Cybernet Café Management System.

Group Members: Umer Ahmed, 19K-0181, Zaid Bin Shahab, 19K-1512, Saud Ahmed Abbasi, 19K-0229

Section: B

Course Name: Object Oriented Programming

Course Code:CS217

Instructor(s): Muhammad Danish Khan.

Department of Computer Science

Language Used: C++

Contents

0. Acknowledgment

https://www.asciiart.eu/
http://patorjk.com/software/taag/#p=display&h=0&v=0&f=Rozzo&t=Type%20Something%20

1. Introduction

Jasmine Café has been around since 1977 and has only recently started to revamp its structure from the ground up. Formerly a small traditional café, it now functions as both a regular café, and a net café, letting the customer enjoy the best of both worlds with an arsenal of one of the high-class desktop systems and an unmatched internet connection, but still retains its eastern influences.

The popularity of the café has garnered so much popularity that now it requires an online booking in advance, even allowing you to occupy specific seats (flexible enough to detect your current time).

The Cybernet Café Management System is a multifaceted system that allows up to 5 quintessential participants of the Jasmine Café to access their accounts. These participants being the **Accountant**, **Customer**, **Manager**, **Technical Staff** and the **Service Department Workers/Regular Staff**. It is robust enough to support every account separately and without the need to access a specific user-type portal individually. It not only oscillates between all of these 5 participants but also allows "Non-Account Functionality", such as booking an offline ticket, or viewing the User-Base Infographic data.

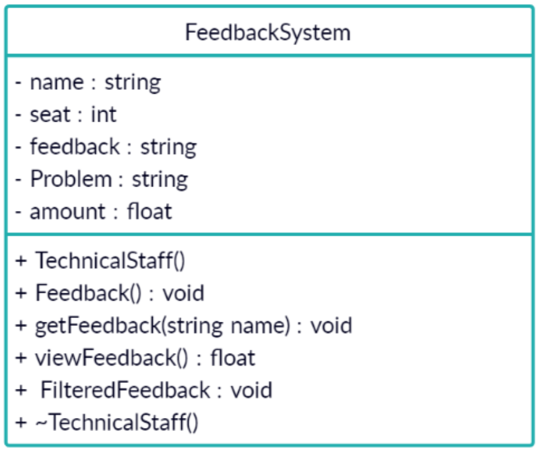
Additionally, the Management System allows some ambitious features such as the ability to give feedback, chatting between seat occupants, being able to set the menu, auto generating the schedule routine the workers have to follow, defining the stock of technical equipment, all of which is supported by the permanency of filing and the extensive yet flexible usage of strings.

2. Tools and Technologies Used

The **"Windows.h"** library was extensively used to support much of the visual content in the System, especially the utilization of **Curser Positioning** and **Color Scheming**.

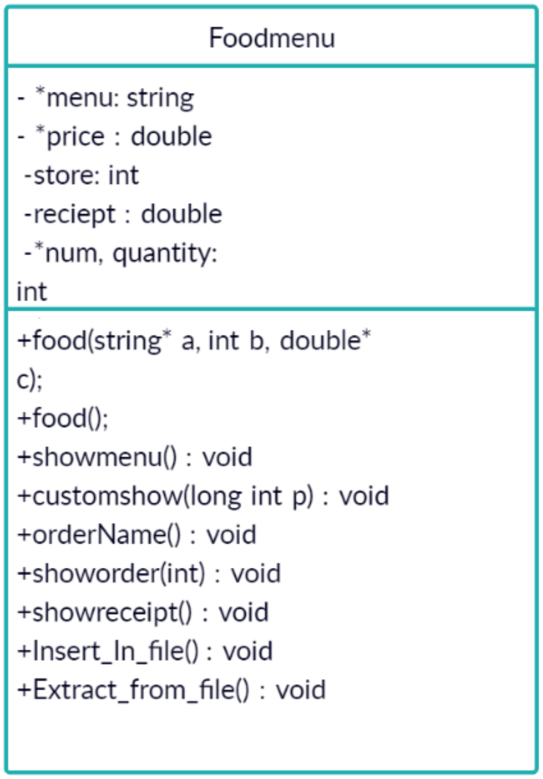
3. Class diagram

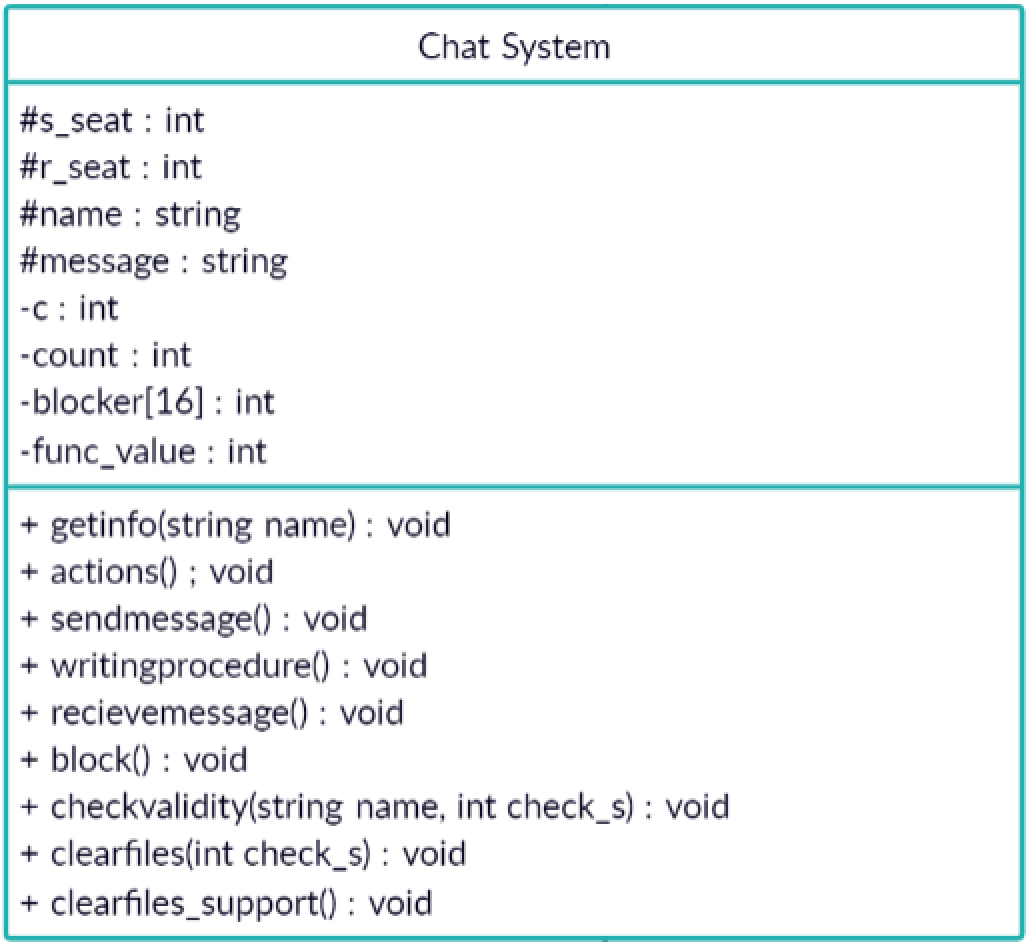
General Tools and their short descriptions:



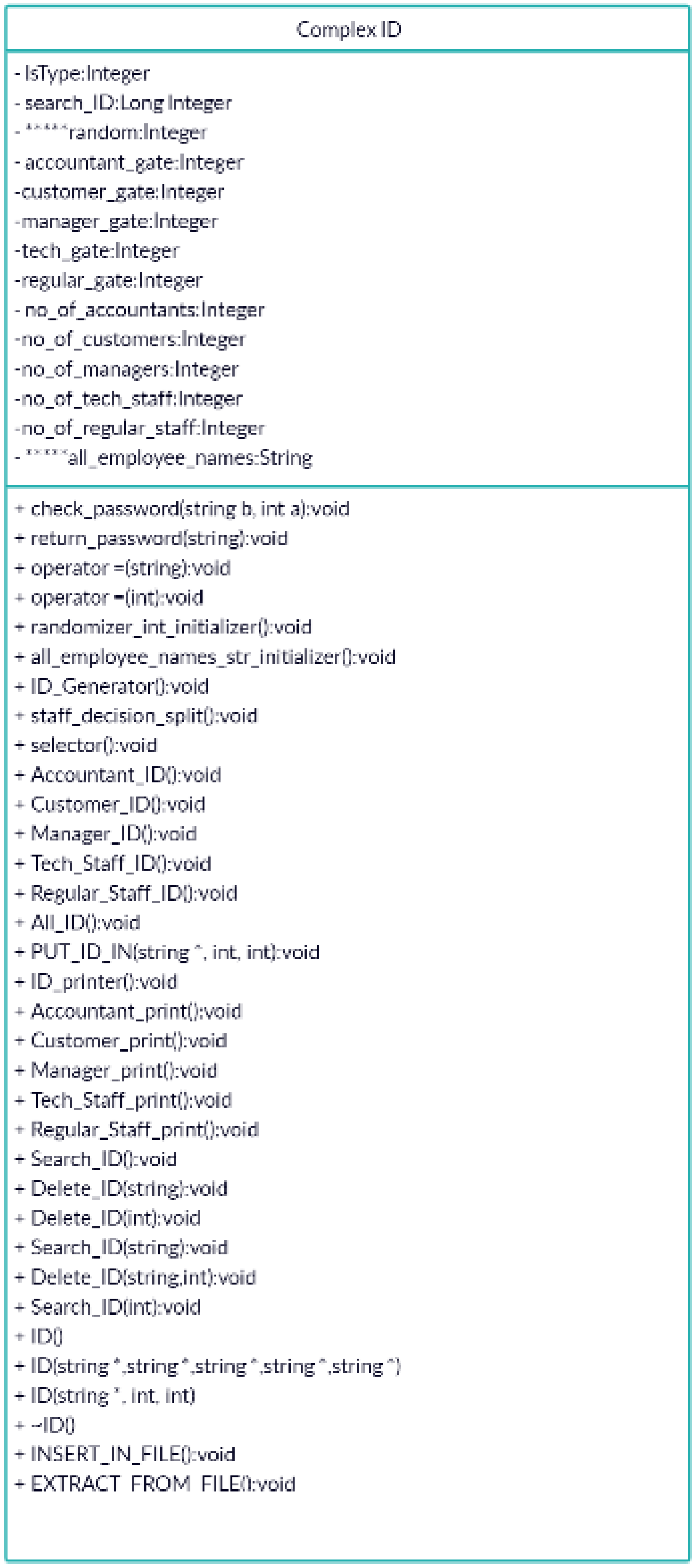
Feedback System is responsible to take feedback from the user to report for any problem in the particular seat. The user can submit their feedback in their time range. For the user who wants to submit their feedback after fully experiencing all the features or at the end of their time. This system has the ability to take feedback for upto 2 hours after their normal time.

Food menu can show the menu available or set by the café staff. This modern menu can arrange the available goods according to your entered amount. This system has the ability to order food from the breakfast café.




