and admin had selected to order, then first, we declare csh, an object of Cash. It is initialized via initialize_cash(csh). If it fails, error is displayed and function ends. If it does not, admin is asked the quantity to order. This quantity must be positive and must be such that there is enough cash to be able to order. A do-while loop inside an if statement is once again used to validate the quantity. Once a correct quantity is entered, the stock is increased and the cash deducted. We then call update_goods(g) and update_cash(csh) to update "goods.txt" and "cash.txt" respectively. If either function fails, error is printed; otherwise, a message denoting successful ordering is displayed.

If a matching good is found and admin had selected to modify, the admin is asked to enter the new price, new re-order level and the new re-order amount. All these are validated, with the help of a do-while enclosed in an if statement, to ensure that they are positive. Then, update_goods(g) is called to update "goods.txt". If it is successful, a message indicating so is displayed to the screen; otherwise, error is displayed.

Goods:

It is defined in Goods.h. Person is its friend class.



Attributes:

- **string name:** It stores the name of the good.
- **int quantity:** It stores the quantity in which the good has been bought (if it has been bought)

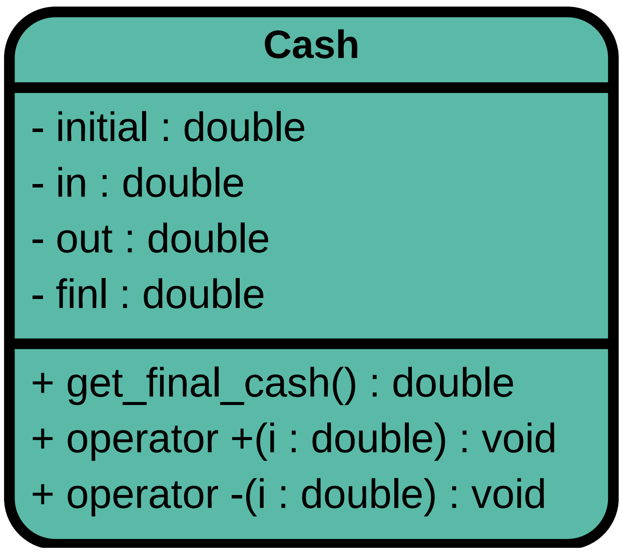
- **int reorder_level:** It stores the level to which goods fall after which a reorder is placed.
- **int reorder_amount:** It stores how many goods are to be reordered.
- **int stock:** It stores the number of items in inventory.
- **double cost:** It stores the per unit cost.
- **double price:** It stores the per unit selling price.
- **bool bought:** It indicates whether the good has been bought or not. It is false if it has not been bought and true otherwise.

Methods

- **friend void Admin::stock(void):** Friend function defined in Admin. NOT a member function.

Cash:

It is defined in Cash.h.


Cash
- initial : double
- in : double
- out : double
- finl : double
+ get_final_cash() : double
+ operator +(i : double) : void
+ operator -(i : double) : void

Attributes:

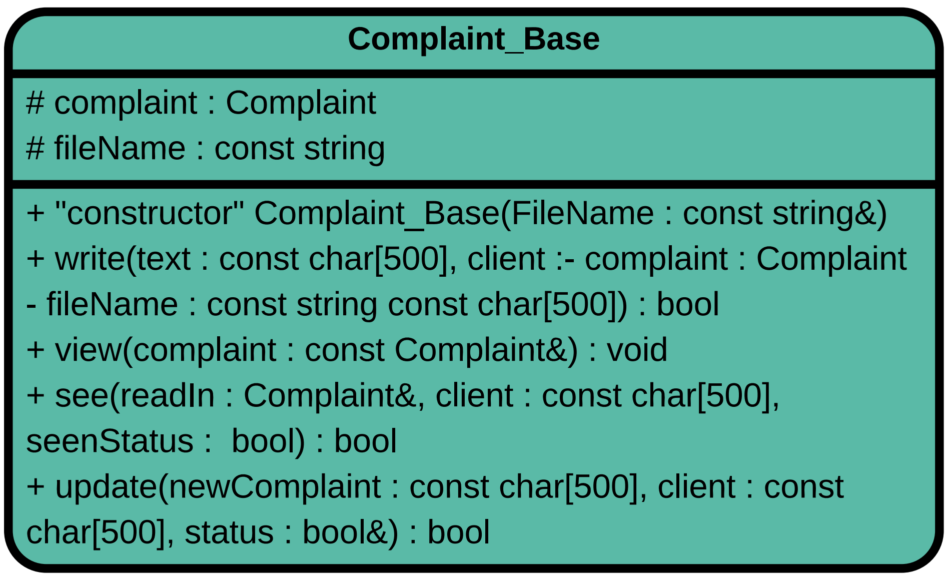
- **double initial:** It stores the initial cash.
- **double in:** It stores cash in.
- **double out:** It stores cash out.
- **double finl:** It stores the current cash (initial + in – out)

Methods

- **double get_final_cash(void):** Returns finl. Inline because of short size.
- **+:** The '+' operator has been overloaded for Cash. It is an operator for cash in. This operator accepts an amount as argument. That amount is added to in and finl.
- **-:** The '-' operator has been overloaded for Cash. It is an operator for cash out. This operator accepts an amount as argument. That amount is added to out and subtracted from finl.

Complaint_Base:

It is defined in Complaint_Base.h



```
Complaint_Base
# complaint : Complaint
# fileName : const string
+ "constructor" Complaint_Base(FileName : const string&)
+ write(text : const char[500], client :- complaint : Complaint
- fileName : const string const char[500]) : bool
+ view(complaint : const Complaint&) : void
+ see(readIn : Complaint&, client : const char[500],
seenStatus :  bool) : bool
+ update(newComplaint : const char[500], client : const
char[500], status : bool&) : bool
```

Attributes:

- **Complaint complaint:** Complaint is a struct, defined in Complaint_Base.h. It has five attributes – char text[500] which stores the complain, char answer[500] which stores the answer to the complaint, char client[500] which stores the username of the customer who made the complaint, bool seen which is false if complaint is yet to be seen by an employee (or admin) and is true if it has been seen, and lastly, bool answered, which is false if complain is yet to be answered and true otherwise. complaint is a variable of this struct.
- **const string fileName:** It stores the name of the file which stores complaints, i.e “complaint.dat”. It is initialized by a parameterized constructor.

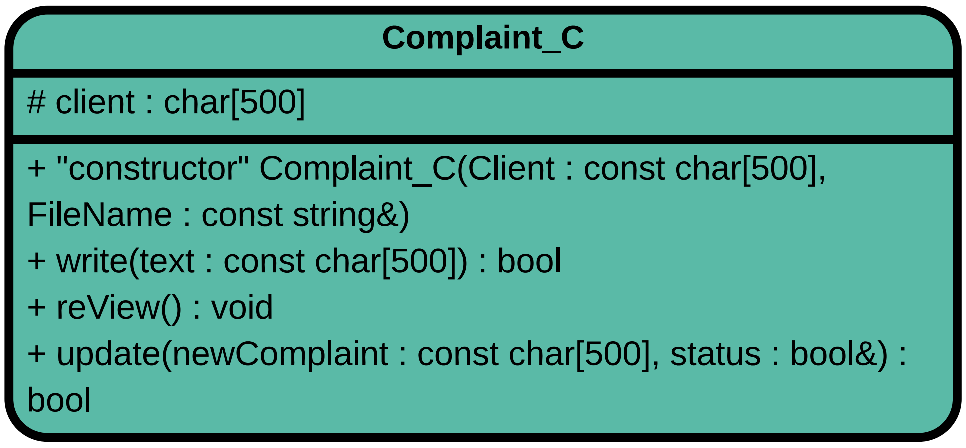
Methods

- **Complaint_Base(const string& FileName):** It is a parameterized constructor which initializes fileName to FileName.
- **bool write(const char text[500], const char client[500]):** It writes complains to the file. text stores the complain, and client stores the username of the customer who complained. The file is opened in binary mode, and its size stored in size. A bool called unique is initialized to false; its purpose is to ensure that a customer can have only one complaint at any given time. We go through the file reading complaints. If any complaint has the same client as the client who is currently complaining, unique becomes false and the person’s complaint is not registered. If unique is true, the complaint is appended to the file. The function ends by returning unique, which shows whether the complaint has been written or not.

- **void view(const Complaint& complaint):** This function displays the complaint passed to it as an argument.
- **bool see(Complaint& readIn, const char client[500], bool seenStatus):** client stores the username of the client whose complaint the user wants to see. seenStatus is true for employees and admin (provided that they are not seeing anonymously) and false for customers. We open “complaint.dat” and store its size in size. We go through the entire file reading complaints until we find a complaint with the same client as the one passed as argument (the one whose complaint we want to see). When the complaint is found, found (a bool initialized to false) becomes true. If seenStatus is true (as it is for employees and admin unless they go anonymous), then the seen attribute of the complaint is changed to true indicating that the complaint has been seen. After changing the seen attribute, the complaint is re-written to file so that this change gets stored. The function ends by returning found.
- **bool update(const char newComplaint[500], const char client[500], bool& status):** This function allows the complaint to be updated. newComplaint stores the new complain, client stores the username of the customer whose complain is to be updated, and status is a reference to a bool which is set to false. First, we open “complaint.dat” and store its size in size. We go through the entire file reading complaints until we find a complaint with the same client as the one passed as argument (the one whose complaint we want to update). We first check whether the complaint has been answered or not. If it has, then we do not update the complaint and simply change status to true. If it has not, we update the complaint and re-write it to the file to store the changes. Along with this, we set found (a bool initialized to false) to true to indicate that the job has been done. The function ends by returning found.

Complaint_C:

It is defined in Complaint_C.h and is derived from Complaint_Base.



Complaint_C
# client : char[500]
+ "constructor" Complaint_C(Client : const char[500], FileName : const string&)
+ write(text : const char[500]) : bool
+ reView() : void
+ update(newComplaint : const char[500], status : bool&) : bool

Attributes:

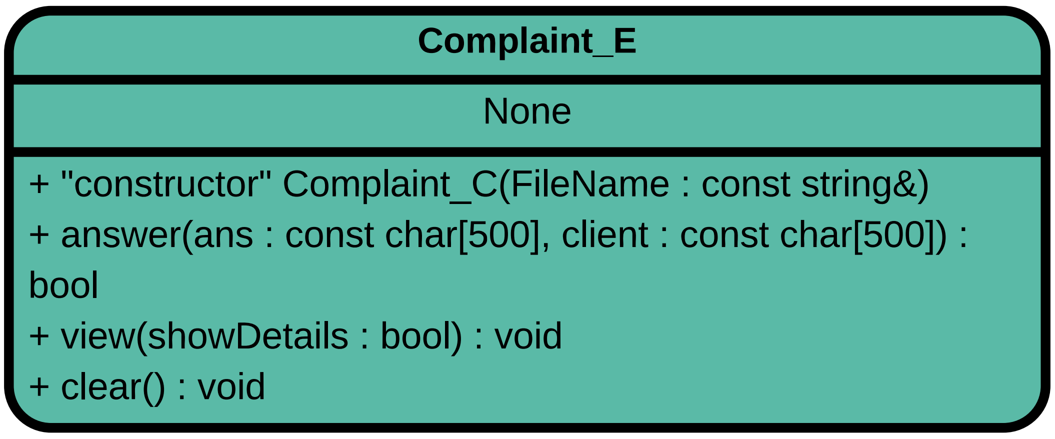
- **char client[500]:** It stores the username of the complaining customer. It is initialized by a parameterized constructor.

Methods

- **Complaint_C(const char Client[500], const string& FileName):** It is a parameterized constructor which initializes client to Client. FileName is passed as argument to Complaint_Base's constructor.
- **bool write(const char text[500]):** It writes complaint by summoning Complaint_Base::write(text, this->client). Here text, stores the complaint to be written. The bool returned by Complaint_Base::write(text, this->client) is also returned by this function.
- **void reView(void):** It calls Complaint_Base::see(complaint, this->client, false). If it returns true, Complaint_Base::view(complaint) is called to display the complaint; otherwise, error message is displayed.
- **bool update(const char newComplaint[500], bool& status):** Its purpose is to update complaints. For this purpose, it calls Complaint_Base::update(newComplaint, this->client, status). The bool returned by Complaint_Base::update(newComplaint, this->client, status) is also returned by this function.

Complaint_E:

It is defined in Complaint_E.h and is derived from Complaint_Base.



Complaint_E

None

+ "constructor" Complaint_C(FileName : const string&)
+ answer(ans : const char[500], client : const char[500]) : bool
+ view(showDetails : bool) : void
+ clear() : void

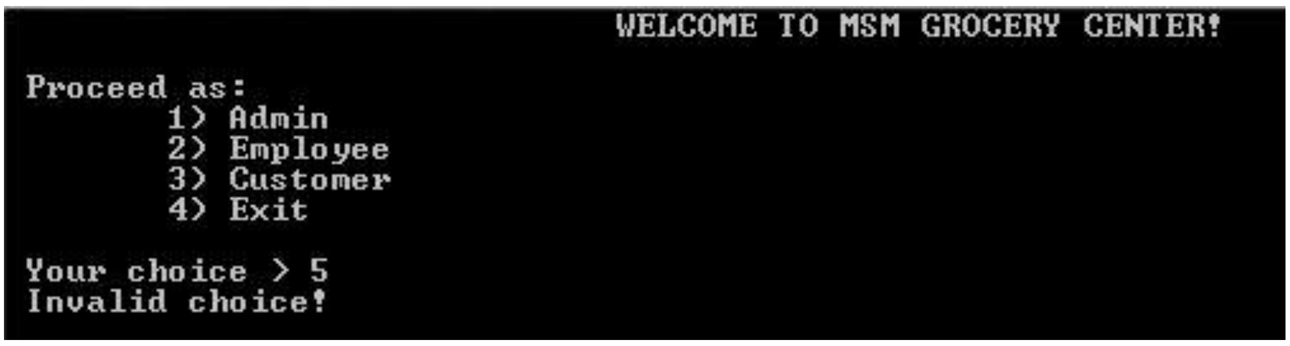
Methods

- **Complaint_E(const string& FileName):** It is a parameterized constructor which passes FileName as an argument to Complaint_Base's constructor.
- **void clear(void):** This function clears all the complaints in "complaint.dat" by opening the file in trunc mode.

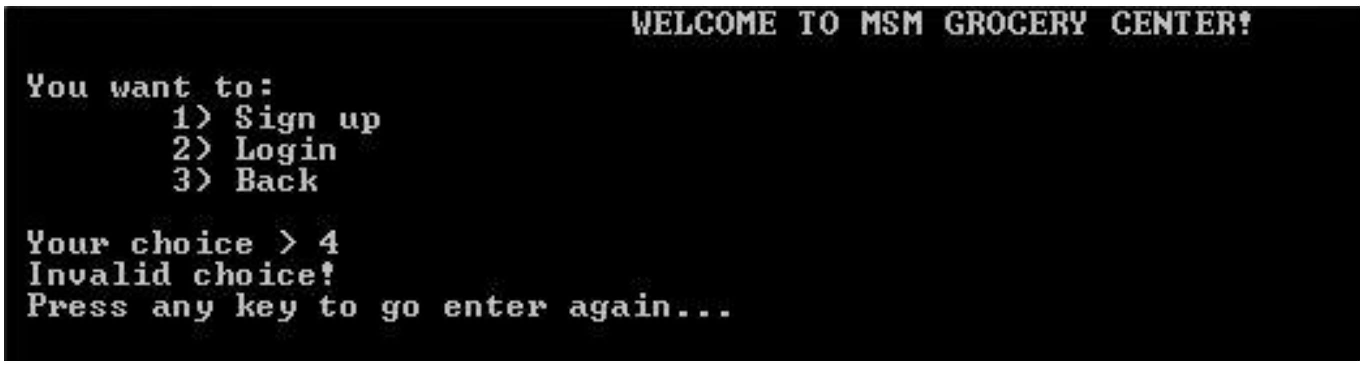
- **bool answer(const char ans[500], const char client[500]):** This function enables employees and admin to answer complaints. ans stores the reply whereas client stores the username of the customer whose complaint is to be addressed. "complaint.dat" is first opened and its size stored in size. We then go through the entire file reading complaints until we find a complaint whose client matches the client passed as argument (the one we want to answer). Upon this, the seen and answered attributes are changed to true (to indicate that the complaint has been seen and answered respectively) and the reply is written to answer attribute. The complaint is re-written to the file to store the changes. found, a bool initialized to false, is set to true. The function ends by returning found.
- **void view(bool showDetails):** It displays all complaints if showDetails is true; otherwise, it only displays the names of complaintees. "complaint.dat" is opened and its size stored in size. We go through the entire file reading complaints. If showDetails is true, Complaint_Base::view(complaint) is called to display the complain; otherwise, only the client attribute is printed to the screen.

Working of Project:

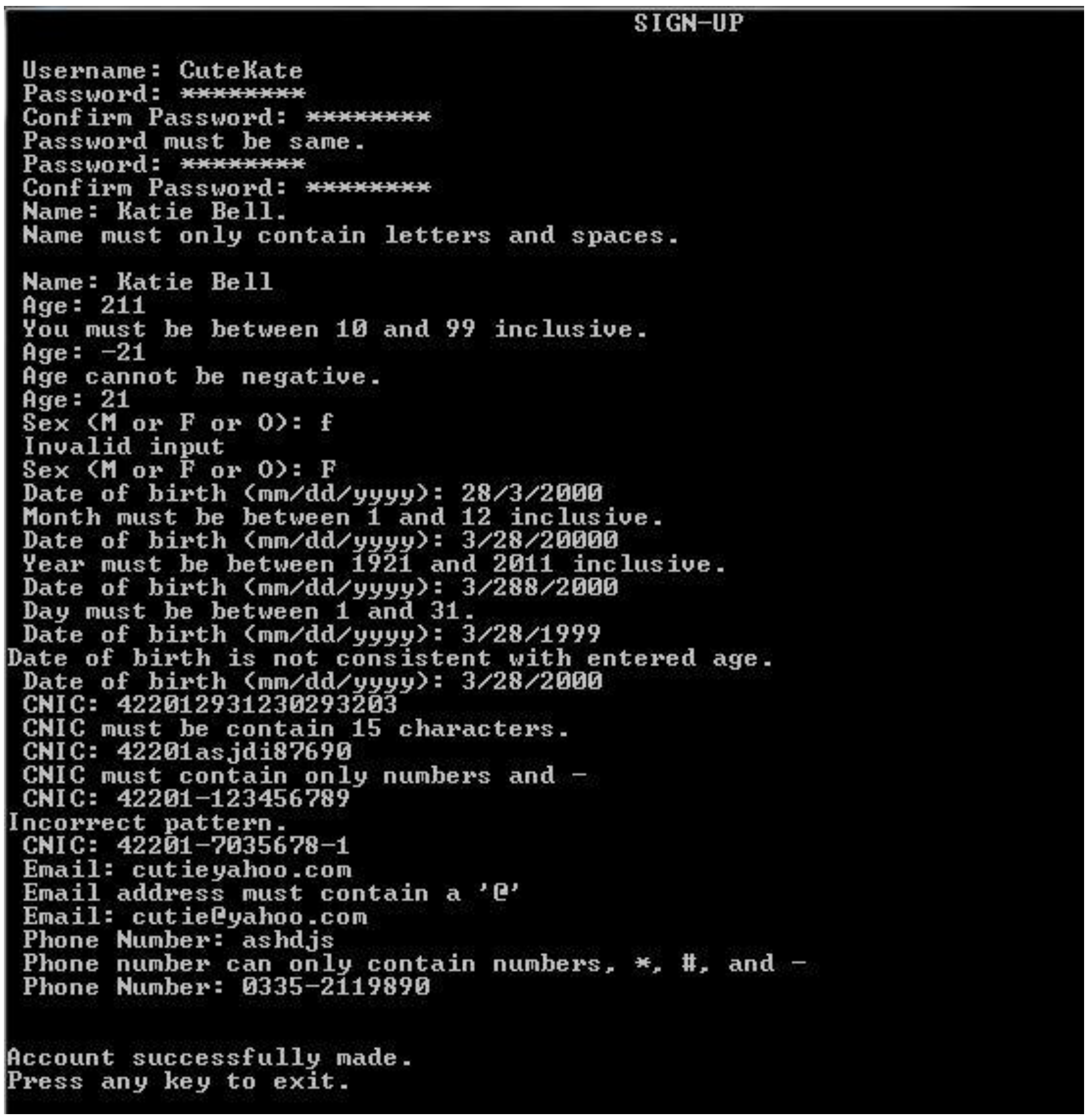
These screenshots show how the project will look like:



```
                                        WELCOME TO MSM GROCERY CENTER!

Proceed as:
         1) Admin
         2) Employee
         3) Customer
         4) Exit

Your choice > 5
Invalid choice!
```

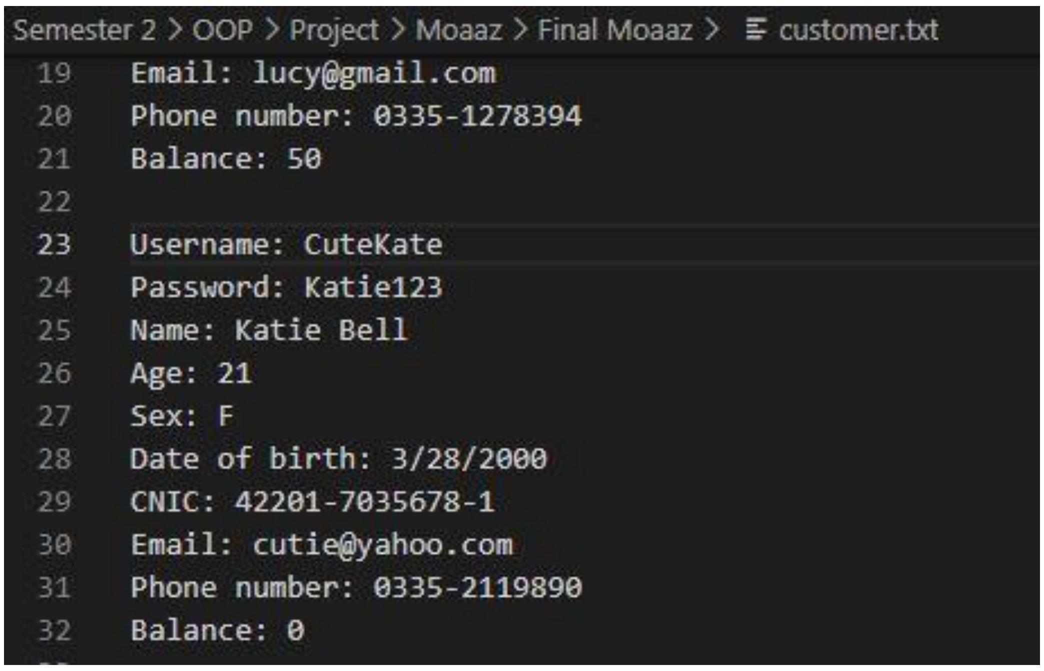


```
                                        WELCOME TO MSM GROCERY CENTER!

You want to:
         1) Sign up
         2) Login
         3) Back

Your choice > 4
Invalid choice!
Press any key to go enter again...
```



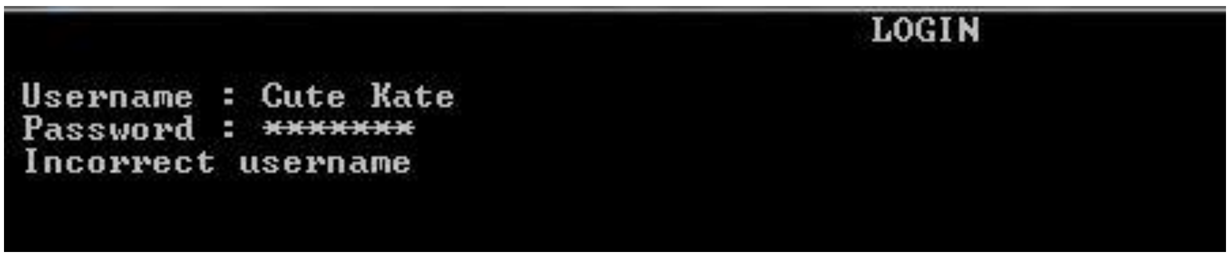
```
                                SIGN-UP
 Username: CuteKate
 Password: ********
 Confirm Password: ********
 Password must be same.
 Password: ********
 Confirm Password: ********
 Name: Katie Bell.
 Name must only contain letters and spaces.

 Name: Katie Bell
 Age: 211
 You must be between 10 and 99 inclusive.
 Age: -21
 Age cannot be negative.
 Age: 21
 Sex (M or F or O): f
 Invalid input
 Sex (M or F or O): F
 Date of birth (mm/dd/yyyy): 28/3/2000
 Month must be between 1 and 12 inclusive.
 Date of birth (mm/dd/yyyy): 3/28/20000
 Year must be between 1921 and 2011 inclusive.
 Date of birth (mm/dd/yyyy): 3/288/2000
 Day must be between 1 and 31.
 Date of birth (mm/dd/yyyy): 3/28/1999
Date of birth is not consistent with entered age.
 Date of birth (mm/dd/yyyy): 3/28/2000
 CNIC: 42201293123029320З
 CNIC must be contain 15 characters.
 CNIC: 42201asjdi87690
 CNIC must contain only numbers and -
 CNIC: 42201-123456789
Incorrect pattern.
 CNIC: 42201-7035678-1
 Email: cutieyahoo.com
 Email address must contain a '@'
 Email: cutie@yahoo.com
 Phone Number: ashdjs
 Phone number can only contain numbers, *, #, and -
 Phone Number: 0335-2119890


Account successfully made.
Press any key to exit.
```

The record gets added to file:

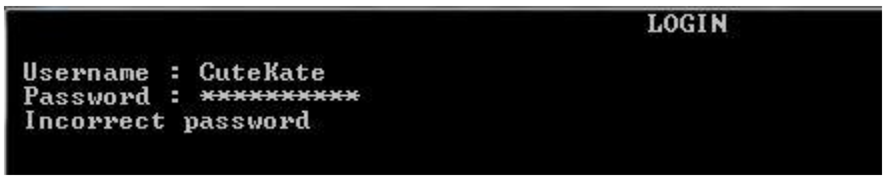


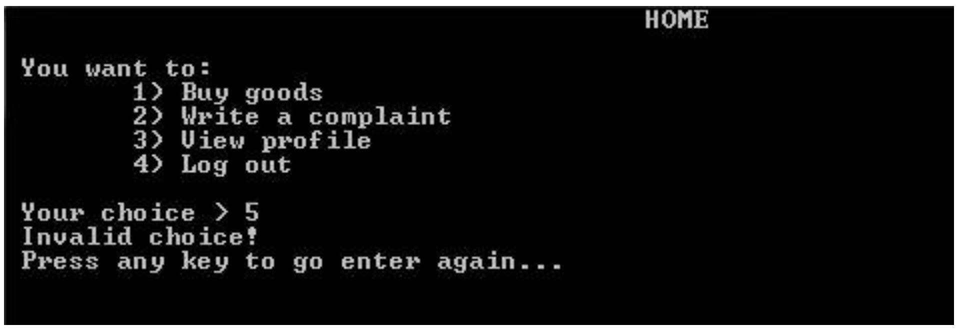
Semester 2 > OOP > Project > Moaaz > Final Moaaz > customer.txt

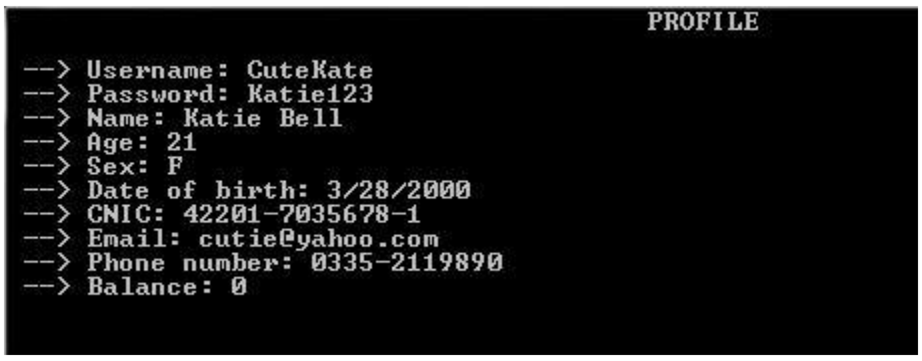
```
Email: lucy@gmail.com
Phone number: 0335-1278394
Balance: 50

Username: CuteKate
Password: Katie123
Name: Katie Bell
Age: 21
Sex: F
Date of birth: 3/28/2000
CNIC: 42201-7035678-1
Email: cutie@yahoo.com
Phone number: 0335-2119890
Balance: 0
```

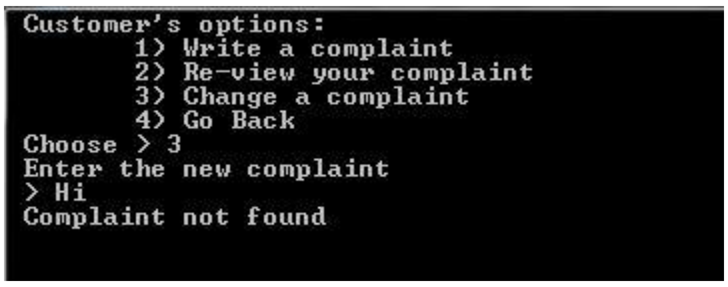


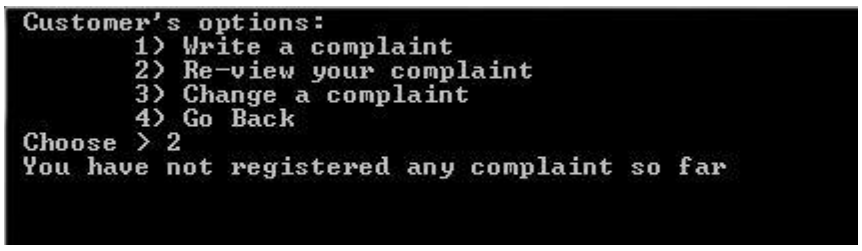
```
                                LOGIN
Username : Cute Kate
Password : ********
Incorrect username
```