Chat System
#s_seat : int
#r_seat : int
#name : string
#message : string
-c : int
-count : int
-blocker[16] : int
-func_value : int
+ getinfo(string name) : void
+ actions() ; void
+ sendmessage() : void
+ writingprocedure() : void
+ recievemessage() : void
+ block() : void
+ checkvalidity(string name, int check_s) : void
+ clearfiles(int check_s) : void
+ clearfiles_support() : void

The chat system is a traditional chat system that can connect you with all the current seat users in your time range. You can message to any person available in the seat in the current time. The system also an integrated block system where you have the ability to block or unblock the seats. Every message is being stored and saved in different files to ensure low memory consumption as files are being cleared after a certain user time has been exceeded



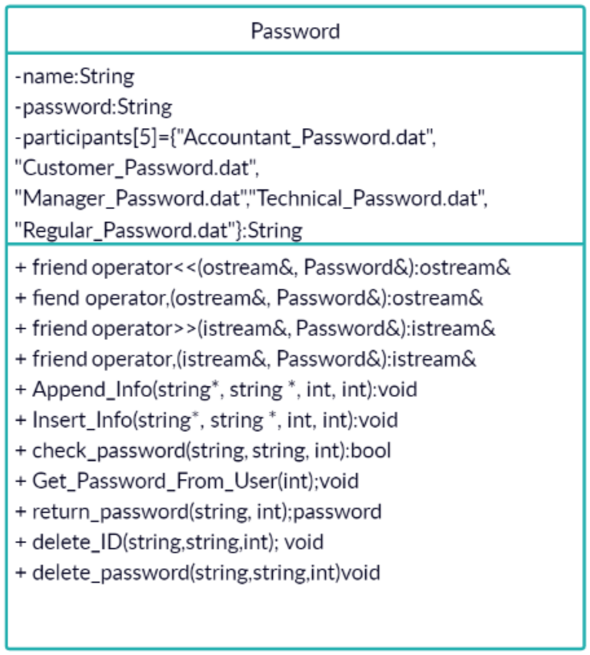
```
Complex ID

- IsType:Integer
- search_ID:Long Integer
- *****random:Integer
- accountant_gate:Integer
-customer_gate:Integer
-manager_gate:Integer
-tech_gate:Integer
-regular_gate:Integer
- no_of_accountants:Integer
-no_of_customers:Integer
-no_of_managers:Integer
-no_of_tech_staff:Integer
-no_of_regular_staff:Integer
- *****all_employee_names:String

+ check_password(string b, int a):void
+ return_password(string):void
+ operator =(string):void
+ operator =(int):void
+ randomizer_int_initializer():void
+ all_employee_names_str_initializer():void
+ ID_Generator():void
+ staff_decision_split():void
+ selector():void
+ Accountant_ID():void
+ Customer_ID():void
+ Manager_ID():void
+ Tech_Staff_ID():void
+ Regular_Staff_ID():void
+ All_ID():void
+ PUT_ID_IN(string *, int, int):void
+ ID_printer():void
+ Accountant_print():void
+ Customer_print():void
+ Manager_print():void
+ Tech_Staff_print():void
+ Regular_Staff_print():void
+ Search_ID():void
+ Delete_ID(string):void
+ Delete_ID(int):void
+ Search_ID(string):void
+ Delete_ID(string,int):void
+ Search_ID(int):void
+ ID()
+ ID(string *,string *,string *,string *,string *)
+ ID(string *, int, int)
+ ~ID()
+ INSERT_IN_FILE():void
+ EXTRACT_FROM_FILE():void
```

1. Manages all Participant IDs inside pointer data types.

2. Allows appending and deletion of IDs based on both assigned Roll Number and Name of Participant.

3. Carries Searching and Deletion functionalities.

4. Supports filing for all aspects contained within the class,


Password
-name:String
-password:String
-participants[5]={"Accountant_Password.dat",
"Customer_Password.dat",
"Manager_Password.dat","Technical_Password.dat",
"Regular_Password.dat"}:String
+ friend operator<<(ostream&, Password&):ostream&
+ fiend operator,(ostream&, Password&):ostream&
+ friend operator>>(istream&, Password&):istream&
+ friend operator,(istream&, Password&):istream&
+ Append_Info(string*, string *, int, int):void
+ Insert_Info(string*, string *, int, int):void
+ check_password(string, string, int):bool
+ Get_Password_From_User(int);void
+ return_password(string, int);password
+ delete_ID(string,string,int); void
+ delete_password(string,string,int)void

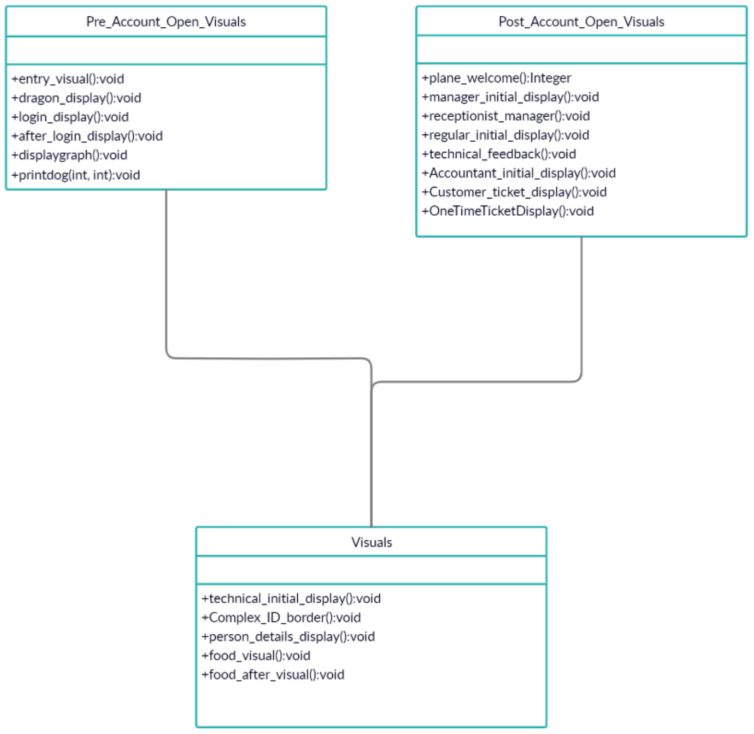
1. Saves Username and Password.

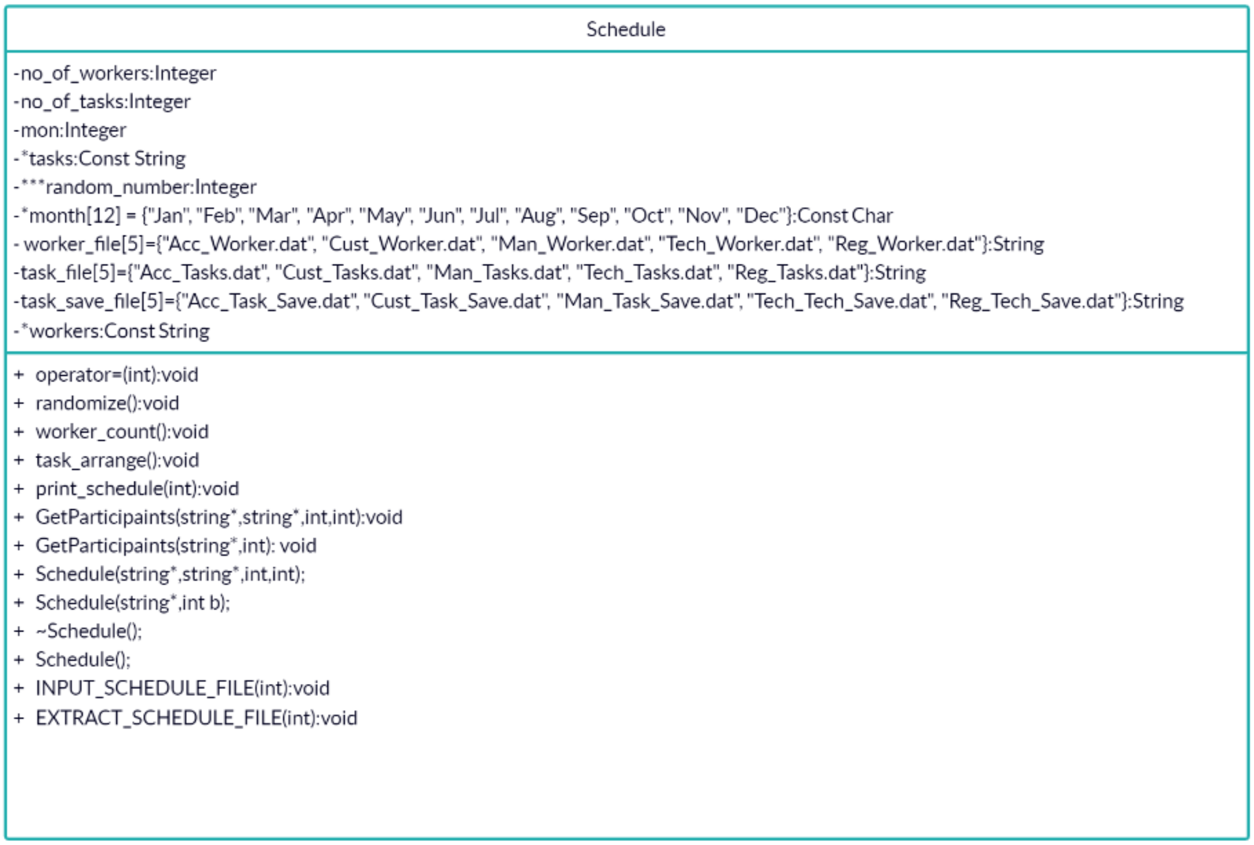
2. Ability to catch User Type based on unique User Name.

3. Supports deletion of Username and password.

1. Visual class is responsible for all the Visuals utilized during the runtime of the Management System.

2. Uses a custom-defined gotoXY to smartly craft visuals, flexible usage includes graphical representation and animated ASCIIs.


Pre_Account_Open_Visuals
+entry_visual():void
+dragon_display():void
+login_display():void
+after_login_display():void
+displaygraph():void
+printdog(int, int):void
Post_Account_Open_Visuals
+plane_welcome():Integer
+manager_initial_display():void
+receptionist_manager():void
+regular_initial_display():void
+technical_feedback():void
+Accountant_initial_display():void
+Customer_ticket_display():void
+OneTimeTicketDisplay():void
Visuals
+technical_initial_display():void
+Complex_ID_border():void
+person_details_display():void
+food_visual():void
+food_after_visual():void



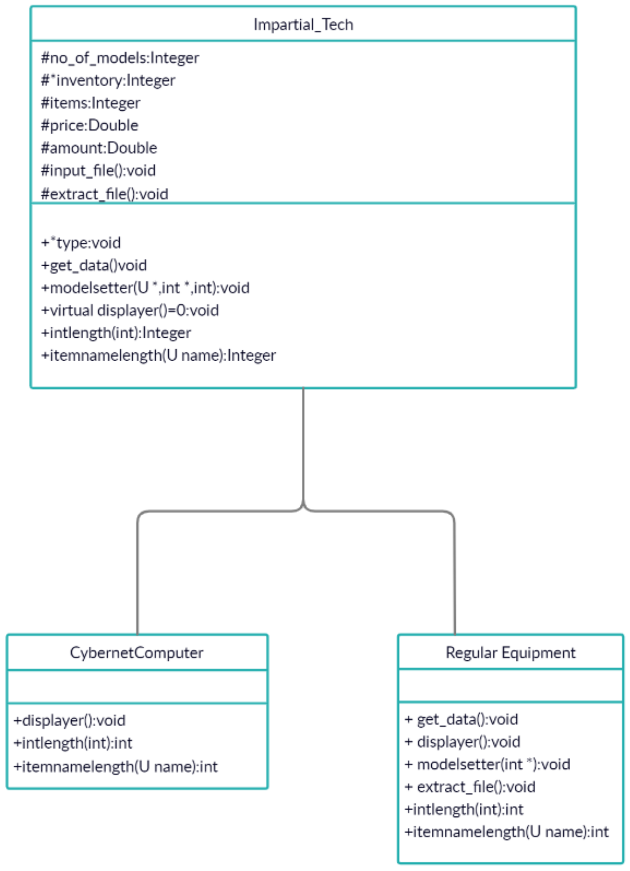
Schedule

-no_of_workers:Integer
-no_of_tasks:Integer
-mon:Integer
-*tasks:Const String
-***random_number:Integer
-*month[12] = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"}:Const Char
- worker_file[5]={"Acc_Worker.dat", "Cust_Worker.dat", "Man_Worker.dat", "Tech_Worker.dat", "Reg_Worker.dat"}:String
-task_file[5]={"Acc_Tasks.dat", "Cust_Tasks.dat", "Man_Tasks.dat", "Tech_Tasks.dat", "Reg_Tasks.dat"}:String
-task_save_file[5]={"Acc_Task_Save.dat", "Cust_Task_Save.dat", "Man_Task_Save.dat", "Tech_Tech_Save.dat", "Reg_Tech_Save.dat"}:String
-*workers:Const String

+ operator=(int):void
+ randomize():void
+ worker_count():void
+ task_arrange():void
+ print_schedule(int):void
+ GetParticipaints(string*,string*,int,int):void
+ GetParticipaints(string*,int): void
+ Schedule(string*,string*,int,int);
+ Schedule(string*,int b);
+ ~Schedule();
+ Schedule();
+ INPUT_SCHEDULE_FILE(int):void
+ EXTRACT_SCHEDULE_FILE(int):void

1. Schedule Class is responsible for asking for relevant tasks(upto 9) from the Manager and then setting them accordingly throughout the year. Generates work assignment randomly, takes off-days into account.

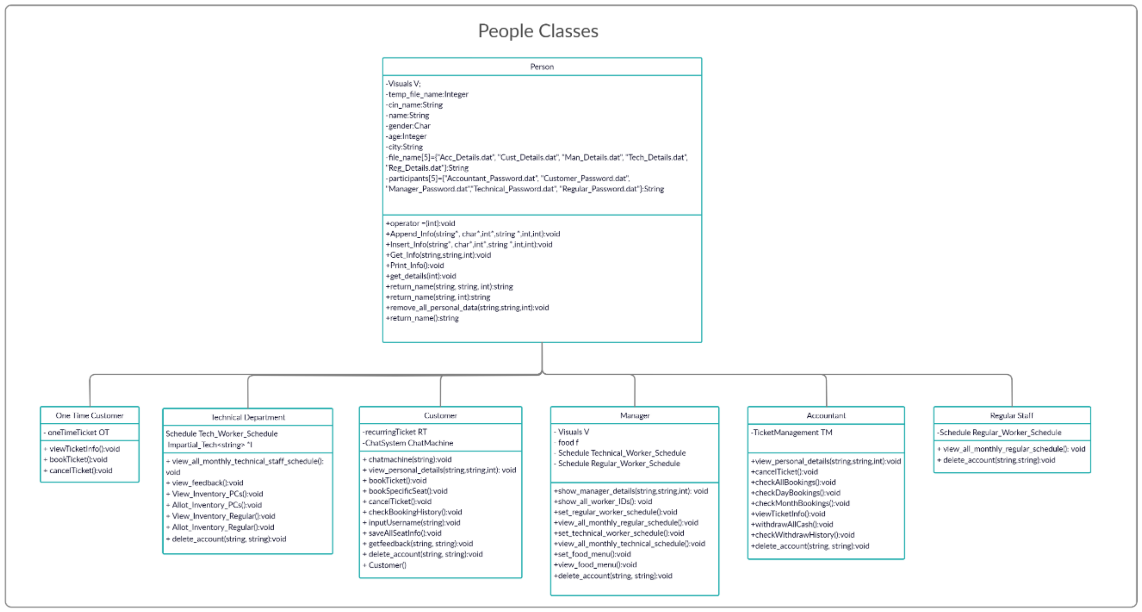
2. Schedule supports data permanency via filing.



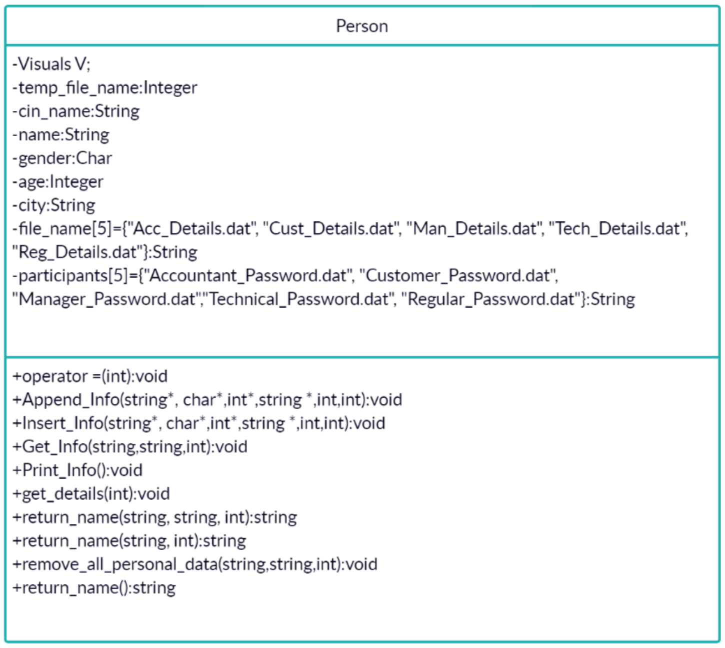
1. These classes are used for setting the technical stock amount for regular technical equipment and the desktop systems being used.

2. All stock equiment is filed and supports permanency.

MAIN CLASSES PART I:


People Classes
Person
-Visuals V;
-temp_file_name:Integer
-cin_name:String
-name:String
-gender:Char
-age:Integer
-city:String
-file_name[5]={"Acc_Details.dat", "Cust_Details.dat", "Man_Details.dat", "Tech_Details.dat", "Reg_Details.dat"}:String
-participants[5]={"Accountant_Password.dat", "Customer_Password.dat", "Manager_Password.dat","Technical_Password.dat", "Regular_Password.dat"}:String
+operator =(int):void
+Append_Info(string*, char*,int*,string *,int,int):void
+Insert_Info(string*, char*,int*,string *,int,int):void
+Get_Info(string,string,int):void
+Print_Info():void
+get_details(int):void
+return_name(string, string, int):string
+return_name(string, int):string
+remove_all_personal_data(string,string,int):void
+return_name():string
One Time Customer
- oneTimeTicket OT
+ viewTicketInfo():void
+ bookTicket():void
+ cancelTicket():void
Technical Department
Schedule Tech_Worker_Schedule
Impartial_Tech<string> *I
+ view_all_monthly_technical_staff_schedule(): void
+ view_feedback():void
+ View_Inventory_PCs():void
+ Allot_Inventory_PCs():void
+ View_Inventory_Regular():void
+ Allot_Inventory_Regular():void
+ delete_account(string, string):void
Customer
-recurringTicket RT
-ChatSystem ChatMachine
+ chatmachine(string):void
+ view_personal_details(string,string,int): void
+ bookTicket():void
+ bookSpecificSeat():void
+ cancelTicket():void
+ checkBookingHistory():void
+ inputUsername(string):void
+ saveAllSeatInfo():void
+ getfeedback(string, string):void
+ delete_account(string, string):void
+ Customer()
Manager
- Visuals V
- food f
- Schedule Technical_Worker_Schedule
- Schedule Regular_Worker_Schedule
+show_manager_details(string,string,int): void
+show_all_worker_IDs(): void
+set_regular_worker_schedule():void
+view_all_monthly_regular_schedule():void
+set_technical_worker_schedule():void
+view_all_monthly_technical_schedule():void
+set_food_menu():void
+view_food_menu():void
+delete_account(string, string):void
Accountant
-TicketManagement TM
+view_personal_details(string,string,int):void
+cancelTicket():void
+checkAllBookings():void
+checkDayBookings():void
+checkMonthBookings():void
+viewTicketInfo():void
+withdrawAllCash():void
+checkWithdrawHistory():void
+delete_account(string, string):void
Regular Staff
-Schedule Regular_Worker_Schedule
+ view_all_monthly_regular_schedule(): void
+ delete_account(string,string):void

Short Explanation For Each Class:

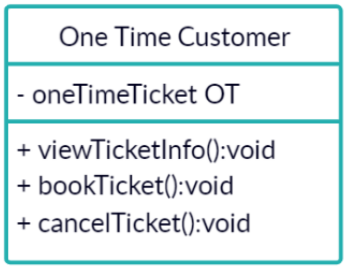

Person
-Visuals V;
-temp_file_name:Integer
-cin_name:String
-name:String
-gender:Char
-age:Integer
-city:String
-file_name[5]={"Acc_Details.dat", "Cust_Details.dat", "Man_Details.dat", "Tech_Details.dat", "Reg_Details.dat"}:String
-participants[5]={"Accountant_Password.dat", "Customer_Password.dat", "Manager_Password.dat","Technical_Password.dat", "Regular_Password.dat"}:String
+operator =(int):void
+Append_Info(string*, char*,int*,string *,int,int):void
+Insert_Info(string*, char*,int*,string *,int,int):void
+Get_Info(string,string,int):void
+Print_Info():void
+get_details(int):void
+return_name(string, string, int):string
+return_name(string, int):string
+remove_all_personal_data(string,string,int):void
+return_name():string

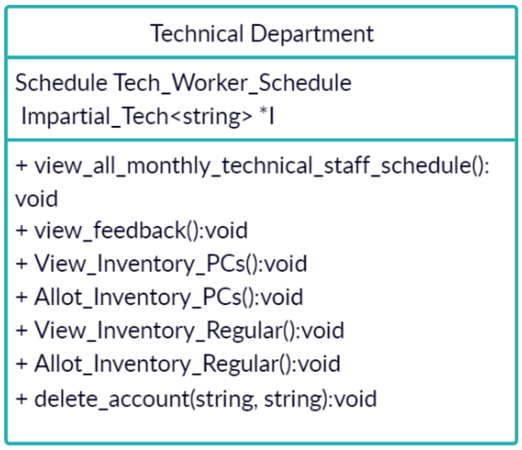
1. The Person Class carries the fundamental details for each participant in the Management System.