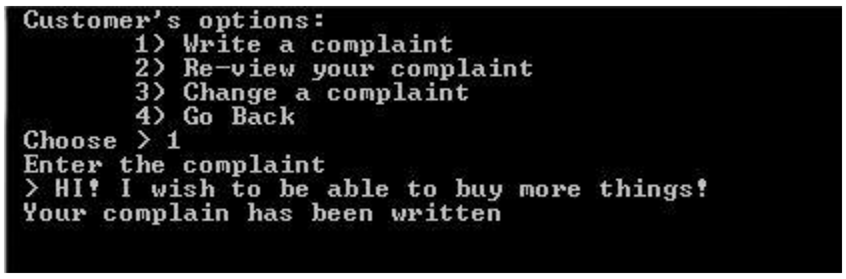

```
                                                                 LOGIN

 Username : CuteKate
 Password : **********
 Incorrect password
```

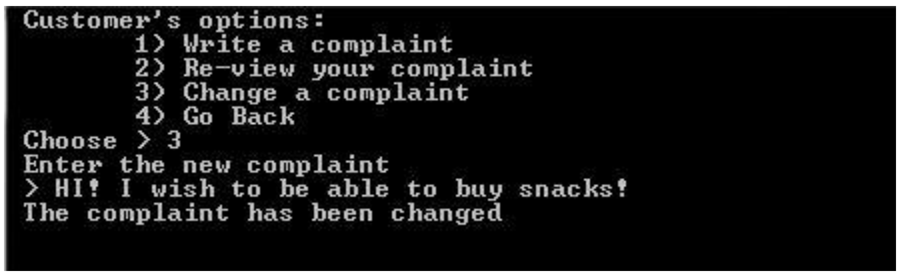

```
                                                                 HOME

You want to:
        1) Buy goods
        2) Write a complaint
        3) View profile
        4) Log out

Your choice > 5
Invalid choice!
Press any key to go enter again...
```

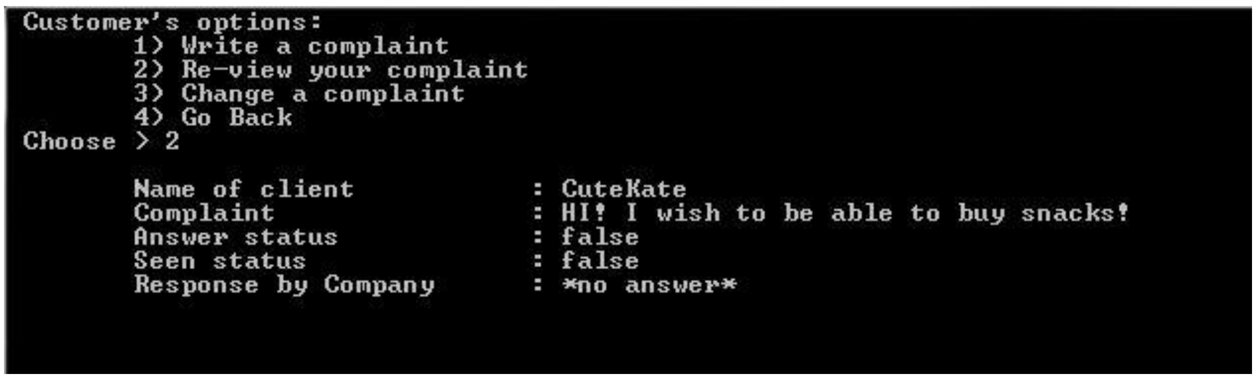

```
                                                                 PROFILE

 --> Username: CuteKate
 --> Password: Katie123
 --> Name: Katie Bell
 --> Age: 21
 --> Sex: F
 --> Date of birth: 3/28/2000
 --> CNIC: 42201-7035678-1
 --> Email: cutie@yahoo.com
 --> Phone number: 0335-2119890
 --> Balance: 0
```


```
 Customer's options:
        1) Write a complaint
        2) Re-view your complaint
        3) Change a complaint
        4) Go Back
 Choose > 3
 Enter the new complaint
 > Hi
 Complaint not found
```

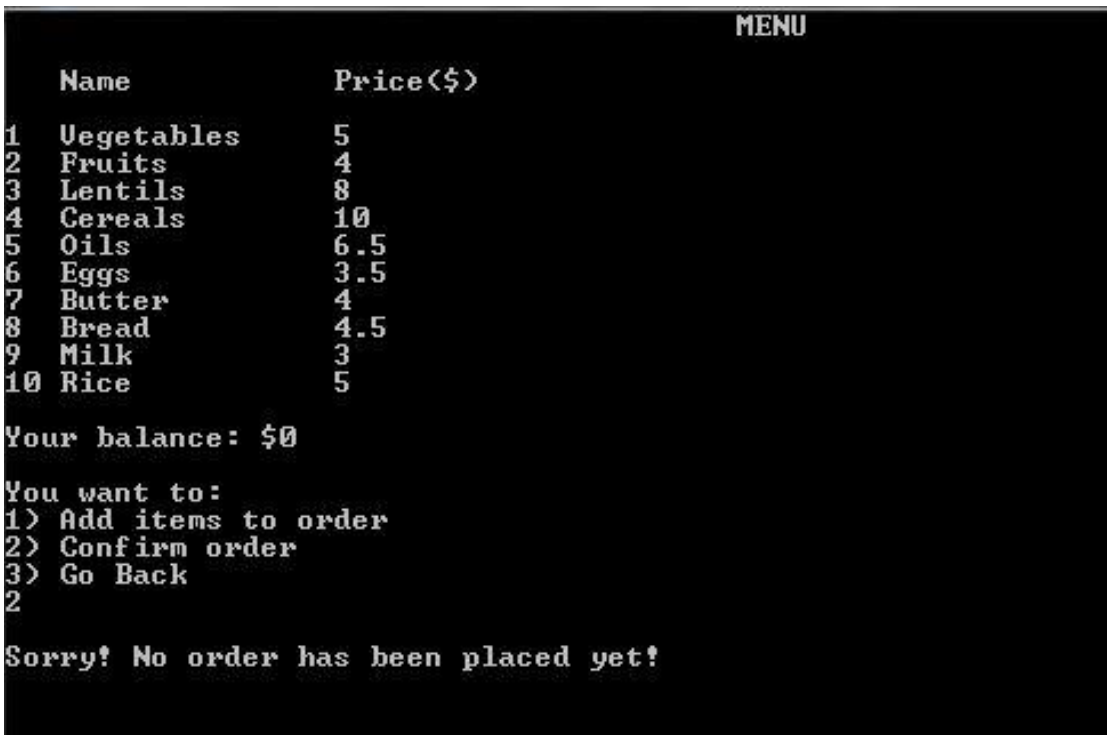

```
 Customer's options:
        1) Write a complaint
        2) Re-view your complaint
        3) Change a complaint
        4) Go Back
 Choose > 2
 You have not registered any complaint so far
```


```
 Customer's options:
        1) Write a complaint
        2) Re-view your complaint
        3) Change a complaint
        4) Go Back
 Choose > 1
 Enter the complaint
 > HI! I wish to be able to buy more things!
 Your complain has been written
```

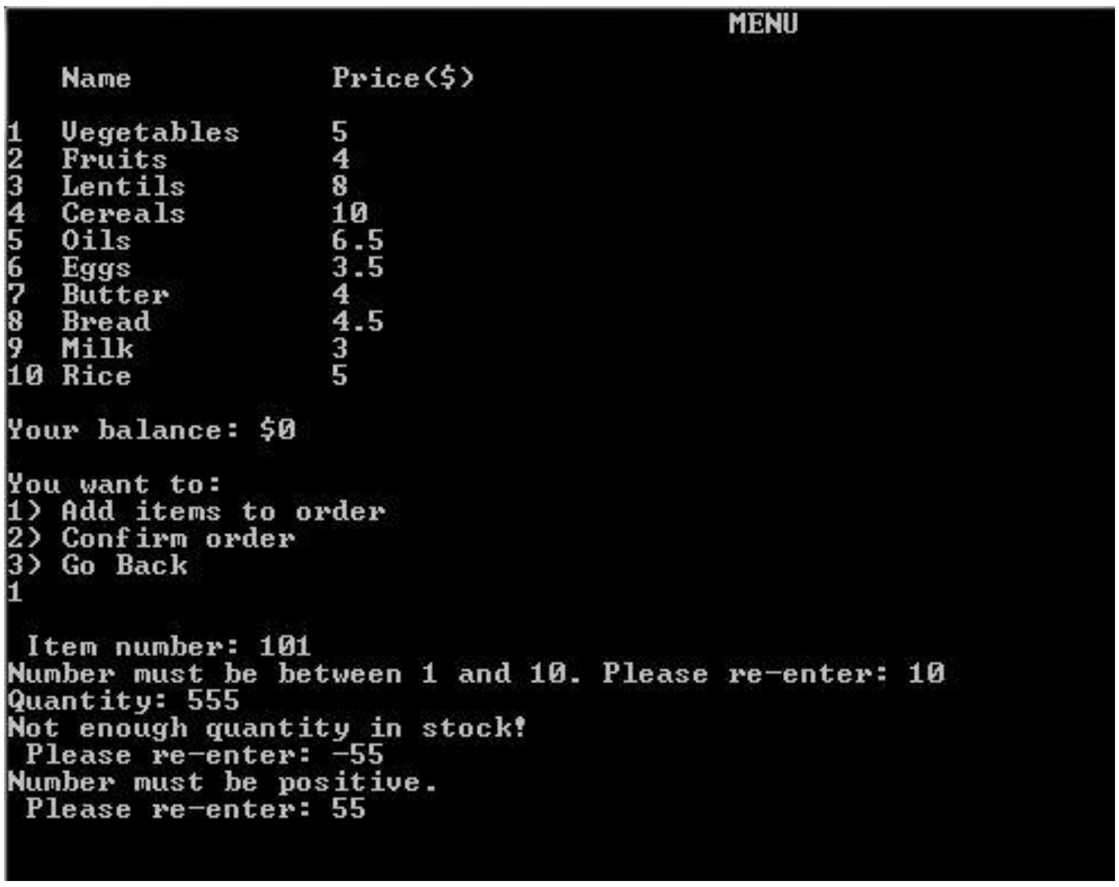

```
 Customer's options:
        1) Write a complaint
        2) Re-view your complaint
        3) Change a complaint
        4) Go Back
 Choose > 3
 Enter the new complaint
 > HI! I wish to be able to buy snacks!
 The complaint has been changed
```



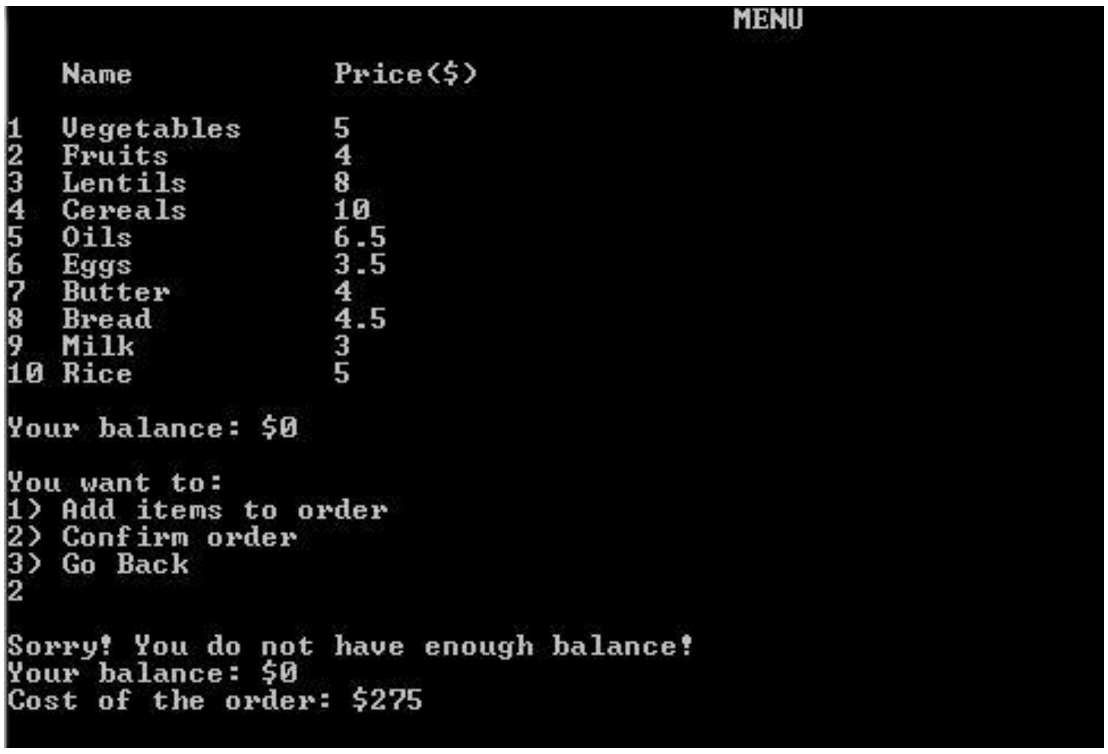
```
Customer's options:
       1) Write a complaint
       2) Re-view your complaint
       3) Change a complaint
       4) Go Back
Choose > 2

       Name of client             : CuteKate
       Complaint                  : HI! I wish to be able to buy snacks!
       Answer status              : false
       Seen status                : false
       Response by Company        : *no answer*
```



```
                                       MENU

   Name           Price($)

1  Vegetables     5
2  Fruits         4
3  Lentils        8
4  Cereals        10
5  Oils           6.5
6  Eggs           3.5
7  Butter         4
8  Bread          4.5
9  Milk           3
10 Rice           5

Your balance: $0

You want to:
1) Add items to order
2) Confirm order
3) Go Back
2

Sorry! No order has been placed yet!
```