2. Every data is filed individually and whenever the data is meant to printed, there is a Visual display assisting the details of the particpant.

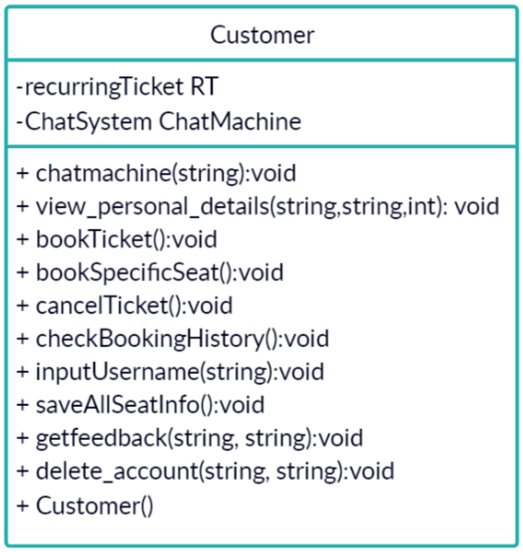
3. Also supports deletion of all personal data.

1. Short Class for Customer-Without_Account Access.
2. Allows Customer to view ticketing information, book a ticket and cancel a ticket, however cancellation is only legitimately usable during the runtime of the program.



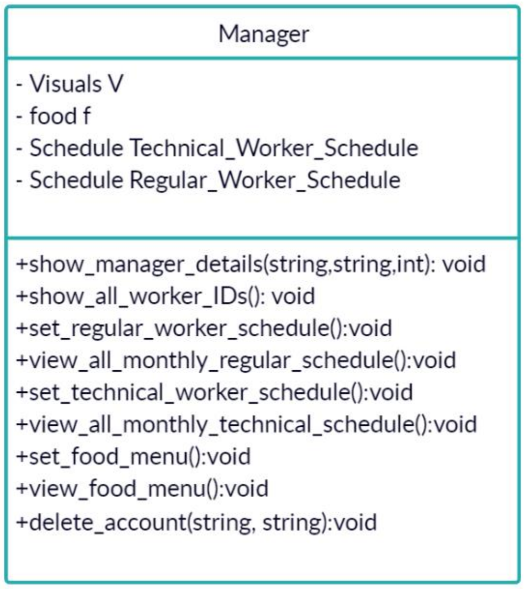

Technical Department
Schedule Tech_Worker_Schedule
Impartial_Tech<string> *I
+ view_all_monthly_technical_staff_schedule(): void
+ view_feedback():void
+ View_Inventory_PCs():void
+ Allot_Inventory_PCs():void
+ View_Inventory_Regular():void
+ Allot_Inventory_Regular():void
+ delete_account(string, string):void

1. Technical Department Class allows the user to set the stock for technical equipment, view their assigned schedule.
2. Technical Department Employee can view Personal Details as inherited by the Person Class.
3. They can also view technical feedback.



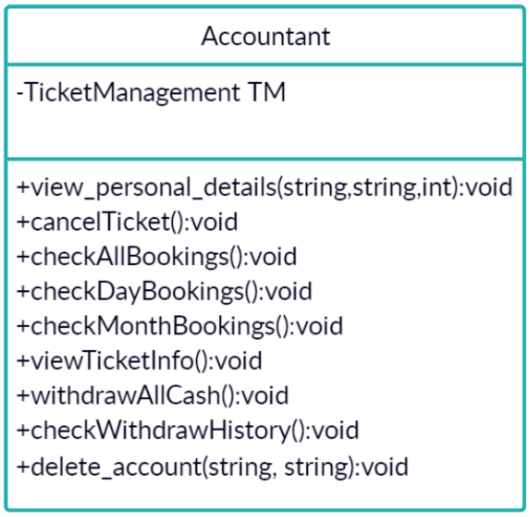
```
Customer
-recurringTicket RT
-ChatSystem ChatMachine
+ chatmachine(string):void
+ view_personal_details(string,string,int): void
+ bookTicket():void
+ bookSpecificSeat():void
+ cancelTicket():void
+ checkBookingHistory():void
+ inputUsername(string):void
+ saveAllSeatInfo():void
+ getfeedback(string, string):void
+ delete_account(string, string):void
+ Customer()
```

1. Customer can view Personal Details.
2. Can book ticket for seats, also separate option to book <u>specific</u> seats.
3. Ability to cancel ticket.
4. Ability to give feedback.
5. Ability to Check Booking History.



```
Manager
- Visuals V
- food f
- Schedule Technical_Worker_Schedule
- Schedule Regular_Worker_Schedule
+show_manager_details(string,string,int): void
+show_all_worker_IDs(): void
+set_regular_worker_schedule():void
+view_all_monthly_regular_schedule():void
+set_technical_worker_schedule():void
+view_all_monthly_technical_schedule():void
+set_food_menu():void
+view_food_menu():void
+delete_account(string, string):void
```

1. Ability to view Personal Details.
2. Ability to set schedules for Technical Department Personnel and Service Department Personnel.
3. Ability to set the food menu.
4. Ability to view All Employee IDs.



```
Accountant
-TicketManagement TM
+view_personal_details(string,string,int):void
+cancelTicket():void
+checkAllBookings():void
+checkDayBookings():void
+checkMonthBookings():void
+viewTicketInfo():void
+withdrawAllCash():void
+checkWithdrawHistory():void
+delete_account(string, string):void
```

1. Ability to view Personal Details.

2. Authority to Cancel Ticket.

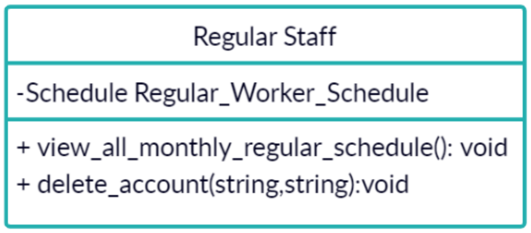
3. Ability to check the Day’s/Month’s/All-Time Bookings.

4. Ability to View Ticket Info.

5. Ability to Withdraw Cash, and check Withdrawn Cash History.

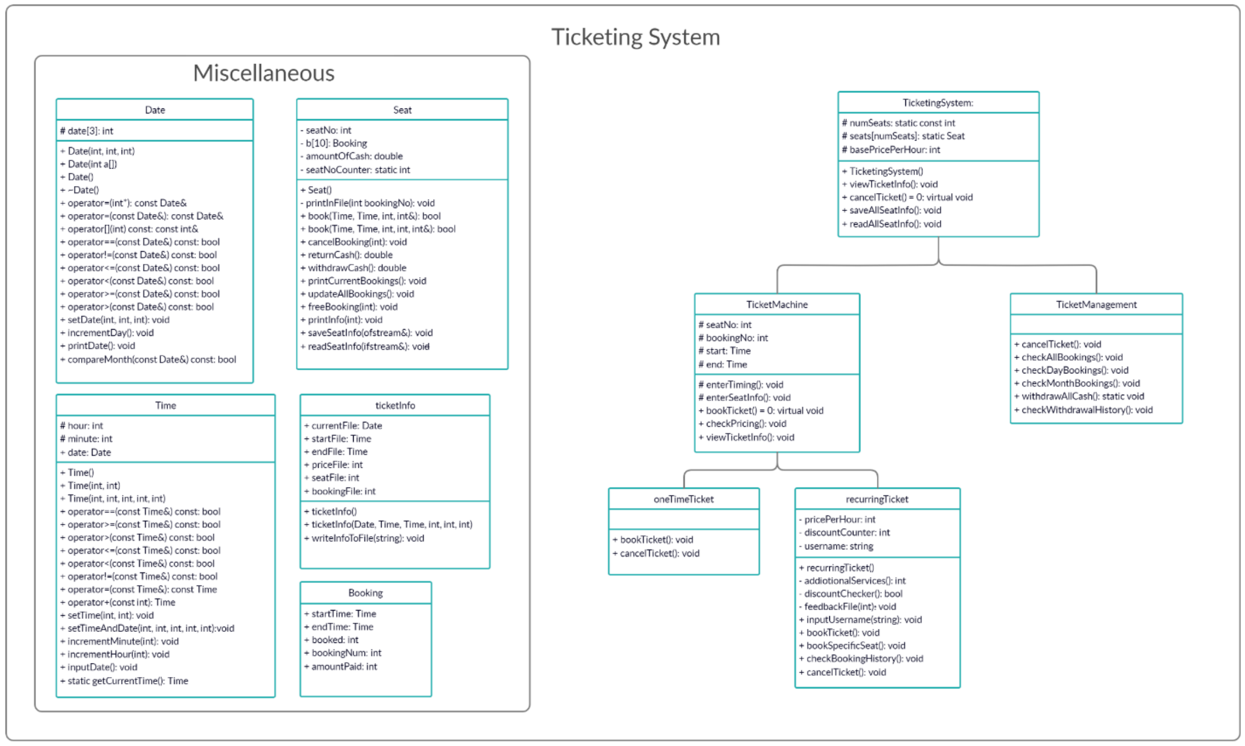
1. Ability to view Personal Details.

2. Ability to view Schedule set by the Managerial Class.



```
Regular Staff
-Schedule Regular_Worker_Schedule
+ view_all_monthly_regular_schedule(): void
+ delete_account(string,string):void
```

<u>MAIN CLASSES PART II:</u>



```
Date
# date[3]: int
+ Date(int, int, int)
+ Date(int a[])
+ Date()
+ ~Date()
+ operator=(int*): const Date&
+ operator=(const Date&): const Date&
+ operator[](int) const: const int&
+ operator==(const Date&) const: bool
+ operator!=(const Date&) const: bool
+ operator<=(const Date&) const: bool
+ operator<(const Date&) const: bool
+ operator>=(const Date&) const: bool
+ operator>(const Date&) const: bool
+ setDate(int, int, int): void
+ incrementDay(): void
+ printDate(): void
+ compareMonth(const Date&) const: bool
```

```
Seat
- seatNo: int
- b[10]: Booking
- amountOfCash: double
- seatNoCounter: static int
+ Seat()
- printInFile(int bookingNo): void
+ book(Time, Time, int, int&): bool
+ book(Time, Time, int, int, int&): bool
+ cancelBooking(int): void
+ returnCash(): double
+ withdrawCash(): double
+ printCurrentBookings(): void
+ updateAllBookings(): void
+ freeBooking(int): void
+ printInfo(int): void
+ saveSeatInfo(ofstream&): void
+ readSeatInfo(ifstream&): void
```

```
Time
# hour: int
# minute: int
+ date: Date
+ Time()
+ Time(int, int)
+ Time(int, int, int, int, int)
+ operator==(const Time&) const: bool
+ operator>=(const Time&) const: bool
+ operator>(const Time&) const: bool
+ operator<=(const Time&) const: bool
+ operator<(const Time&) const: bool
+ operator!=(const Time&) const: bool
+ operator=(const Time&): const Time
+ operator+(const int): Time
+ setTime(int, int): void
+ setTimeAndDate(int, int, int, int, int):void
+ incrementMinute(int): void
+ incrementHour(int): void
+ inputDate(): void
+ static getCurrentTime(): Time
```

```
ticketInfo
+ currentFile: Date
+ startFile: Time
+ endFile: Time
+ priceFile: int
+ seatFile: int
+ bookingFile: int
+ ticketInfo()
+ ticketInfo(Date, Time, Time, int, int, int)
+ writeInfoToFile(string): void
```

```
Booking
+ startTime: Time
+ endTime: Time
+ booked: int
+ bookingNum: int
+ amountPaid: int
```

```
TicketingSystem:
# numSeats: static const int
# seats[numSeats]: static Seat
# basePricePerHour: int
+ TicketingSystem()
+ viewTicketInfo(): void
+ cancelTicket() = 0: virtual void
+ saveAllSeatInfo(): void
+ readAllSeatInfo(): void
```

```
TicketMachine
# seatNo: int
# bookingNo: int
# start: Time
# end: Time
# enterTiming(): void
# enterSeatInfo(): void
+ bookTicket() = 0: virtual void
+ checkPricing(): void
+ viewTicketInfo(): void
```

```
TicketManagement
+ cancelTicket(): void
+ checkAllBookings(): void
+ checkDayBookings(): void
+ checkMonthBookings(): void
+ withdrawAllCash(): static void
+ checkWithdrawalHistory(): void
```

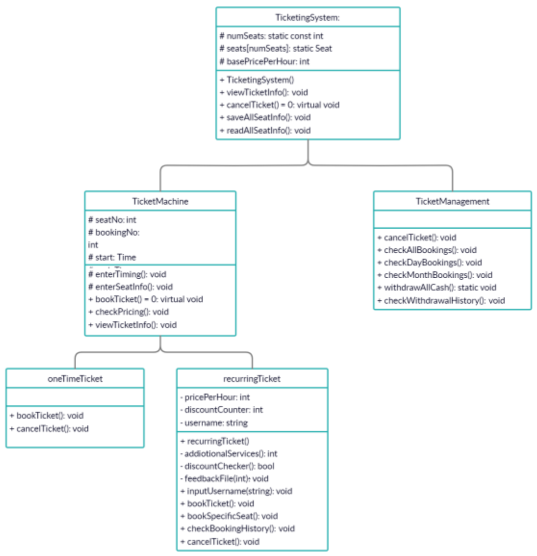
```
oneTimeTicket
+ bookTicket(): void
+ cancelTicket(): void
```

```
recurringTicket
- pricePerHour: int
- discountCounter: int
- username: string
+ recurringTicket()
- addiotionalServices(): int
- discountChecker(): bool
- feedbackFile(int): void
+ inputUsername(string): void
+ bookTicket(): void
+ bookSpecificSeat(): void
+ checkBookingHistory(): void
+ cancelTicket(): void
```

The Ticketing System is a system that keeps track of all the seats and all the bookings made for those seats. It is the system through which customers will book their seats as well as the Café staff will monitor and manage the seats and bookings.

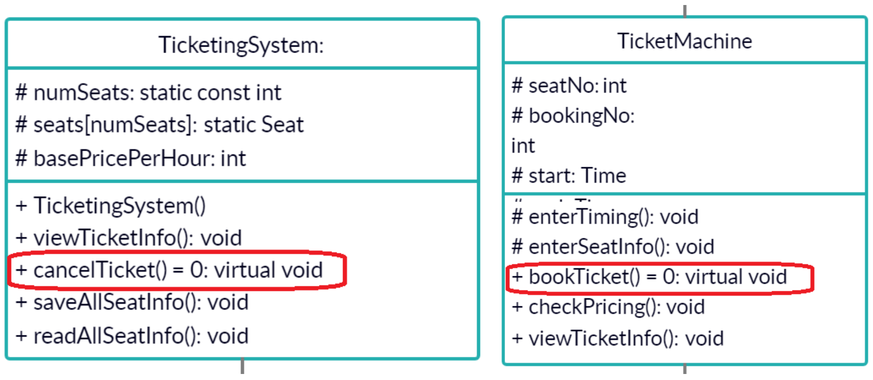
OOP Concepts Used:

Multilevel Inheritance:


TicketingSystem:
# numSeats: static const int
# seats[numSeats]: static Seat
# basePricePerHour: int
+ TicketingSystem()
+ viewTicketInfo(): void
+ cancelTicket() = 0: virtual void
+ saveAllSeatInfo(): void
+ readAllSeatInfo(): void
TicketMachine
# seatNo: int
# bookingNo:
int
# start: Time
# enterTiming(): void
# enterSeatInfo(): void
+ bookTicket() = 0: virtual void
+ checkPricing(): void
+ viewTicketInfo(): void
TicketManagement
+ cancelTicket(): void
+ checkAllBookings(): void
+ checkDayBookings(): void
+ checkMonthBookings(): void
+ withdrawAllCash(): static void
+ checkWithdrawalHistory(): void
oneTimeTicket
+ bookTicket(): void
+ cancelTicket(): void
recurringTicket
- pricePerHour: int
- discountCounter: int
- username: string
+ recurringTicket()
- addiotionalServices(): int
- discountChecker(): bool
- feedbackFile(int): void
+ inputUsername(string): void
+ bookTicket(): void
+ bookSpecificSeat(): void
+ checkBookingHistory(): void
+ cancelTicket(): void

In the first level, only the attributes and functions used throughout the system are defined. In the second level, more interactive systems are implemented. Finally, in the last level, the functions and attributes relating directly to the user are implemented.

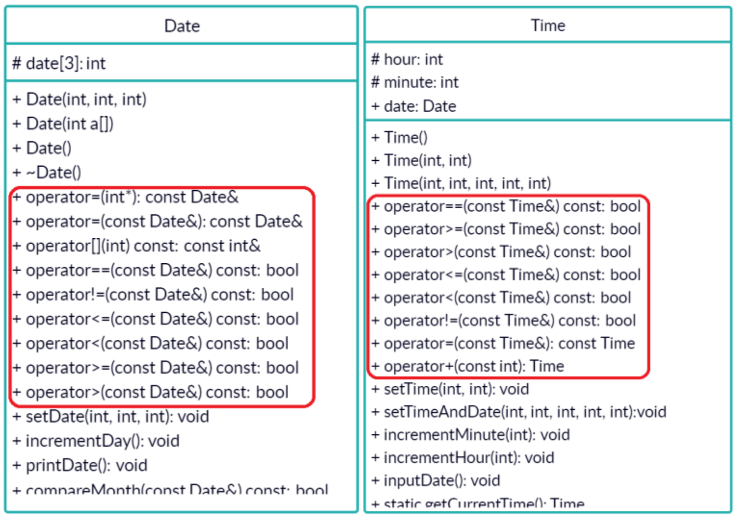
Pure Virtual Functions and Abstract Classes:



These pure virtual functions are used to connect the derived classes together. They are defined in the derived classes.

Due to the pure virtual functions, these classes are Abstract. They are not used directly in the system. Only interaction occurs in their derived classes which use their attributes and functionality.

Operator Overloading:


Date
# date[3]: int
+ Date(int, int, int)
+ Date(int a[])
+ Date()
+ ~Date()
+ operator=(int*): const Date&
+ operator=(const Date&): const Date&
+ operator[](int) const: const int&
+ operator==(const Date&) const: bool
+ operator!=(const Date&) const: bool
+ operator<=(const Date&) const: bool
+ operator<(const Date&) const: bool
+ operator>=(const Date&) const: bool
+ operator>(const Date&) const: bool
+ setDate(int, int, int): void
+ incrementDay(): void
+ printDate(): void
+ compareMonth(const Date&) const: bool
Time
# hour: int
# minute: int
+ date: Date
+ Time()
+ Time(int, int)
+ Time(int, int, int, int, int)
+ operator==(const Time&) const: bool
+ operator>=(const Time&) const: bool
+ operator>(const Time&) const: bool
+ operator<=(const Time&) const: bool
+ operator<(const Time&) const: bool
+ operator!=(const Time&) const: bool
+ operator=(const Time&): const Time
+ operator+(const int): Time
+ setTime(int, int): void
+ setTimeAndDate(int, int, int, int, int):void
+ incrementMinute(int): void
+ incrementHour(int): void
+ inputDate(): void
+ static getCurrentTime(): Time

The comparison operators as well as a few other operators are overloaded for Data and Time classes to aid in the required processes in the system. Tickets require rigorous time management therefore; operator overloading is a necessity.

Function Overriding:

viewTicketInfo() function in the base class is overridden in the TicketMachine class as the function is defined and works a bit differently in the customer side of the system as compared to the management side.

Time Keeping in the Ticketing System:

An essential feature of the Ticketing System is the time keeping. It is used to set time limits on bookings, check if bookings are active, check if specific time slot is available, free a time slot if the booking time has ended, book a seat in the future without conflicts with other bookings, and so on. It is also used outside the ticketing system in the feedback and chat classes.

It functions by first extracting the current time from your computer's clock. The code used for that is quite simple. It uses the ctime library to extract the absolute time and then convert it to a format which is understandable to us. Then the Hour, minute, day, month and year are sent to the Time/Date class to be put in a format easily usable by the system. getCurrentTime() works as follows:

> time( ) function of ctime extracts absolute time

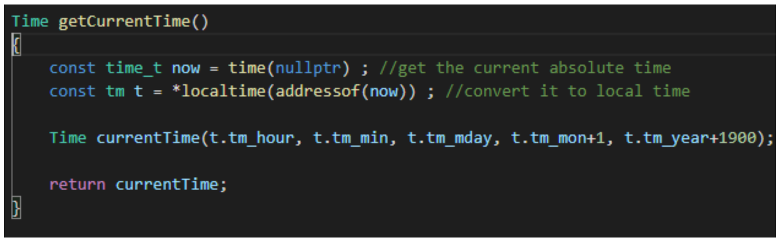
> convert it using localtime( ) function to get hour, min, day, month, year values

> send the hour, minute, day, month, year values to Time class object's constructor

> return the Time class object

This Time class object will contain the current time of your computer in the:

HH:MM - DD/MM/YYYY format.



```
Time getCurrentTime()
{
    const time_t now = time(nullptr) ; //get the current absolute time
    const tm t = *localtime(addressof(now)) ; //convert it to local time

    Time currentTime(t.tm_hour, t.tm_min, t.tm_mday, t.tm_mon+1, t.tm_year+1900);

    return currentTime;
}
```

Now the comparison operators in Time class are used to compare the inputted time or the time at which a booking is starting and ending etc. All of these overloaded operators work as follows:

> Compare year values.