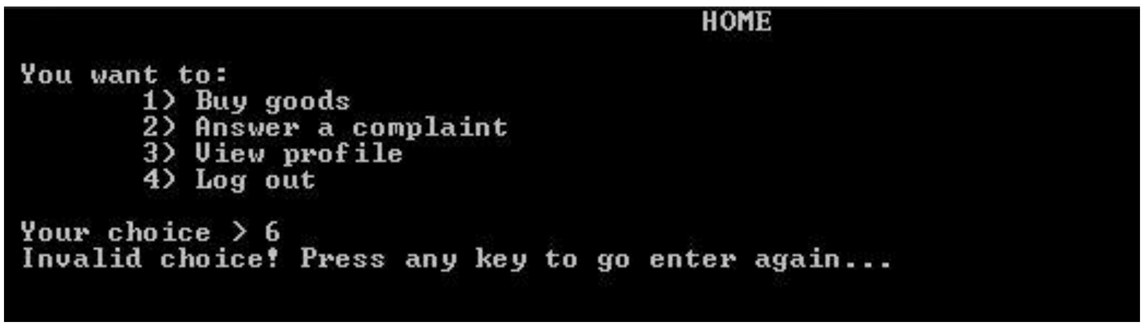


```
                                       MENU

   Name           Price($)

1  Vegetables     5
2  Fruits         4
3  Lentils        8
4  Cereals        10
5  Oils           6.5
6  Eggs           3.5
7  Butter         4
8  Bread          4.5
9  Milk           3
10 Rice           5

Your balance: $0

You want to:
1) Add items to order
2) Confirm order
3) Go Back
1

 Item number: 101
Number must be between 1 and 10. Please re-enter: 10
Quantity: 555
Not enough quantity in stock!
 Please re-enter: -55
Number must be positive.
 Please re-enter: 55
```

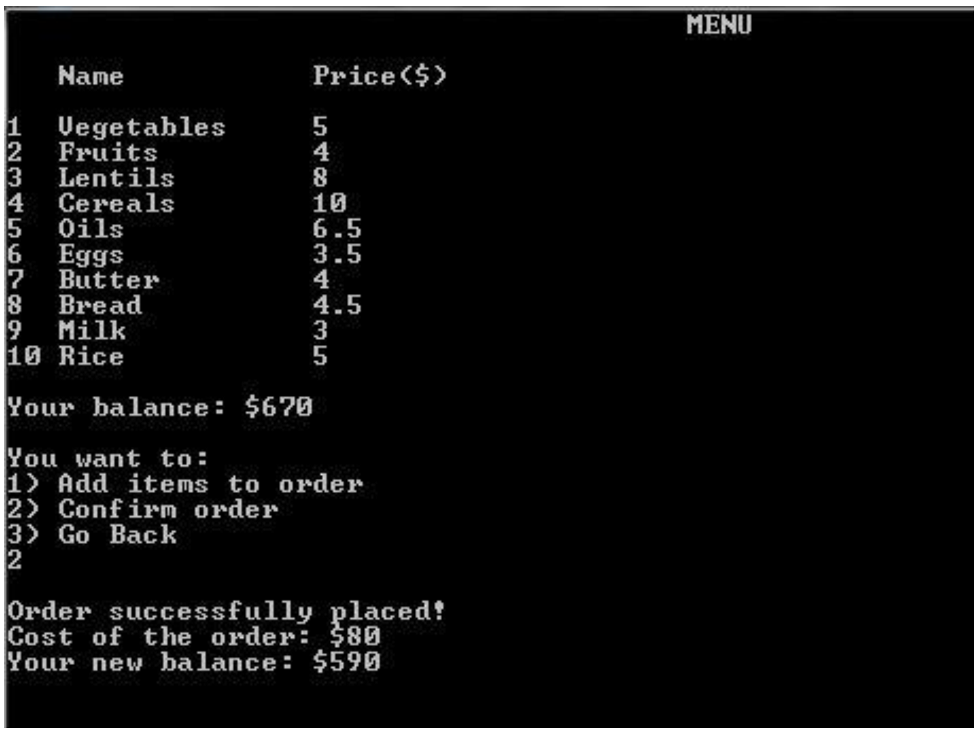


```
                                        MENU

   Name           Price($)

1  Vegetables     5
2  Fruits         4
3  Lentils        8
4  Cereals        10
5  Oils           6.5
6  Eggs           3.5
7  Butter         4
8  Bread          4.5
9  Milk           3
10 Rice           5

Your balance: $0

You want to:
1) Add items to order
2) Confirm order
3) Go Back
2

Sorry! You do not have enough balance!
Your balance: $0
Cost of the order: $275
```

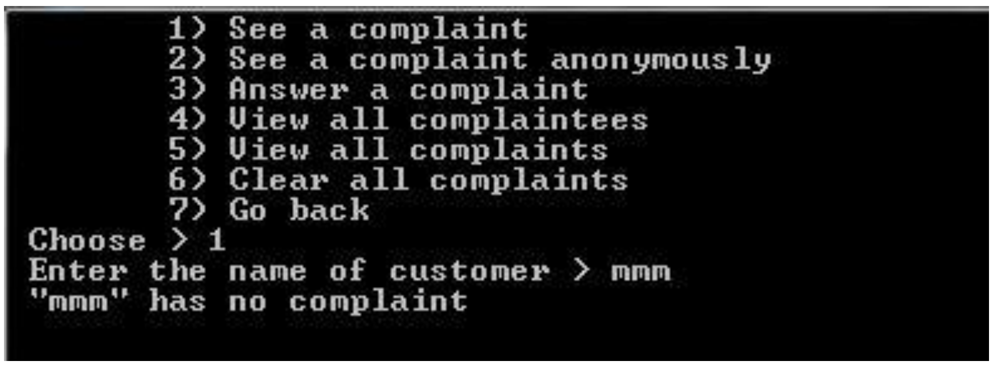
Logging in as employee:



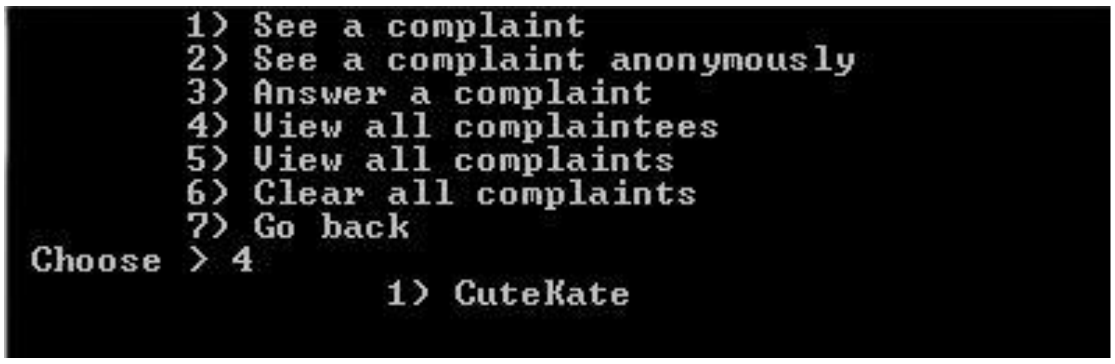
```
                                       HOME

You want to:
       1) Buy goods
       2) Answer a complaint
       3) View profile
       4) Log out

Your choice > 6
Invalid choice! Press any key to go enter again...
```



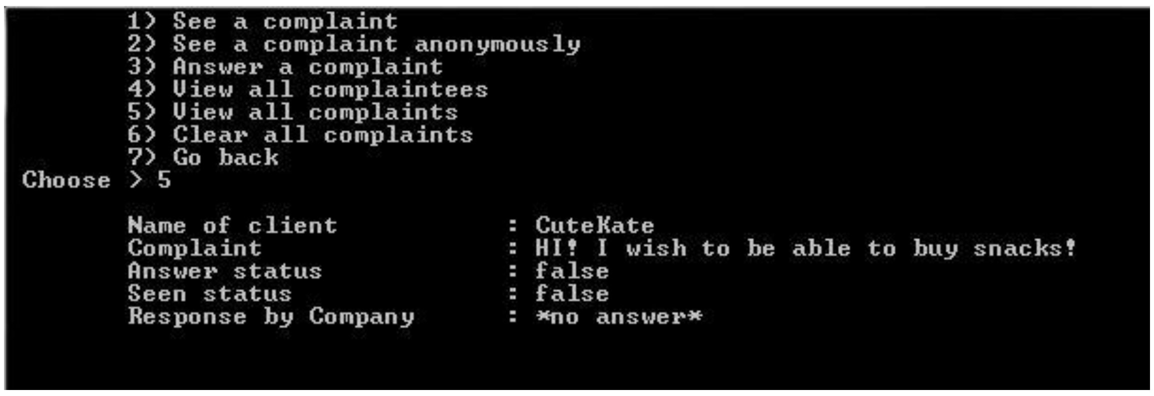
```
                                        MENU

   Name           Price($)

1  Vegetables     5
2  Fruits         4
3  Lentils        8
4  Cereals        10
5  Oils           6.5
6  Eggs           3.5
7  Butter         4
8  Bread          4.5
9  Milk           3
10 Rice           5

Your balance: $670

You want to:
1) Add items to order
2) Confirm order
3) Go Back
2

Order successfully placed!
Cost of the order: $80
Your new balance: $590
```



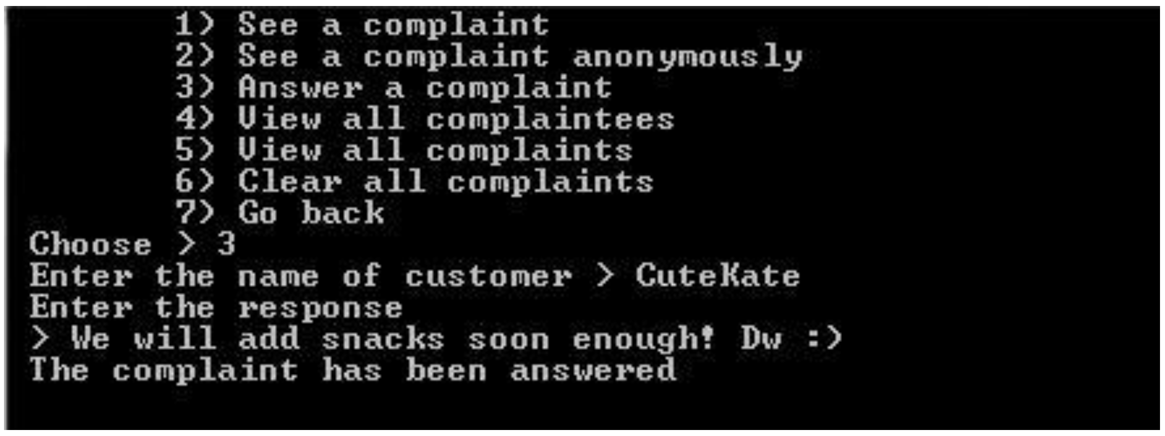
```
         1) See a complaint
         2) See a complaint anonymously
         3) Answer a complaint
         4) View all complaintees
         5) View all complaints
         6) Clear all complaints
         7) Go back
 Choose > 1
 Enter the name of customer > mmm
 "mmm" has no complaint
```



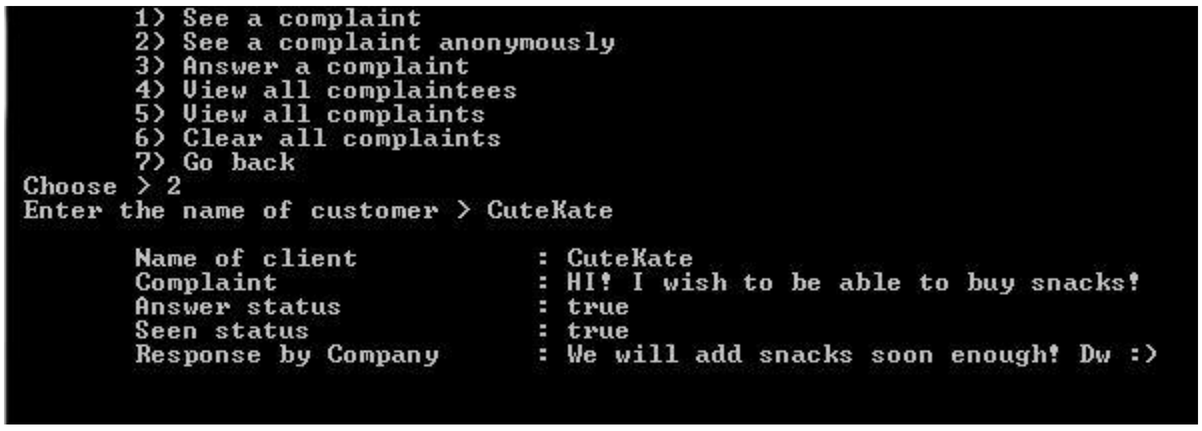
```
        1) See a complaint
        2) See a complaint anonymously
        3) Answer a complaint
        4) View all complaintees
        5) View all complaints
        6) Clear all complaints
        7) Go back
 Choose > 4
                 1) CuteKate
```



```
        1) See a complaint
        2) See a complaint anonymously
        3) Answer a complaint
        4) View all complaintees
        5) View all complaints
        6) Clear all complaints
        7) Go back
 Choose > 5

        Name of client            : CuteKate
        Complaint                 : HI! I wish to be able to buy snacks!
        Answer status             : false
        Seen status               : false
        Response by Company       : *no answer*
```

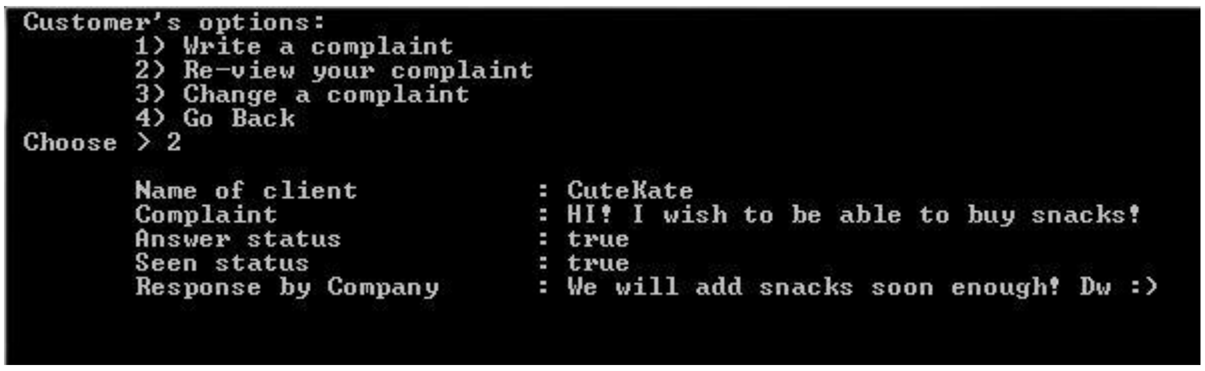


```
        1) See a complaint
        2) See a complaint anonymously
        3) Answer a complaint
        4) View all complaintees
        5) View all complaints
        6) Clear all complaints
        7) Go back
 Choose > 3
 Enter the name of customer > CuteKate
 Enter the response
 > We will add snacks soon enough! Dw :)
 The complaint has been answered
```