> If year values return false, compare month values

> If month values return false, compare day values

> If day values return false, compare hour values

> If hour values return false, compare minute values

> If at any of the above steps, true is returned, the whole overloaded operator returns true. Else it returns false.

This way all time values are comparable with each other. Whether they are from your computer's time, an inputted time, a time calculated by the program etc.

The incrementHour, minute, day, month, year functions will increment the value as much as the parameter specifies. So, if a certain value is added to a Time object by the program, it will affect other values. For example, if the time is 22:31 and the program adds 30 minutes to it, it becomes 23:01 rather than 22:01 this is then carried on with Day, month and year values. Minute Increment function works as follows: (all subsequent value functions are dealt with in the same way)

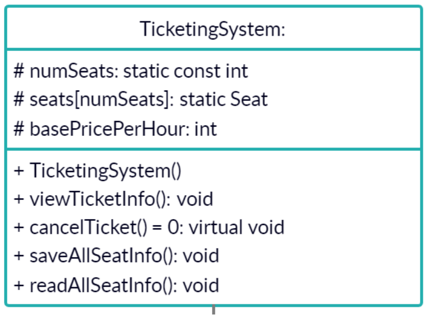
> Minute += Received Minutes

> If Minute value is greater than 59

> Call Increment Hour function with incrementation 1. (1 sent as parameter)

Short Explanation of Each Class:

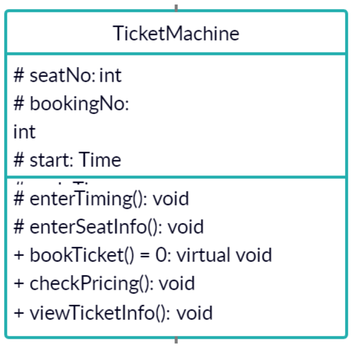
<u>TicketingSystem:</u>


TicketingSystem:
# numSeats: static const int
# seats[numSeats]: static Seat
# basePricePerHour: int
+ TicketingSystem()
+ viewTicketInfo(): void
+ cancelTicket() = 0: virtual void
+ saveAllSeatInfo(): void
+ readAllSeatInfo(): void

This is the Base class of this whole system. It is an abstract class as all the functionality is done by objects of base class. It provides the following functionalities:

- Save all seat information to file.
- Read all seat information from file.
- View a particular ticket information.
- Keeping track of all the seat attributes and conditions.

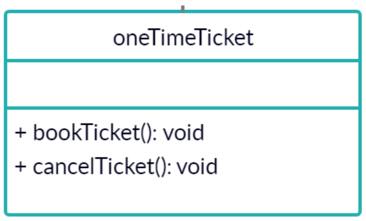
TicketMachine:



This class is derived from TicketingSystem. This is the class whose derived classes manage all the ticket booking by the customer. Therefore, it only has functions relevant to a customer like booking a ticket etc. It provides the following functionalities:

- Check price per hour and premium feature prices.
- Choose a time slot.
- Select seat number and booking number.

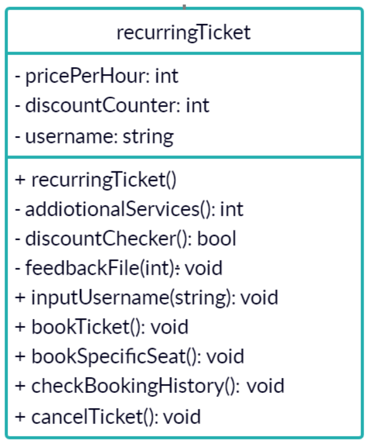
oneTimeTicket (system):



This class is derived from TicketMachine. The object of this class will let a customer book a ticket without an account I.e. a one-time customer who has no association with the café. This class provides only the bare bones functions of booking a seat and viewing prices as to give incentive to customers to create an account with our store. It provides the following functionalities:

- Book a single ticket without an account.
- Check cancellation method.

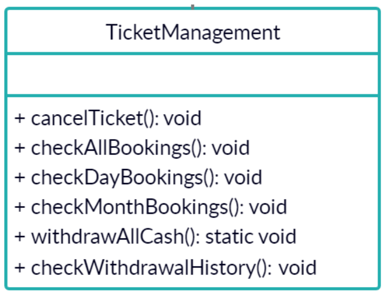
recurringTicket (system):


recurringTicket
- pricePerHour: int
- discountCounter: int
- username: string
+ recurringTicket()
- addiotionalServices(): int
- discountChecker(): bool
- feedbackFile(int): void
+ inputUsername(string): void
+ bookTicket(): void
+ bookSpecificSeat(): void
+ checkBookingHistory(): void
+ cancelTicket(): void

This class is derived from TicketMachine. The object of this class will provide a platform for the recurring customer i.e. Customer with an Account. It provides discounts, deals, extra perks for the customer and the ability to check your booking history. It provides the following functionalities:

- Choose premium services.
- Check for discounts.
- Book multiple randomly selected seats.
- Book a specific seat from our café layout.
- View your ticket booking history.
- Cancel a specific ticket booking.
- Write all the tickets information to a file which will be used in Feedback class.

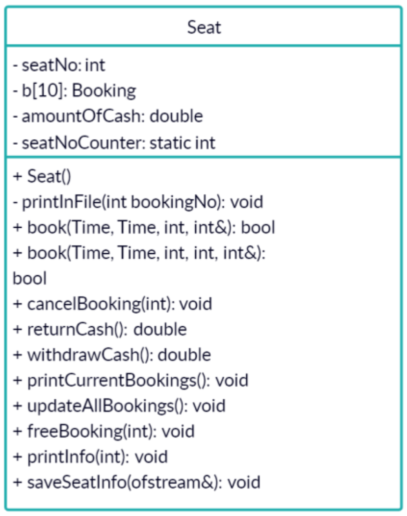
TicketManagement (system):


TicketManagement
+ cancelTicket(): void
+ checkAllBookings(): void
+ checkDayBookings(): void
+ checkMonthBookings(): void
+ withdrawAllCash(): static void
+ checkWithdrawalHistory(): void

This class is derived from TicketingSystem. This system allows the staff of the café to monitor and change the settings for seats as well as view history of the money gained from the seats. It allows staff to view all the tickets, tickets on a particular day or a particular month. It also has functions to view a specific ticket or delete it in case of emergency or maintenance during that time. It provides the following functionalities:

- Cancel any ticket any customer has booked.
- Check all the booking history.
- Check a specific date’s booking history.
- Check a specific month’s booking history.
- Write all the money made from seats to a file along with the current date.
- Check the file to see how much money has been made on which days.

Seat:


Seat
- seatNo: int
- b[10]: Booking
- amountOfCash: double
- seatNoCounter: static int
+ Seat()
- printInFile(int bookingNo): void
+ book(Time, Time, int, int&): bool
+ book(Time, Time, int, int, int&): bool
+ cancelBooking(int): void
+ returnCash(): double
+ withdrawCash(): double
+ printCurrentBookings(): void
+ updateAllBookings(): void
+ freeBooking(int): void
+ printInfo(int): void
+ saveSeatInfo(ofstream&): void

The Seat class is responsible for all the functions of a single seat. The main system consists of 25 of these Seat objects in an array. This class provides functions to book the seat for specific times and it saves the timings and bookings through filing. Further functions include cancelling of the particular seat. Keeping track of how much money has been earned through the particular seat. And various functions to manage the seat by the staff. It keeps track of the time when seats are booked and if a particular seat is free at another time if it is booked right now. It also periodically refreshes it’s booking to see which time slots have become free through cancelling by customer or the time of that booking passing.

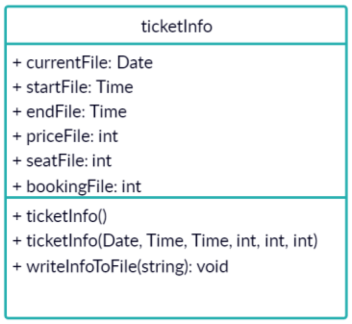
Time:

This is the class provides the functionality of time keeping relevant to our specific projects. The function and operator overloading in this class provides us with the ability to compare and calculate the specific time for seat bookings such that no clashes occur.

Date:

This class works hand in hand with the time class as it has “has-a” relationship with Time class. It also has operator and function overloading to provide the specific functionality needed for this project.

TicketInfo:

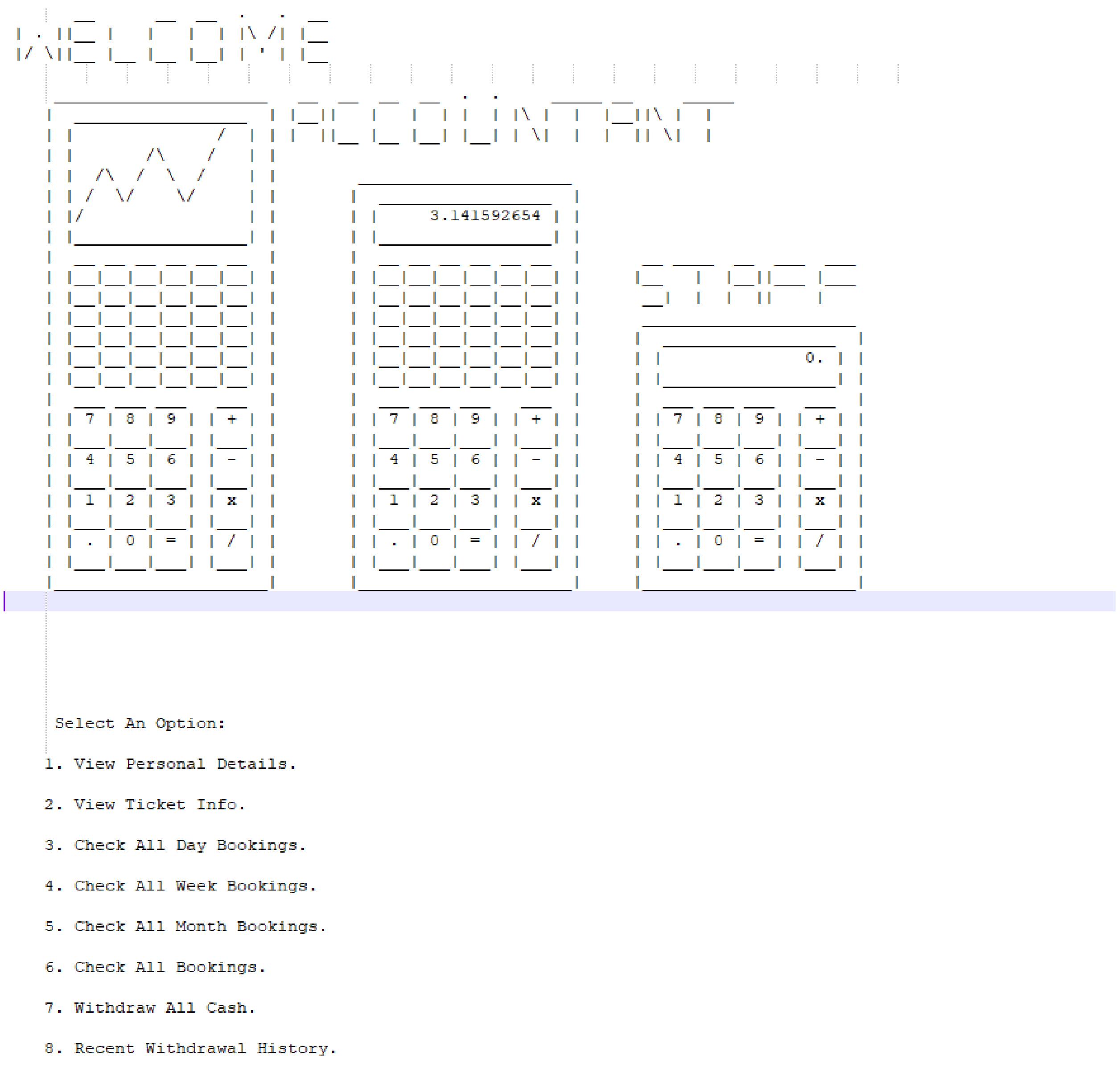


Objects of this class store the information of a particular ticket booked by the customer. It allows the system to keep track of all the tickets, and view the history of tickets for both customers and staff. Its main function is to write the information of the tickets to a file through object writing.