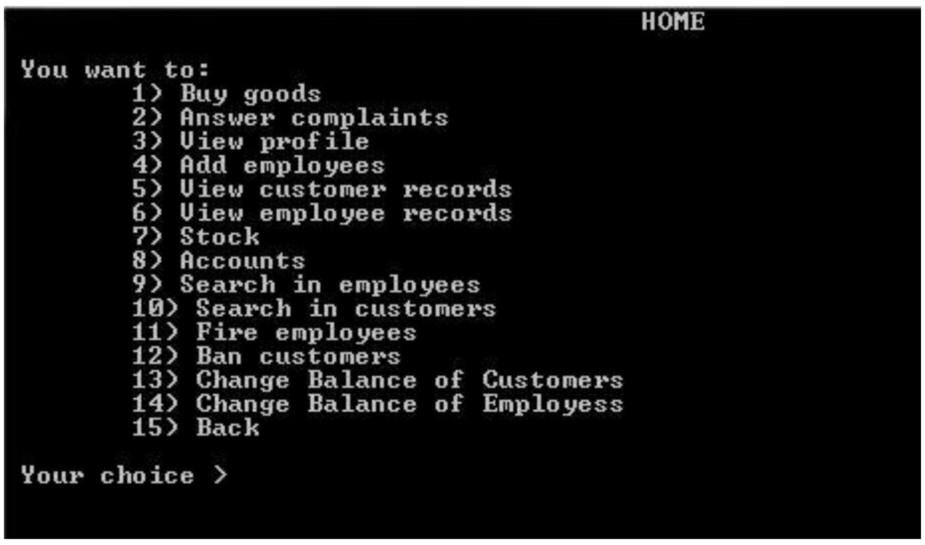
```
        1) See a complaint
        2) See a complaint anonymously
        3) Answer a complaint
        4) View all complaintees
        5) View all complaints
        6) Clear all complaints
        7) Go back
 Choose > 2
 Enter the name of customer > CuteKate

        Name of client            : CuteKate
        Complaint                 : HI! I wish to be able to buy snacks!
        Answer status             : true
        Seen status               : true
        Response by Company       : We will add snacks soon enough! Dw :)
```

On customer's end:

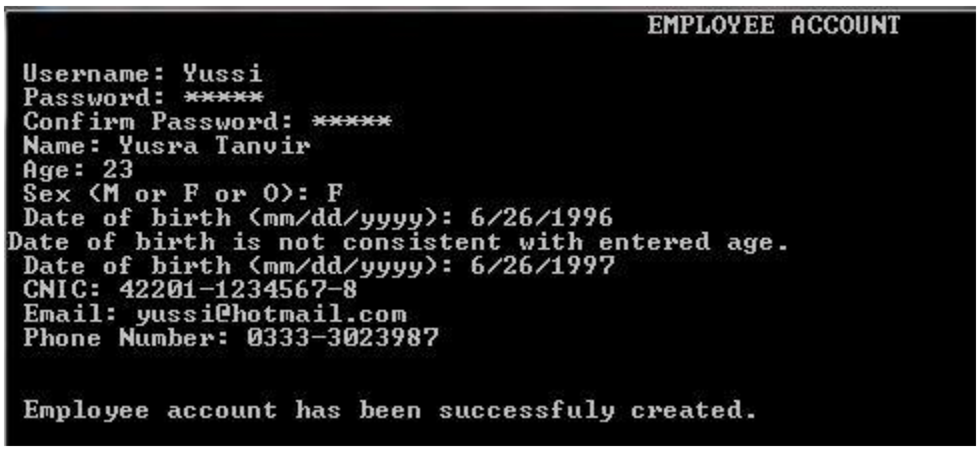


```
Customer's options:
        1) Write a complaint
        2) Re-view your complaint
        3) Change a complaint
        4) Go Back
Choose > 2

        Name of client            : CuteKate
        Complaint                 : HI! I wish to be able to buy snacks!
        Answer status             : true
        Seen status               : true
        Response by Company       : We will add snacks soon enough! Dw :)
```

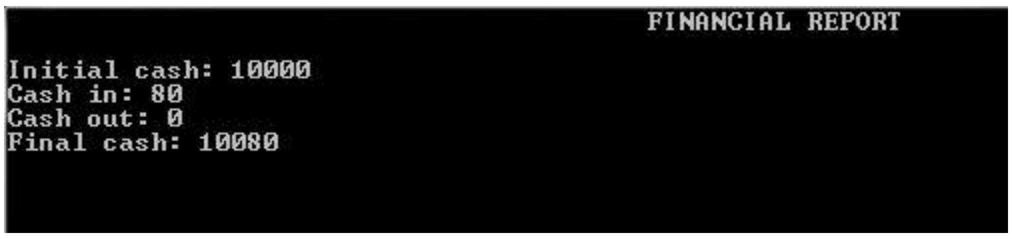
Logging in as admin:



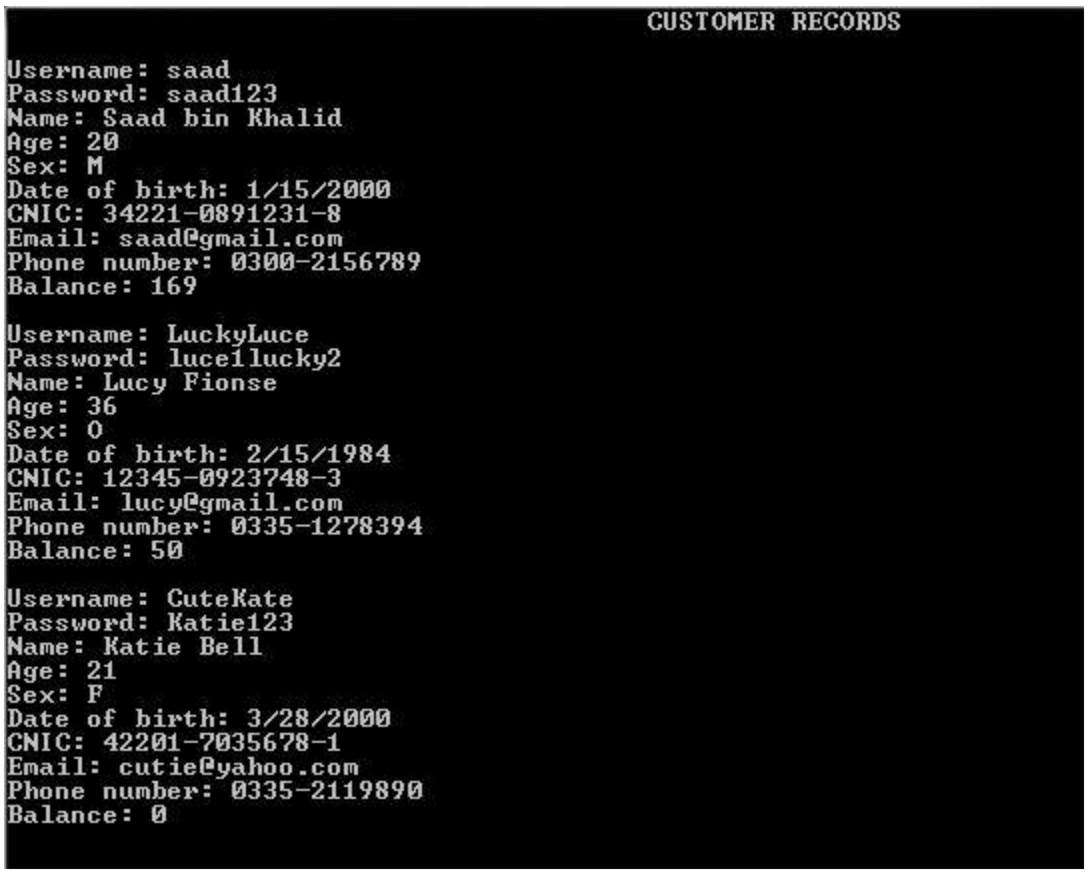
```
                                                HOME

You want to:
       1) Buy goods
       2) Answer complaints
       3) View profile
       4) Add employees
       5) View customer records
       6) View employee records
       7) Stock
       8) Accounts
       9) Search in employees
       10) Search in customers
       11) Fire employees
       12) Ban customers
       13) Change Balance of Customers
       14) Change Balance of Employess
       15) Back

Your choice >
```



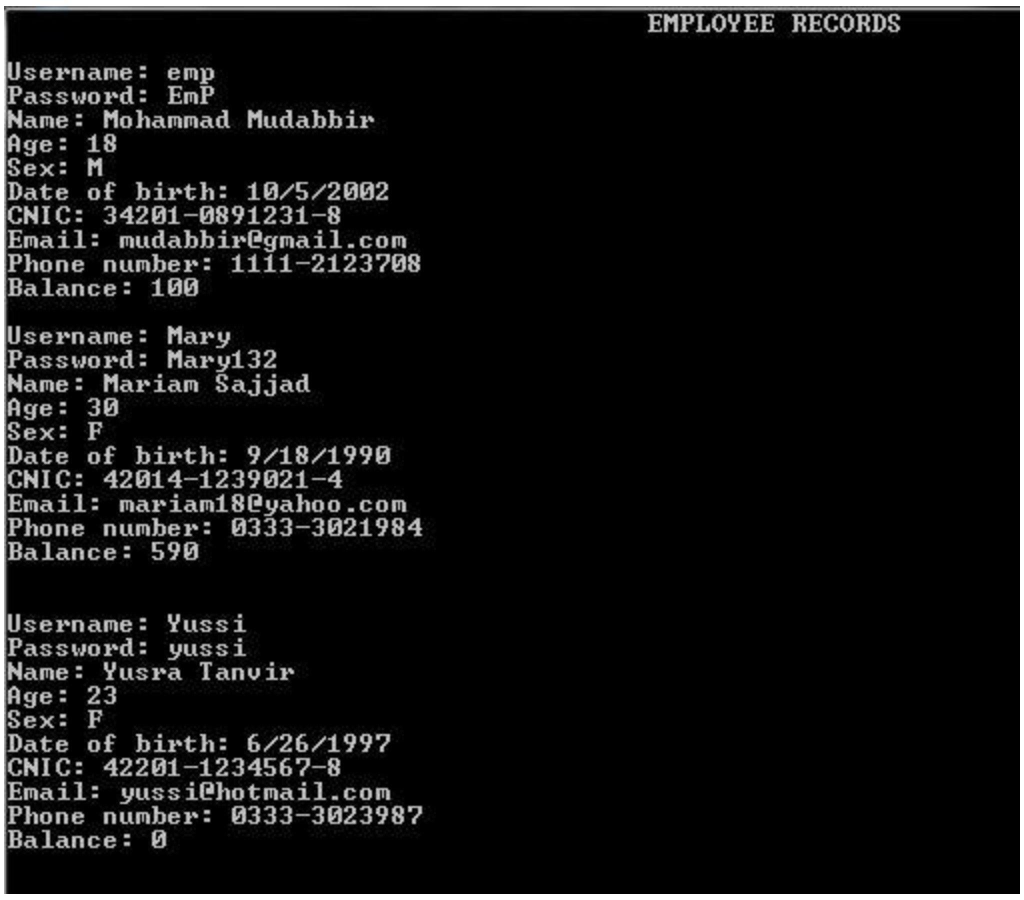
```
                                                EMPLOYEE ACCOUNT

 Username: Yussi
 Password: *****
 Confirm Password: *****
 Name: Yusra Tanvir
 Age: 23
 Sex (M or F or O): F
 Date of birth (mm/dd/yyyy): 6/26/1996
Date of birth is not consistent with entered age.
 Date of birth (mm/dd/yyyy): 6/26/1997
 CNIC: 42201-1234567-8
 Email: yussi@hotmail.com
 Phone Number: 0333-3023987


 Employee account has been successfuly created.
```



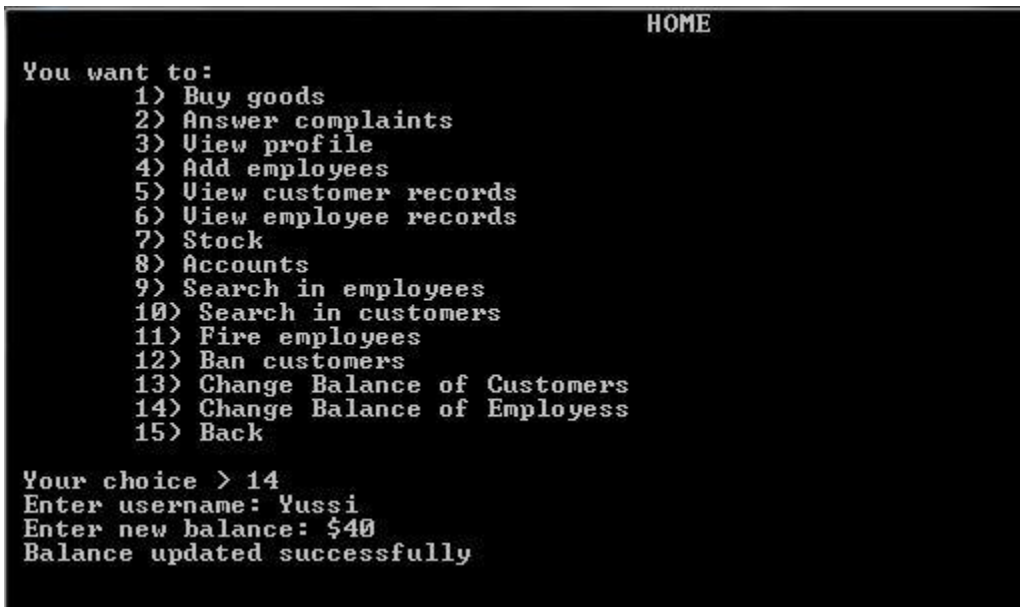
```
                                                FINANCIAL REPORT

Initial cash: 10000
Cash in: 80
Cash out: 0
Final cash: 10080
```