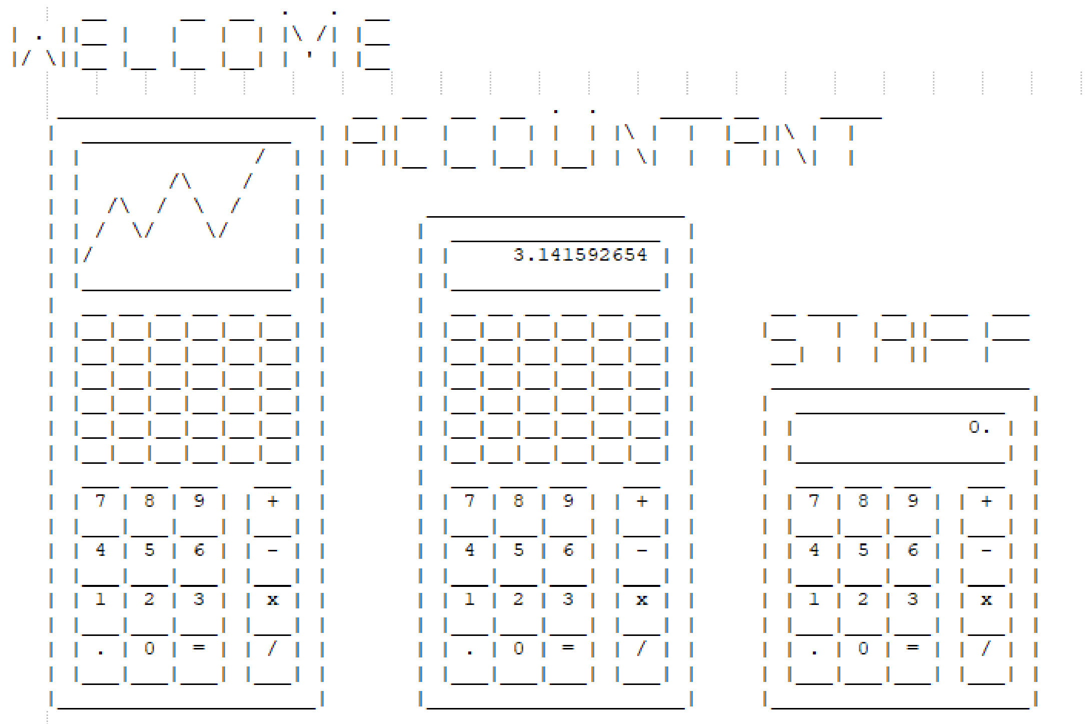
Extensive Explanation of Particular Functionality:

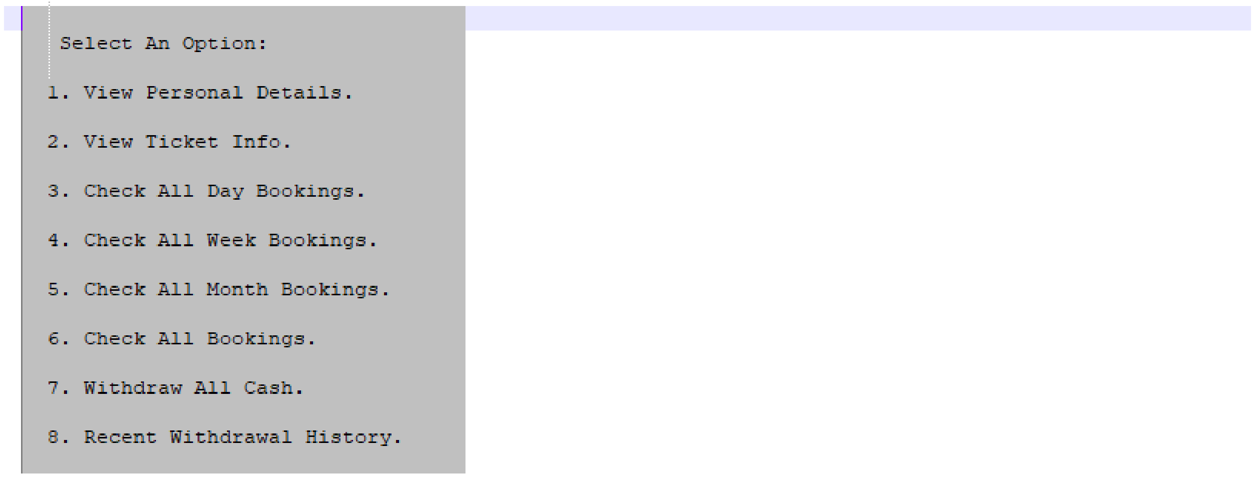
Visuals:

By abusing the vertical selection of notepad++, we can use the backwards compatibility and then read the text in the notepad character by character:



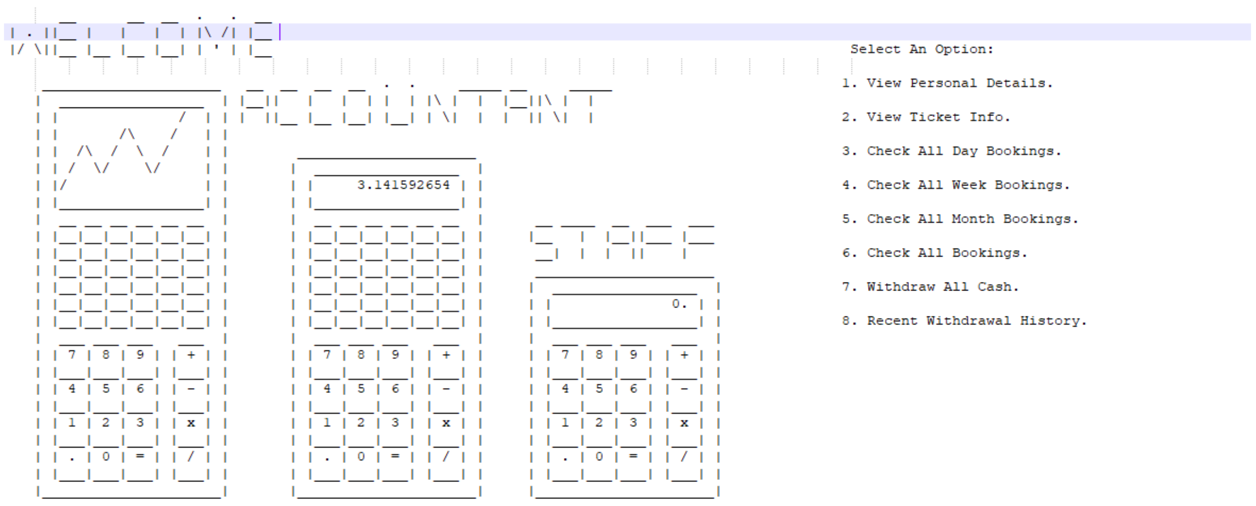
```
Select An Option:

1. View Personal Details.

2. View Ticket Info.

3. Check All Day Bookings.

4. Check All Week Bookings.

5. Check All Month Bookings.

6. Check All Bookings.

7. Withdraw All Cash.

8. Recent Withdrawal History.
```





```
 Select An Option:
1. View Personal Details.
2. View Ticket Info.
3. Check All Day Bookings.
4. Check All Week Bookings.
5. Check All Month Bookings.
6. Check All Bookings.
7. Withdraw All Cash.
8. Recent Withdrawal History.
```

ALT+CLICK DRAG.



```
Select An Option:

1. View Personal Details.

2. View Ticket Info.

3. Check All Day Bookings.

4. Check All Week Bookings.

5. Check All Month Bookings.

6. Check All Bookings.

7. Withdraw All Cash.

8. Recent Withdrawal History.
```

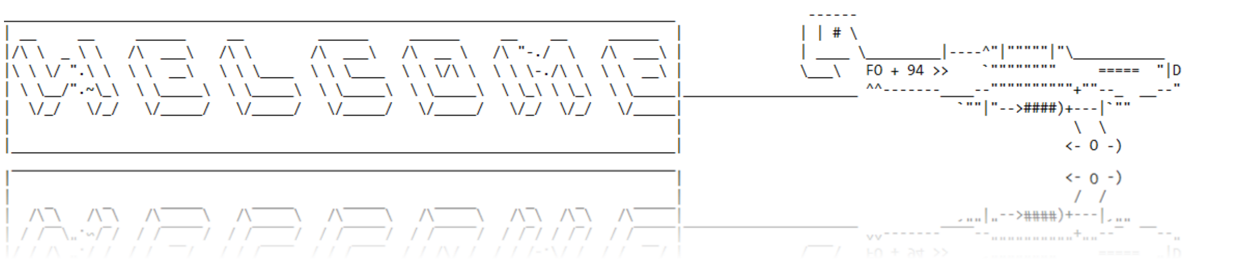
ALT+CLICK (place cursor where you want to put the text)

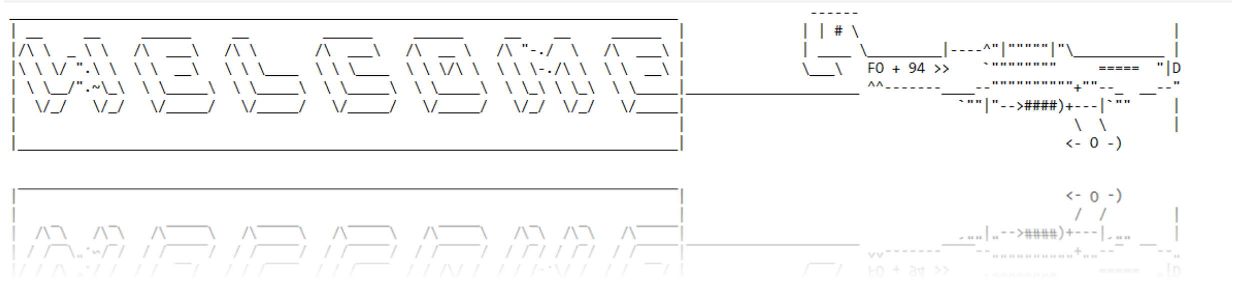
Copy paste the entire text on notepad and then read it character by character.