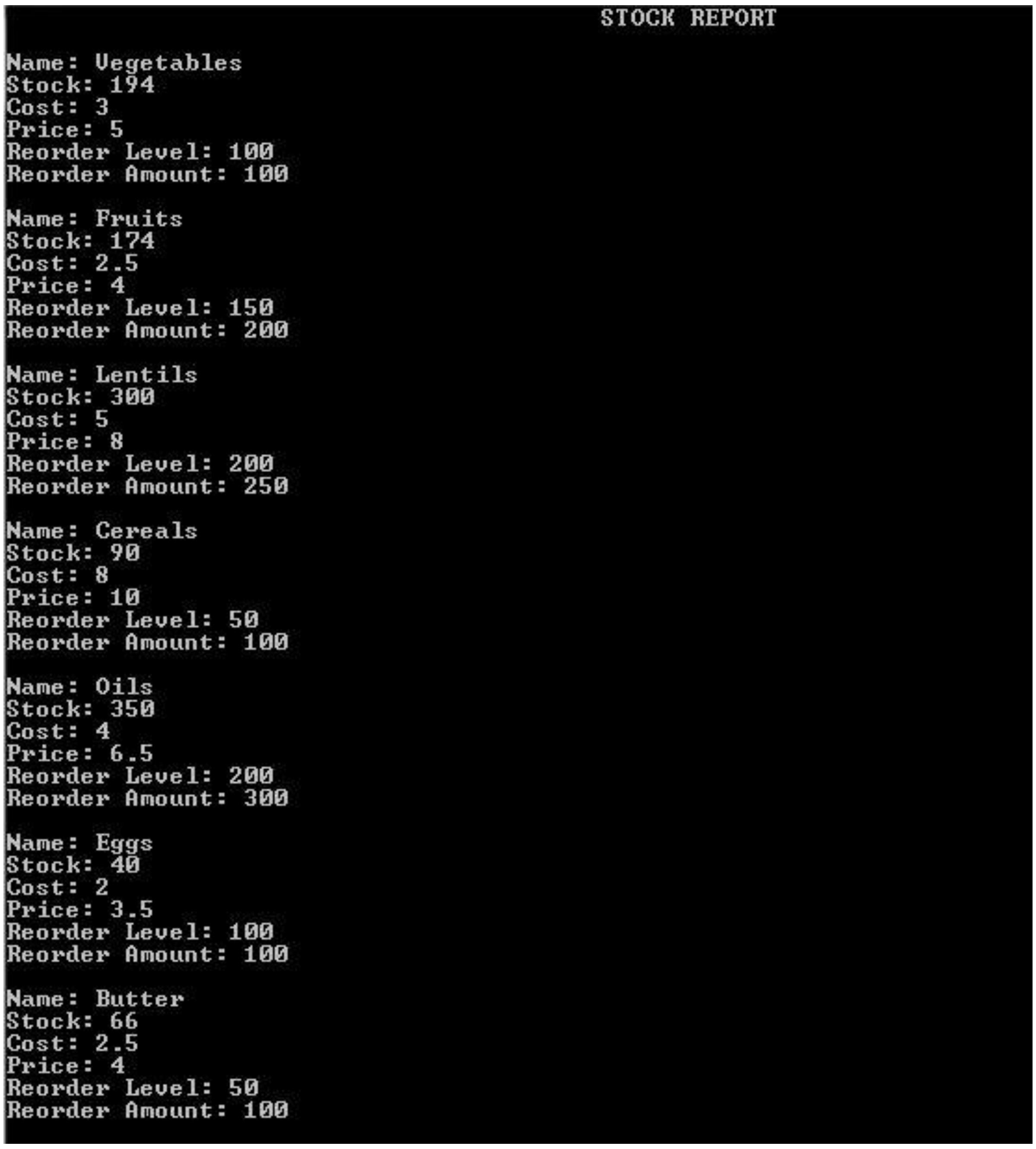
```
                                                CUSTOMER RECORDS

Username: saad
Password: saad123
Name: Saad bin Khalid
Age: 20
Sex: M
Date of birth: 1/15/2000
CNIC: 34221-0891231-8
Email: saad@gmail.com
Phone number: 0300-2156789
Balance: 169

Username: LuckyLuce
Password: luce1lucky2
Name: Lucy Fionse
Age: 36
Sex: O
Date of birth: 2/15/1984
CNIC: 12345-0923748-3
Email: lucy@gmail.com
Phone number: 0335-1278394
Balance: 50

Username: CuteKate
Password: Katie123
Name: Katie Bell
Age: 21
Sex: F
Date of birth: 3/28/2000
CNIC: 42201-7035678-1
Email: cutie@yahoo.com
Phone number: 0335-2119890
Balance: 0
```



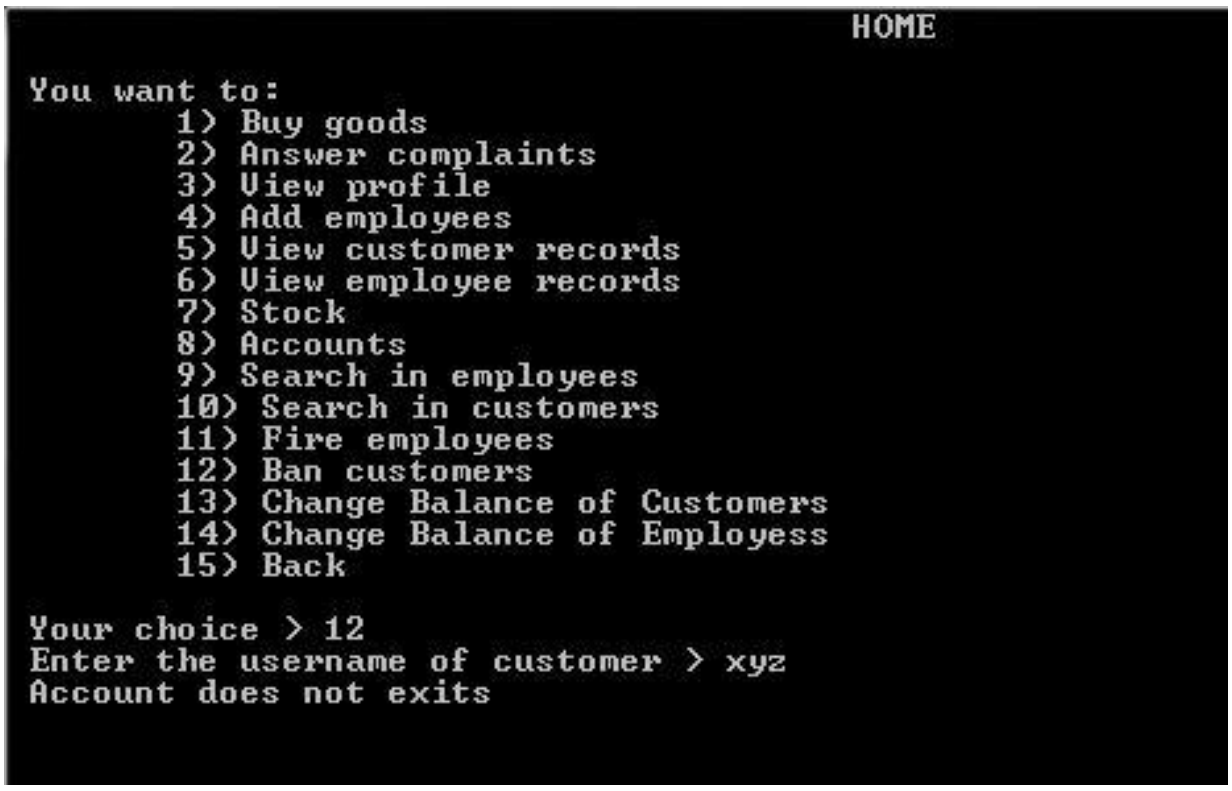
```
                                                 EMPLOYEE RECORDS

Username: emp
Password: EmP
Name: Mohammad Mudabbir
Age: 18
Sex: M
Date of birth: 10/5/2002
CNIC: 34201-0891231-8
Email: mudabbir@gmail.com
Phone number: 1111-2123708
Balance: 100

Username: Mary
Password: Mary132
Name: Mariam Sajjad
Age: 30
Sex: F
Date of birth: 9/18/1990
CNIC: 42014-1239021-4
Email: mariam18@yahoo.com
Phone number: 0333-3021984
Balance: 590


Username: Yussi
Password: yussi
Name: Yusra Tanvir
Age: 23
Sex: F
Date of birth: 6/26/1997
CNIC: 42201-1234567-8
Email: yussi@hotmail.com
Phone number: 0333-3023987
Balance: 0
```



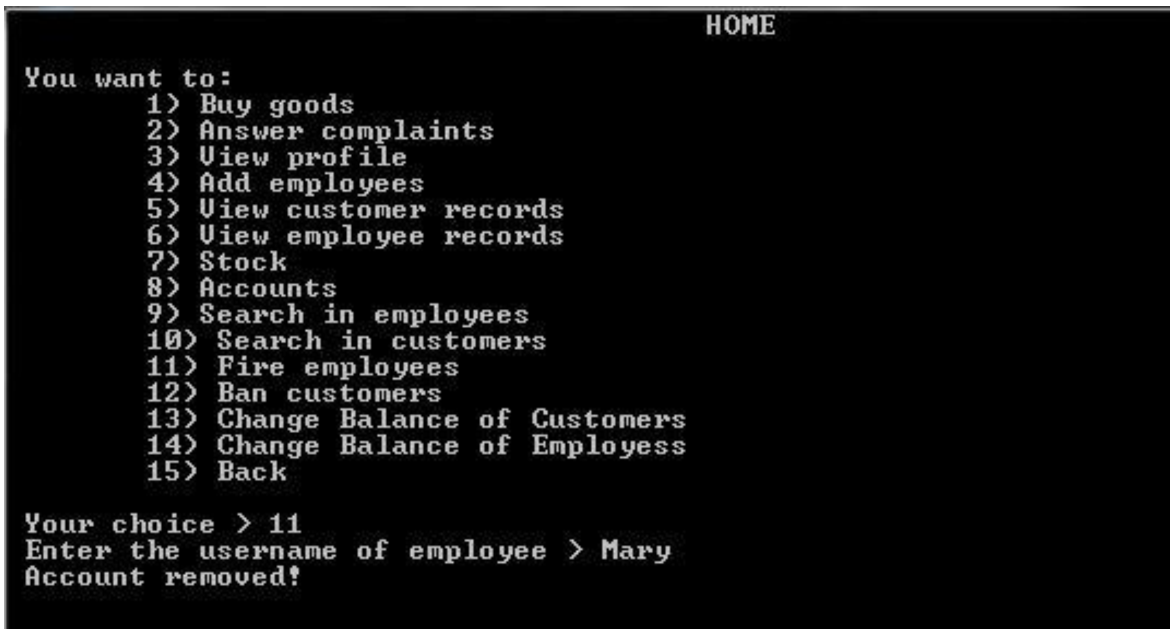
```
                                                 HOME

 You want to:
        1) Buy goods
        2) Answer complaints
        3) View profile
        4) Add employees
        5) View customer records
        6) View employee records
        7) Stock
        8) Accounts
        9) Search in employees
        10) Search in customers
        11) Fire employees
        12) Ban customers
        13) Change Balance of Customers
        14) Change Balance of Employess
        15) Back

 Your choice > 14
 Enter username: Yussi
 Enter new balance: $40
 Balance updated successfully
```



```
                                        STOCK REPORT

Name: Vegetables
Stock: 194
Cost: 3
Price: 5
Reorder Level: 100
Reorder Amount: 100

Name: Fruits
Stock: 174
Cost: 2.5
Price: 4
Reorder Level: 150
Reorder Amount: 200

Name: Lentils
Stock: 300
Cost: 5
Price: 8
Reorder Level: 200
Reorder Amount: 250

Name: Cereals
Stock: 90
Cost: 8
Price: 10
Reorder Level: 50
Reorder Amount: 100

Name: Oils
Stock: 350
Cost: 4
Price: 6.5
Reorder Level: 200
Reorder Amount: 300

Name: Eggs
Stock: 40
Cost: 2
Price: 3.5
Reorder Level: 100
Reorder Amount: 100

Name: Butter
Stock: 66
Cost: 2.5
Price: 4
Reorder Level: 50
Reorder Amount: 100
```

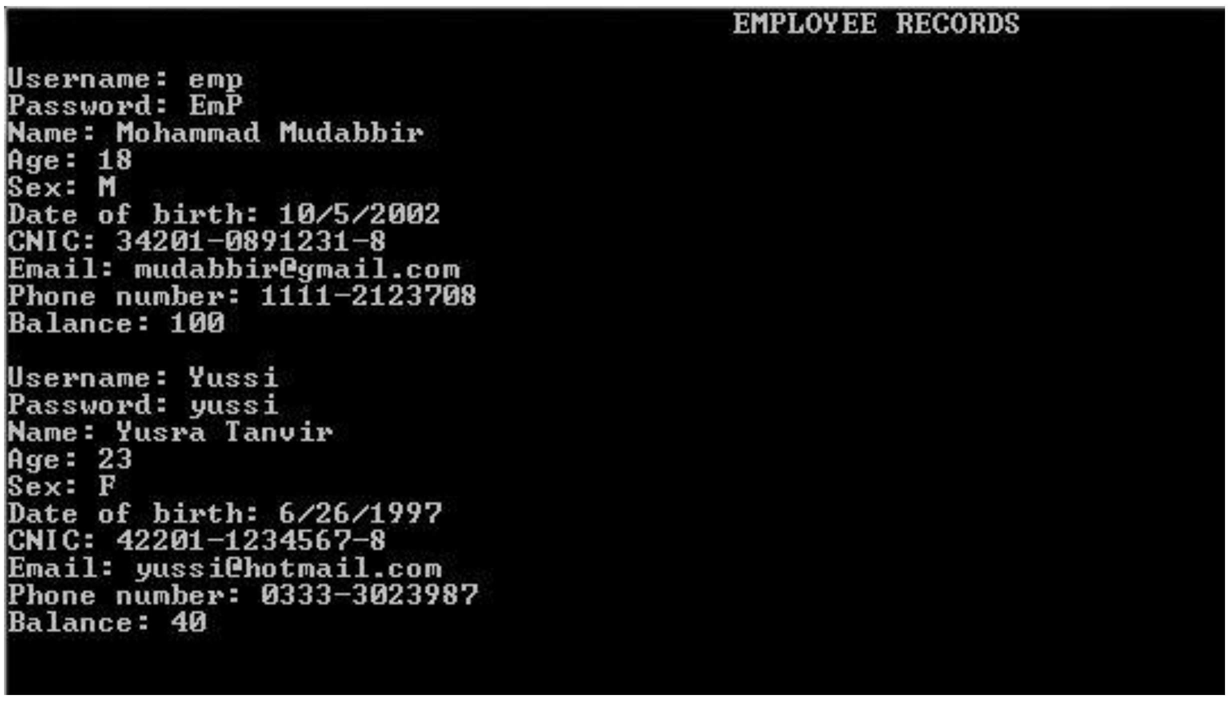


```
                                        HOME

You want to:
        1) Buy goods
        2) Answer complaints
        3) View profile
        4) Add employees
        5) View customer records
        6) View employee records
        7) Stock
        8) Accounts
        9) Search in employees
        10) Search in customers
        11) Fire employees
        12) Ban customers
        13) Change Balance of Customers
        14) Change Balance of Employess
        15) Back

Your choice > 12
Enter the username of customer > xyz
Account does not exits
```



```
                                        HOME

You want to:
        1) Buy goods
        2) Answer complaints
        3) View profile
        4) Add employees
        5) View customer records
        6) View employee records
        7) Stock
        8) Accounts
        9) Search in employees
        10) Search in customers
        11) Fire employees
        12) Ban customers
        13) Change Balance of Customers
        14) Change Balance of Employess
        15) Back

Your choice > 11
Enter the username of employee > Mary
Account removed!
```

The record is removed from the file:



```
                                        EMPLOYEE RECORDS

Username: emp
Password: EmP
Name: Mohammad Mudabbir
Age: 18
Sex: M
Date of birth: 10/5/2002
CNIC: 34201-0891231-8
Email: mudabbir@gmail.com
Phone number: 1111-2123708
Balance: 100

Username: Yussi
Password: yussi
Name: Yusra Tanvir
Age: 23
Sex: F
Date of birth: 6/26/1997
CNIC: 42201-1234567-8
Email: yussi@hotmail.com
Phone number: 0333-3023987
Balance: 40
```

Networking Layer

Introduction:

The networking layer consists of those classes and functions which are responsible for establishing a connection between a client and the remote sever.

Structure:

The layer constitutes of following classes:

1. Server.
2. Thread (workers).
3. Client.

These classes have been designed to provide general functionalities which can be utilized in any type of C++ program. By default, the connection is designed to operate on local host.

Concepts covered:

1. Abstraction:

The classes provide high levels of abstraction as the user needs to be aware of only two functions:

- Send()
- Rec()

The Send() function of server and client classes are responsible for sending information in form of numeric and string values; whereas, the Rec() function is responsible to receive that in formation.

It is worth to note here that the call to Rec() function is a blocking call i.e. the program will wait until the information is received.

A piece of code might help to you to understand the experience of a user clearly:

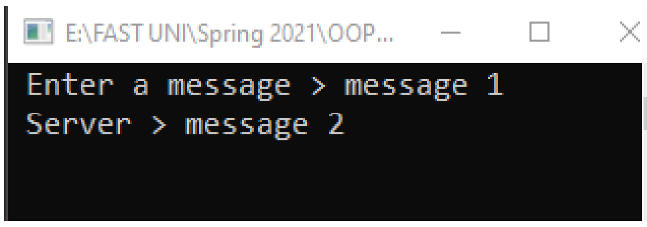
Client side:

```
cout << " Enter a message > ";
string msg;
getline(cin, msg);
// sending a string to server
client.Send(msg);

string rec;
// receiving a string from server
client.Rec(rec);
cout << " Server > " << rec << endl;
```

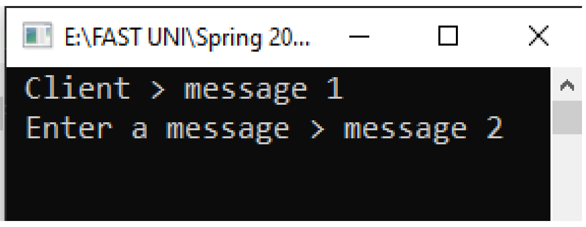
Server side:

```
// receiving a message from client:
string rec;
server.Rec(rec);
cout << " Client > " << rec << endl;

cout << " Enter a message > ";
string msg;
getline(cin, msg);
// sending a string to client
server.Send(msg);
// ending the server
server.endServer();
```

Output:



```
Enter a message > message 1
Server > message 2
```



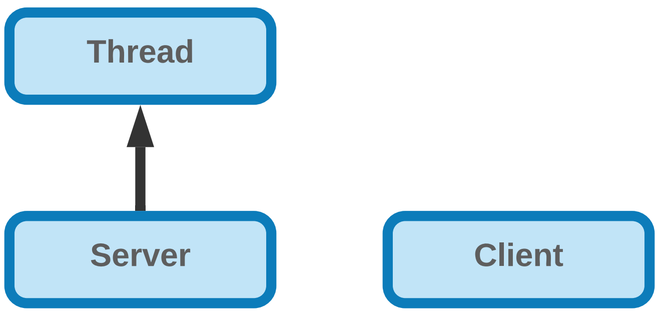
```
Client > message 1
Enter a message > message 2
```

2. Encapsulation:

The class is structured in separate .h files (for declarations) and .cpp files (for definitions) which provides the perfect encapsulation and security for the end-user.

3. Inheritance:

The server class inherits from thread class:



4. Polymorphism:

The class also uses the concepts of polymorphism to reduce the burden from user. As demonstrated earlier, The Send() function provided in the classes are designed to accept three kinds of datatypes: int, double and std::string.

```
int Send(const std::string&) const
int Send(const std::double&) const
int Send(const std::int&) const
```

5. Networking:

The layer utilizes the windows APIs of networking known as Winsock APIs which provides necessary functions required to establish a connection. It is worth to note here that this layer uses the TCP internet protocol of IPv4 version as they are more reliable than any other protocol.

Although Winsock provides no functionality of a multithreading server, the layer uses a little bit of kernel programming to create necessary threads required to serve a client asynchronously.

The header files includes for the mentioned functionalities are:

- ws2tcpip.h
- windows.h

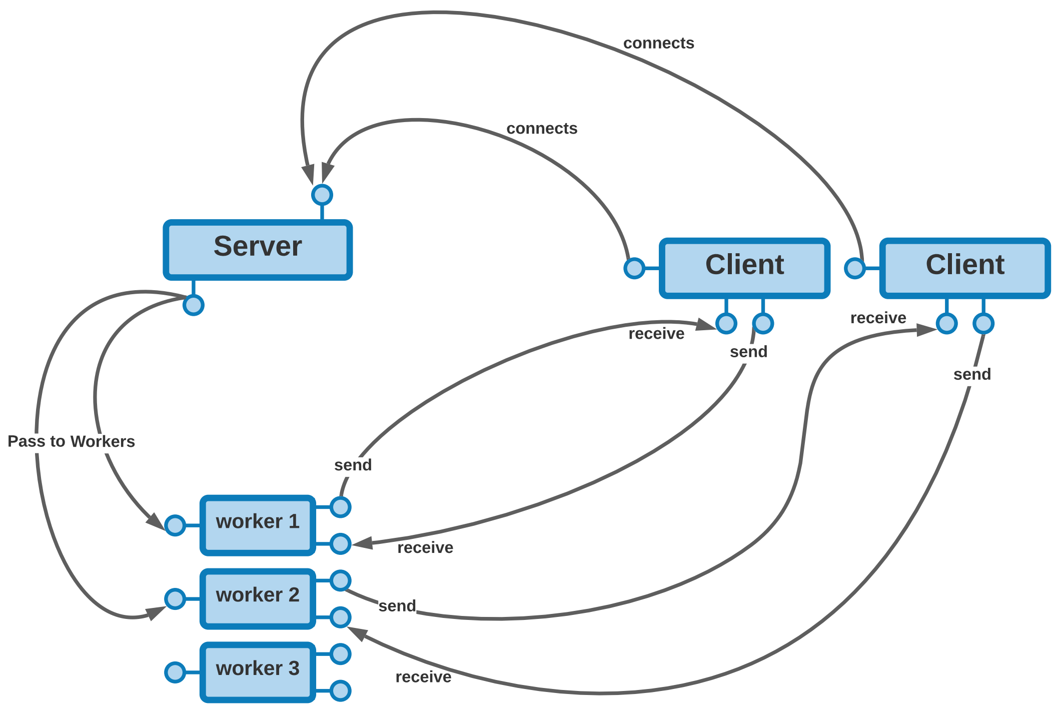
Detailed Overview:

Prerequisites:

The report assumes that the reader is aware of basic terminologies and concepts utilized in networking aspect of Computer Science.

Interaction Diagram:

The classes interact with each other like:



1. Client:

   The client class is responsible to make a successful connection with the server at the provided address. After the connection, the class provides functions of Send() and Rec() to communicate with the server.

2. Server:

   The server class consists of functions that are required to accept a connection from remote client and pass the client to the thread pool where the threads will deal with it independently of the server.

Note that the server is able to accept multiple clients at once and is not bothered by their numbers.

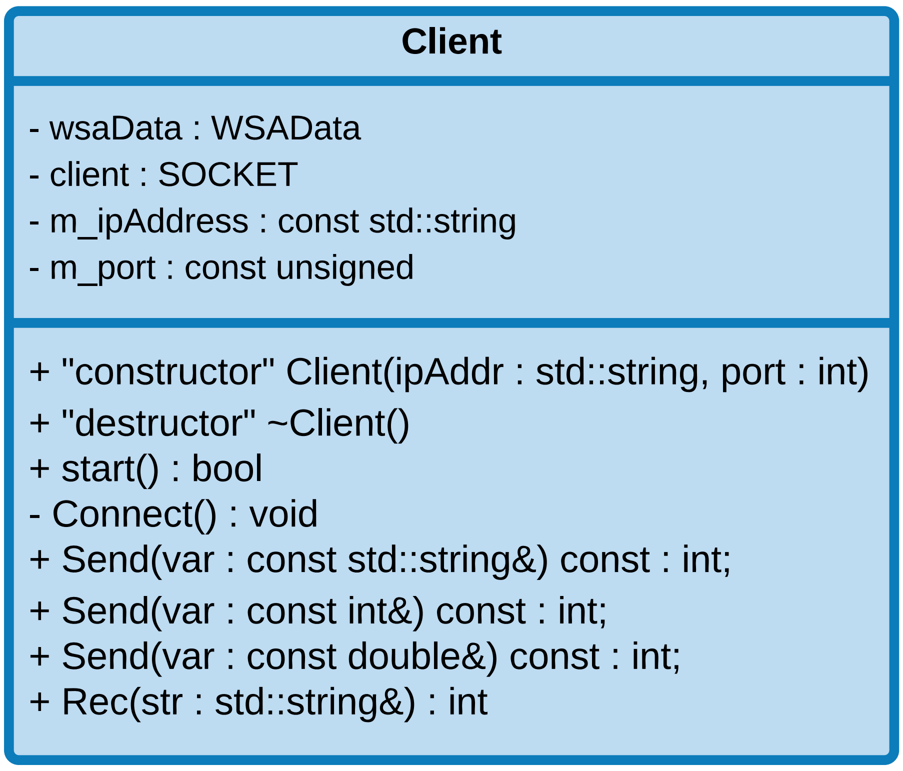
3. Thread Pool:

A thread is a process which is able to progress independently of the parent thread (the server in this case). The thread pool consists of a number of suspended threads. When the pool detects a client, it searches for a free thread, and passes the client to it, where it deals with the client according to the provided logic. The thread also provides functions of Send() and Rec() in order to exchange information with the remote client.

Class Diagrams:

The class diagrams of classes along with a summarized explanation of their member data fields and functions are mentioned below:

1. Client:



```
Client
- wsaData : WSAData
- client : SOCKET
- m_ipAddress : const std::string
- m_port : const unsigned
+ "constructor" Client(ipAddr : std::string, port : int)
+ "destructor" ~Client()
+ start() : bool
- Connect() : void
+ Send(var : const std::string&) const : int;
+ Send(var : const int&) const : int;
+ Send(var : const double&) const : int;
+ Rec(str : std::string&) : int
```

- *Data members:*

1. wsaData : WSAData

It is a private structure defined in winsock.h which is used to initialize winsock APIs of desired version.

2. client : SOCKET

SOCKET is an alias for type u_int. It is actually used as a private identifier for the client socket. Almost all of the operations of winsock are performed using this data type.

3. **m_ipAddress : const std::string**

   A const string private member which holds the ip address of the remote server. By default, its value is "127.0.0.1".

4. **m_port : const unsigned**

   A const unsigned private member which holds the port number of the remote server. By default, its value is "7777".

- *Function members:*

1. **"constructor" Client (ipAddr : const string&, port: const int &)**

   The constructor is used to initialize the ip address and port number of the remote server. The default values are "127.0.0.1" and "7777" respectively if the parameters are omitted.

2. **"destructor" ~Client**

   The destructor is used to cleanup and close the winsock APIs.

3. **start() : bool**

   A public member function which initializes the winsock APIs. If an error occurs, it returns false; otherwise, it returns true and also connects with the server.

4. **Connect() : void**

   A private member function which creates a socket and connects with the server at the specified address.

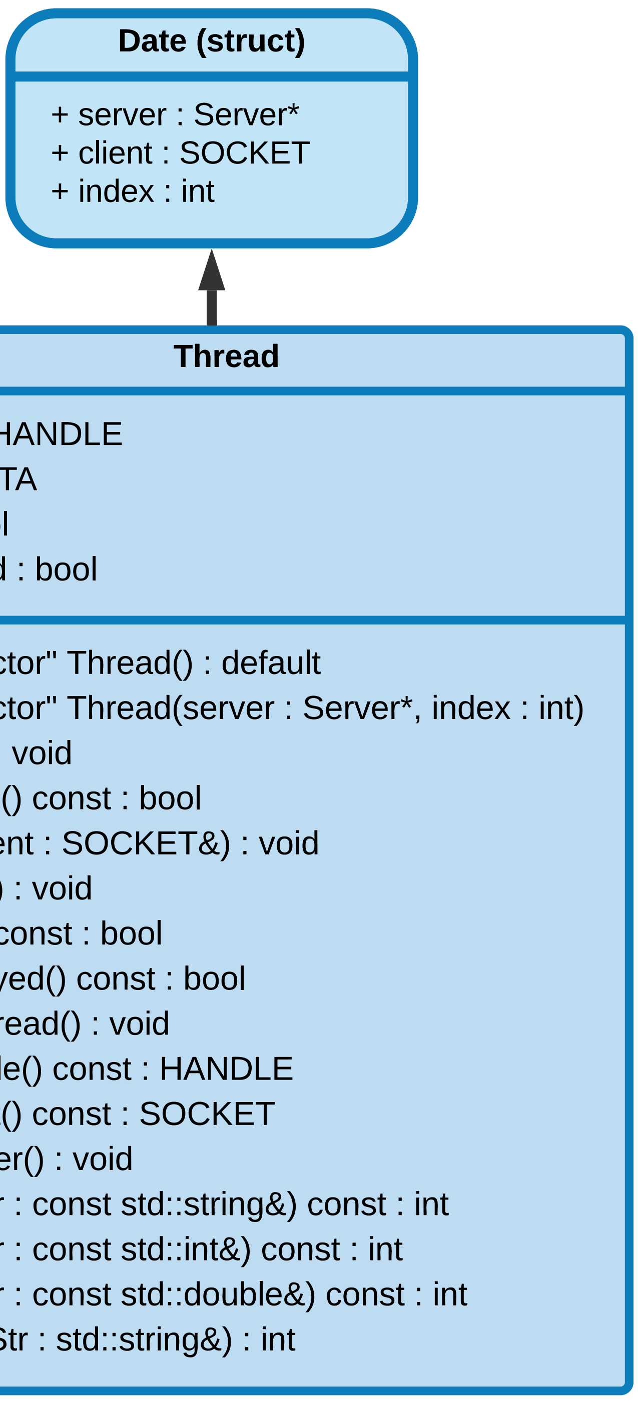
5. **Send(…) : int**

   A public function which is used to send information across the connection. It consists of 3 overloads which are able to take int, double, and strings as their arguments. Note that although the inputs are more than one data types, the information is still converted to a char* and then sent over to the server. The function returns the number of bytes sent.

6. **Rec(str : string&) : int**

   A public function which is used to receive information across the connection. Since the APIs provide the exchange of information in char* only, the arguments of Rec() is a reference of string whose value is overridden by the received information. The function returns the number of bytes received.

2. Thread:


Date (struct)
+ server : Server*
+ client : SOCKET
+ index : int
Thread
- handle : HANDLE
- data : DATA
- free : bool
- destroyed : bool
+ "constructor" Thread() : default
+ "constructor" Thread(server : Server*, index : int)
+ create() : void
+ available() const : bool
+ enter(client : SOCKET&) : void
+ setFree() : void
+ isFree() const : bool
+ isDestroyed() const : bool
+ CloseThread() : void
+ getHandle() const : HANDLE
+ getClient() const : SOCKET
+ endServer() : void
+ Send(var : const std::string&) const : int
+ Send(var : const std::int&) const : int
+ Send(var : const std::double&) const : int
+ Rec(recStr : std::string&) : int

- *Data members:*

1. **handle : HANDLE**

   HANDLE is a private identifier for a kernel object. In this case, the kernel object is the thread which we are going to create.

2. **data : DATA**

   Data is the private structure which is going to house the information about the sever which is going to use the thread(s). it consists of the address of the server, the client which needs to be handled, and the index of this thread in thread pool. Such information is necessary because we are going to control some features of the server from threads as well. Note that the server has declared this Thread class as its own friend.

3. **free : bool**

   A private bool member which holds the state of thread's availability. If it is free i.e not in the working state, the value is true; otherwise, it is false. By default, it is true.

4. **destroyed : bool**

   A private bool member which holds the state of thread's destruction. If it is destroyed i.e used up by a client, the value is true; otherwise, it is false. By default, it is false.

- *Function members:*

1. **"constructor" Thread() : default**

   The default constructor for thread object.

2. **"constructor" Thread (server : Server*,index : int)**

   The constructor is used to initialize the local data structure.

3. **create() : void**

   A public member function which creates a suspended (inactive) thread out of the Handle data member by using the kernel functions provided by windows.h. Note that it also changes the state of thread from free to busy. The thread is set to be directed towards a global function "worker" which acts as the main body of the thread.

4. **available() const : bool**

   A public member function which returns true if the thread is available i.e free and not destroyed.

5. **enter(client : SOCKET&) : void**

   A public function which takes the client, which is to be handled, as its argument and resumes the inactive thread.

6. **setFree() : void**

   A public function which sets the state of thread as free upon its completion or destruction. The thread is now free but destroyed.

7. **isFree() const : bool**

   A public function which returns true if thread is free and false if its busy.

8. **isDestroyed() const : bool**

A public function which returns true if thread is destroyed and false if it is not.

9. **CloseThread() : void**

   A public function which closes the thread.

10. **getHandle() const : HANDLE**

    A public function which returns the HANDLE of the thread.

11. **getClient() const : SOCKET**

    A public function which returns the client of the thread.

12. **endServer() : void**

    A public function which closes the server by changing its exit bool member to true.

13. **Send(…) : int**

    A public function which is used to send information across the connection. It consists of 3 overloads which are able to take int, double, and strings as their arguments. Note that although the inputs are more than one data types, the information is still converted to a char* and then sent over to the server. The function returns the number of bytes sent.

14. **Rec(str : string&) : int**

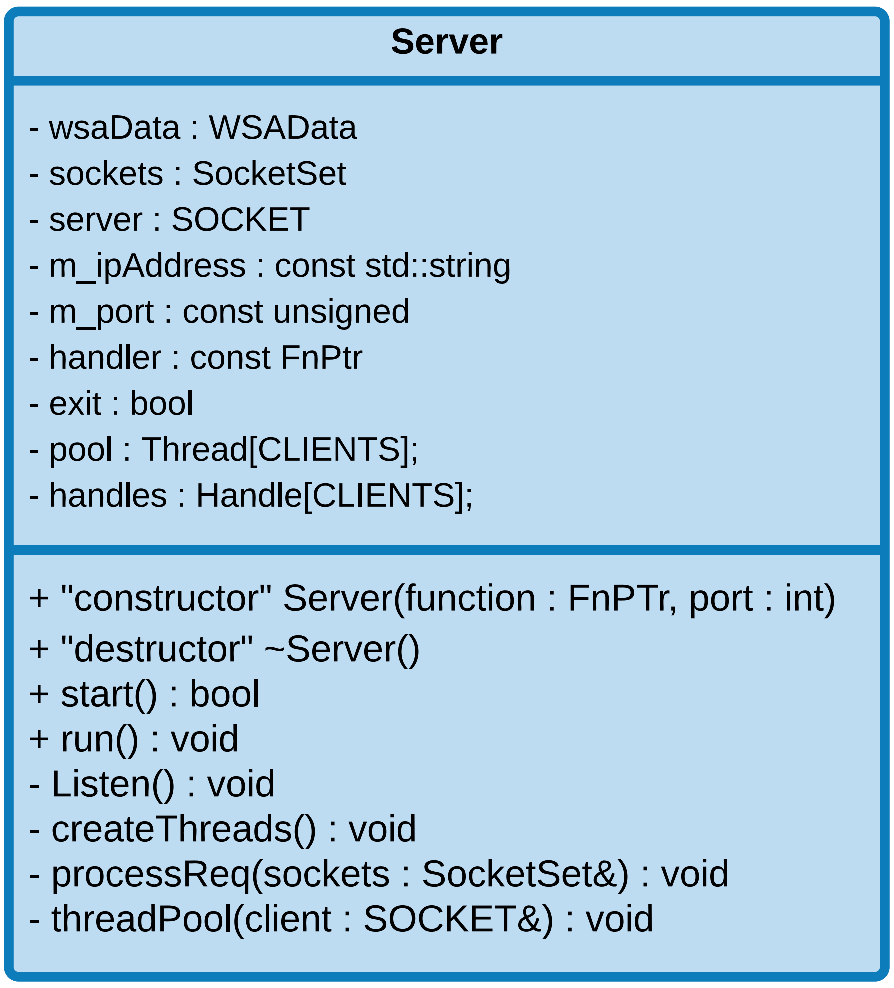
    A public function which is used to receive information across the connection. Since the APIs provide the exchange of information in char* only, the arguments of Rec() is a reference of string whose value is overridden by the received information. The function returns the number of bytes received.

15. **worker(Data : DATA*) : void**

    A static non-member function which acts as the main body of the thread. The suspended thread is resumed into this block of code. Note that the function is termed as static because it is a separate thread and process which is independent of other programs.

    It is able to control the server from the provided data structure (it is also a friend of Server class). It is responsible to pass the thread object into the Handler function of server from where the user can write the code to further handle the client. Once the function has ended, it sets the state of thread to free but destroyed.

3. Server:



Server

```
- wsaData : WSAData
- sockets : SocketSet
- server : SOCKET
- m_ipAddress : const std::string
- m_port : const unsigned
- handler : const FnPtr
- exit : bool
- pool : Thread[CLIENTS];
- handles : Handle[CLIENTS];
```

```
+ "constructor" Server(function : FnPTr, port : int)
+ "destructor" ~Server()
+ start() : bool
+ run() : void
- Listen() : void
- createThreads() : void
- processReq(sockets : SocketSet&) : void
- threadPool(client : SOCKET&) : void
```

- *Data members:*

1. **wsaData : WSAData**

   It is a private structure defined in winsock.h which is used to initialize winsock APIs of desired version.

2. **sockets : SocketSet**

   SocketSet is a local alias of fd_set data type. It is basically a structure which holds information about a set of file descriptors. Since SOCKET is also a kind of file descriptor, SocketSet is able to house a set of SOCKETS as well.

   This variable the main set which is going to include all of the sockets that out server is going to encounter e.g. server socket, client socket etc.

3. **server : SOCKET**

   A private identifier for the server socket

4. **m_ipAddress : const std::string**

   A const string private member which holds the ip address of the server. By default, its value is “127.0.0.1”.

5. **m_port : const unsigned**

A const unsigned private member which holds the port number of the server. By default, its value is “7777”.

6. **handler : const Fnptr**

   FnPtr is a local alias of “void (*) (Thread&)” function pointer. Handler is a const private FnPtr which is initialized by the user in the Constructor. The passed function is later used by the Threads as their secondary process. In short, it is the function which is going to be the thread of our server. The user is going to program the thread in this function.

7. **exit : bool**

   A private member which is responsible to run or stop the server completely. The default value is false.

8. **pool : Thread[CLIENTS]**

   An array of Thread objects which is also known as the thread pool. CLIENT is defined as a macro “#define CLIENT (5)” for various reasons. It is actually the maximum number of clients that the server is able to listen at once. Once can change this number the way he/she wishes; however, for the sake of simplicity, we will keep it 5 for now.

9. **Handles: HANDLE[CLIENTS]**

   An array of HANDLES of the respective threads. It is actually a kernel identifier which is used to close the threads.

- *Function members:*

1. **“constructor” Server (function : FnPtr, port: const int)**

   The constructor is used to initialize thread function and port number of the remote server. The default value of port is “7777” where as the thread function is compulsory.

2. **“destructor” ~Server**

   The destructor is used to close all sockets, threads and the winsock APIs.

3. **start() : bool**

   A public member function which initializes the winsock APIs. If an error occurs, it returns false; otherwise, it returns true and also creates suspended threads and a listening server socket.

4. **createThreads() : void**

   A private member function which creates suspended and free threads out of the array of Thread objects. The thread is now inactive but gets resumed as soon as the client socket encounters it.

5. **Listen() : void**

   A private function which creates a listening socket using winsock functionalities. It creates a socket, then binds it to the provided address. If binding is performed correctly, the socket is also made to listen at the provided port. Note that the socket is able to listen to CLIENTS number of clients at once.

6. **run(): void**

   A public function from where the server is initiated. It continuously looks out for an activity on the port. The function also keeps a track of all active sockets in another SocektSet variable. If there is some kind of request made at the port, the function passes command to the processRequest() function.

7. **processRequest (activeSockets : SocketSet&): void**

   A private function which handles the activity detected at the port. It iterates through the main socket set and identifies the socket which is responsible for the request. If activity is made for the listening socket itself (a connection is detected), the server accepts the connection and pass the client to the threadPool() function.

8. **threadPool (client: SOCKET&): void**

   A private function which handles the client accepted by the listening socket. It iterates through the thread objects array one by one and checks if a thread is available or not. If it is available and free, the client is passed into the thread and the suspended thread gets resumed; otherwise, if the thread is destroyed (used by a previous client), then the thread is created again and made to accept the client. Note that the thread pool is able to handle CLIENT number of clients only. After the client is passed into the thread, the server recycles to listen for more connection at the port.

Explanation of Networking functions:

Since the layer utilizes windows APIs of networking, we think it is better to explain those functionalities as well. The APIs were used to carry out following task:

For server:

1. Create a socket
2. Bind a socket
3. Listen on a socket
4. Accept a connection

For client:

1. Create a socket
2. Connect to an address

For both client and server:

1. Send information
2. Receive information

1. For Server:

Create a socket:

The socket function creates a socket that is bound to a specific transport service provider.

It has three arguments which are:

- Address family.
- Type of Sockets.
- Protocol specification.

```
SOCKET server = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
// AF_INET for IPv4 family
// SOCK_STREAM for streaming sockets
// IPPROTO_TCP for TCP protocol connections
```

If no error occurs, the function returns a descriptor referencing the new socket. Otherwise, a value of INVALID_SOCKET is returned.

Bind a socket:

The bind function associates a local address with a socket.

It has three arguments which are:

- A descriptor identifying the socket.
- A pointer to a sockaddr structure of address to assign.

- The length of sockaddr structure.

```
// filling the struct with the address
sockaddr_in addr;
// for IPv4
addr.sin_family = AF_INET;
// for port number
addr.sin_port = htons(m_port);
// for ipAddress
addr.sin_addr.S_un.S_addr = inet_addr(m_ipAddress.c_str());

bind(server, (sockaddr *)&addr, sizeof(addr)
```

If no error occurs, bind returns zero. Otherwise, it returns SOCKET_ERROR.

Listen on socket:

The listen function places a socket in a state in which it is listening for an incoming connection.

It has two arguments which are:

- The socket to listen on.
- The number of clients to listen.

```
listen(server, 5);
```

If no error occurs, listen returns zero. Otherwise, a value of SOCKET_ERROR is returned.

Accept a connection:

The accept function permits an incoming connection attempt on a socket.

It has three arguments which are:

- A listening socket
- An optional pointer to a buffer that receives the address.
- A pointer to size of buffer

```
SOCKET client = accept(sock, nullptr, nullptr);
```

If no error occurs, accept returns a value of type SOCKET that is a descriptor for the accepted socket. Otherwise, a value of INVALID_SOCKET is returned.

2. For Client:

Create a socket:

The socket function creates a socket that is bound to a specific transport service provider.

It has three arguments which are:

- Address family.
- Type of Sockets.
- Protocol specification.

```
SOCKET client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
// AF_INET for IPv4 family
// SOCK_STREAM for streaming sockets
// IPPROTO_TCP for TCP protocol connections
```

If no error occurs, the function returns a descriptor referencing the new socket. Otherwise, a value of INVALID_SOCKET is returned.

Connect to an Address:

The connect function establishes a connection to a specified socket.

It has three arguments which are:

- A descriptor identifying an unconnected socket.
- A pointer to the sockaddr structure to which the connection should be established.
- The length of sockaddr structure.

```
// filling the struct with the address
sockaddr_in addr;
// for IPv4
addr.sin_family = AF_INET;
// for port number
addr.sin_port = htons(m_port);
// for ipAddress
addr.sin_addr.S_un.S_addr = inet_addr(m_ipAddress.c_str());

connect(client, (sockaddr *)&addr, sizeof(addr)
```

If no error occurs, connect returns zero. Otherwise, it returns SOCKET_ERROR.

3. For both Client and Server:

Send information:

The send function sends data on a connected socket.

It has three arguments which are:

- A connected socket
- Pointer to the buffer (char* in our case)
- Size of buffer.
- Optional flags for various reasons.

```
send(client, str.c_str(), str.size() + 1, 0);
```

If no error occurs, send returns the total number of bytes sent. Otherwise, a value of SOCKET_ERROR is returned.

Receive information:

The recv function receives data from a connected socket.

It has three arguments which are:

- A connected socket
- Pointer to the buffer (char* in our case)
- Size of buffer.
- Optional flags for various reasons.

```
char buffer[4069];
int bytesRec = recv(client, buffer, sizeof(buffer), 0);
```

If no error occurs, send returns the total number of bytes received. Otherwise, a value of SOCKET_ERROR is returned.

Explanation of Thread functions:

We have also used some functions of windows kernel programming for following purposes:

1. To create threads
2. To resume threads
3. To close thread

1. Create a thread:

The CreateThread() function creates a thread to execute as a separate process.

It has following six parameters:

- A pointer to a structure that determines whether the returned handle can be inherited by child processes.
- The initial size of the stack, in bytes.
- A pointer to the function to be executed by the thread.
- A pointer to a variable to be passed to the thread.
- The flags that control the creation of the thread.
- A pointer to a variable that receives the thread identifier (ID)

```
HANDLE handle = CreateThread(nullptr, 0, (LPTHREAD_START_ROUTINE)worker,
 &data, CREATE_SUSPENDED, nullptr);
// nullptr : to avoid inheritance by child processes
// 0       : the default size of stack
// worker  :  the function of thread
// data    :  the structure passed in thread
// CREATE_SUSPENDED :  thread does not run after creation
// nullptr : to not receive any thread ID
```

If the function succeeds, the return value is a handle to the new thread. If the function fails, the return value is NULL.

2. Resume a thread:

The ResumeThread() function resumes a suspended thread.

The only parameter is a handle to the thread to be resumed.

```
ResumeThread(handle);
```

3. Close a thread:

The CloseThread() function closes a thread.

The only parameter is a handle to the thread to be closed.

```
CloseThread(handle);
```