Animated Visuals:

Plane Example:

Two alternating visuals to give illusion of moving fan:





Positioning logic:

At time=0:

x position ⟶

y position ↓

Plane							

At time=0.01s:

x position →

y position ↓

⟶⟶⟶

↑ ↓

~~Plane~~ Has to be erased	Plane						

We can make a custom gotoXY function to read every line in the notepad document as a string using getline(filename, tempstring, '\n') then print the entire visual by incrementing the y position as each line is printed, the animation happens when the two visuals alternate and the position of X is changed while at the same time erasing the previous iteration of the visual on the previous position of X.

Usage of Templates:

Note: Assume template<class U> has been used.

Templates used with the Class Impartial Tech, and subsequently used in it's derived children. Impartial Tech refers to technological inventory irrespective of type while the derived classes have concrete types.

If the reference to inventory type exists then it can coexist as both integer and string as some products exist in numerical code.

If the member "type" was declared as "void *type;"

Inside the setter function we could do the following:

U *ptr=new U[10]; //10 here is an example digit

For(1→10)
getline(cin,ptr[i]); (if ptr is an integer we can make an exception overloaded function)

And finally we can have the type pointer point to ptr such as:

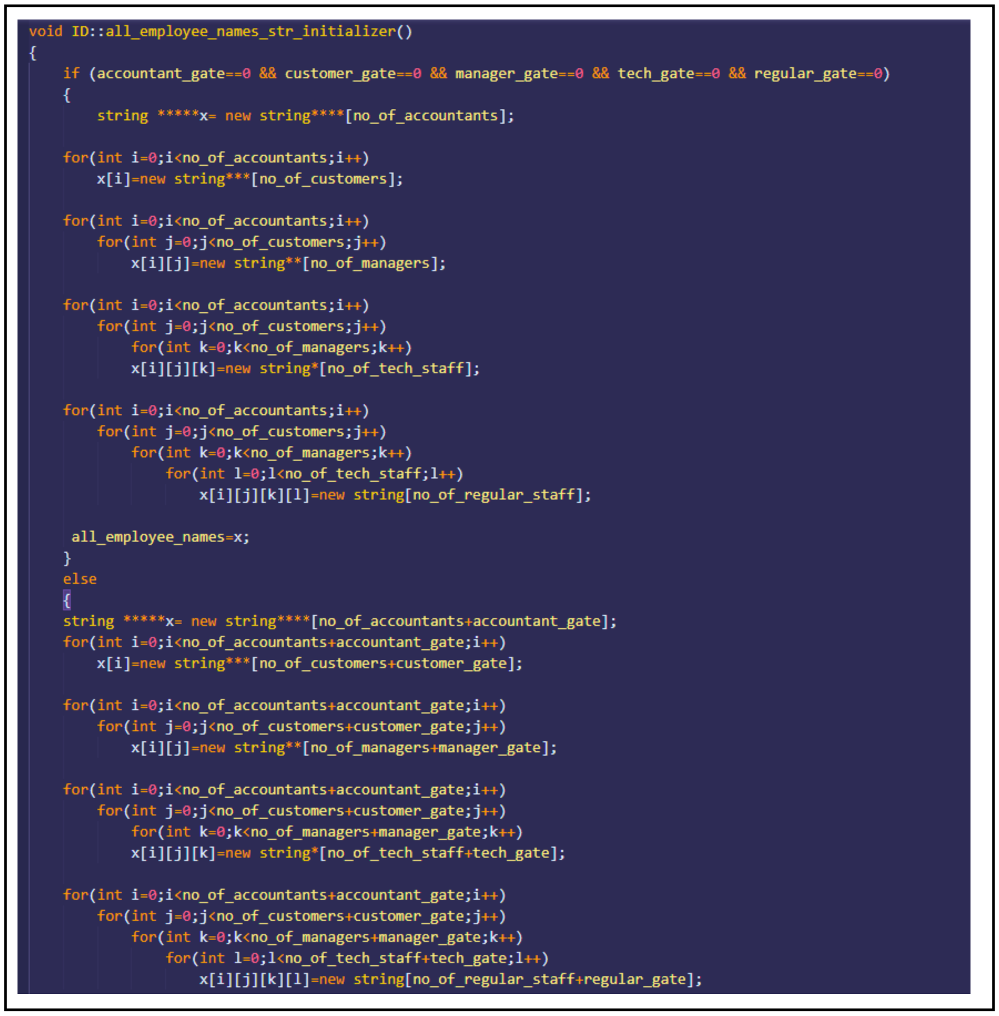
type=reinterpret_cast<U*> (ptr);

However to use to dereference each element separately we have to use the code as the following:
((U)(Impartial_Tech<U>::type)+(ELEMENT DIGIT);

Usage of Complex Pointers:

Example: Class ID
string *****all_employee_names;

Inside setter function we can declare spaces as such:



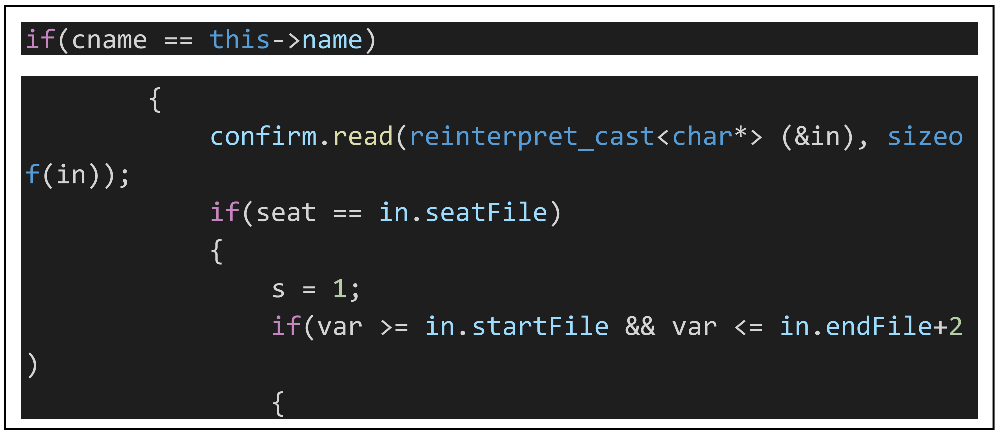
```
void ID::all_employee_names_str_initializer()
{
    if (accountant_gate==0 && customer_gate==0 && manager_gate==0 && tech_gate==0 && regular_gate==0)
    {
        string *****x= new string****[no_of_accountants];

    for(int i=0;i<no_of_accountants;i++)
        x[i]=new string***[no_of_customers];

    for(int i=0;i<no_of_accountants;i++)
        for(int j=0;j<no_of_customers;j++)
            x[i][j]=new string**[no_of_managers];

    for(int i=0;i<no_of_accountants;i++)
        for(int j=0;j<no_of_customers;j++)
            for(int k=0;k<no_of_managers;k++)
            x[i][j][k]=new string*[no_of_tech_staff];

    for(int i=0;i<no_of_accountants;i++)
        for(int j=0;j<no_of_customers;j++)
            for(int k=0;k<no_of_managers;k++)
                for(int l=0;l<no_of_tech_staff;l++)
                    x[i][j][k][l]=new string[no_of_regular_staff];

     all_employee_names=x;
    }
    else
    {
    string *****x= new string****[no_of_accountants+accountant_gate];
    for(int i=0;i<no_of_accountants+accountant_gate;i++)
        x[i]=new string***[no_of_customers+customer_gate];

    for(int i=0;i<no_of_accountants+accountant_gate;i++)
        for(int j=0;j<no_of_customers+customer_gate;j++)
            x[i][j]=new string**[no_of_managers+manager_gate];

    for(int i=0;i<no_of_accountants+accountant_gate;i++)
        for(int j=0;j<no_of_customers+customer_gate;j++)
            for(int k=0;k<no_of_managers+manager_gate;k++)
            x[i][j][k]=new string*[no_of_tech_staff+tech_gate];

    for(int i=0;i<no_of_accountants+accountant_gate;i++)
        for(int j=0;j<no_of_customers+customer_gate;j++)
            for(int k=0;k<no_of_managers+manager_gate;k++)
                for(int l=0;l<no_of_tech_staff+tech_gate;l++)
                    x[i][j][k][l]=new string[no_of_regular_staff+regular_gate];
```

HOWEVER, the complex ID must also be able to perform functions of appending AND also it is important to note none of the data can lie on the array position [0,0,0,0,0], as any data starting from that position would be overwritten by the newest data.

To solve the problem of appending, the last entry digit in each category of personnel is saved in a variable, while the problem of the 0th position is solved by making every data start from the 1st position, however that means all the data that's saved has to be of element position +1, even during filing. This is incredibly complex and hard to keep track of and was only used because the Complex ID was designed prior to filing, otherwise the direct functionality of appending would have been covered by the file pointers.

Usage of current time to operate and manage feedbacks and Chats

As aforementioned, a "time" class has been used to make the feedback and chat system fully flexible and strict. It is so much common to give or submit feedback after the usage period id user ends, to implement the following the logic, a custom time and date class has been used. Whenever user wants to submit feedback, a global function named "GetCurrentTime()" has been used to check the current time with the user's time period. As said earlier, User can give feedback for upto 2 hours after the end time. The following logic has been used:



```
if(cname == this->name)

        {
            confirm.read(reinterpret_cast<char*> (&in), sizeo
f(in));
            if(seat == in.seatFile)
            {
                s = 1;
                if(var >= in.startFile && var <= in.endFile+2
)
                {
```

It is basically the whole process for a person to pass the requirements to give the feedback where, at first, it will check the user name with the name on the record file, if the name matches, it will then go to check the seats and then the time (startFile is the start timing of user whereas endFile is the end time). They are being checked in a standard pattern used to standardize the file with all the records. The end time (in this case, endFile) has been increased by 2 so user can enter his feedback even after end time for up to 2 hours.

Same type of system has been used to make chatting system running smoothly with all the users. The process of validating the user to use the chat system is very much same as the feedback validating setting, the only difference is just that the end time has not been increased. Although, time is not only used to meet the requirements to chat but it is also playing an important role in low memory consumption, as time is the main source which is clearing the chat history files after the user chatting period is over or when the new user takes the seat, so he will be given by a fully refreshed file with zero memory. If a user was previously eligible or going to eligible in future with this seat, then the system will show the user’s all previous interaction with this particular seats. Here is the glimpse of code,

```
 while(!infile.eof())

    {
        getline(infile, check, '\0');

        if(infile.eof())
            break;

        if(check == name)
        {
            infile.read(reinterpret_cast<char*> (&in), sizeof(in)
);
            if(check_s == in.seatFile)
            {
                if(var < in.startFile || var > in.endFile)
                {
```

As you can see, it will open, extract and show all previous record of particular user with the particular seats.

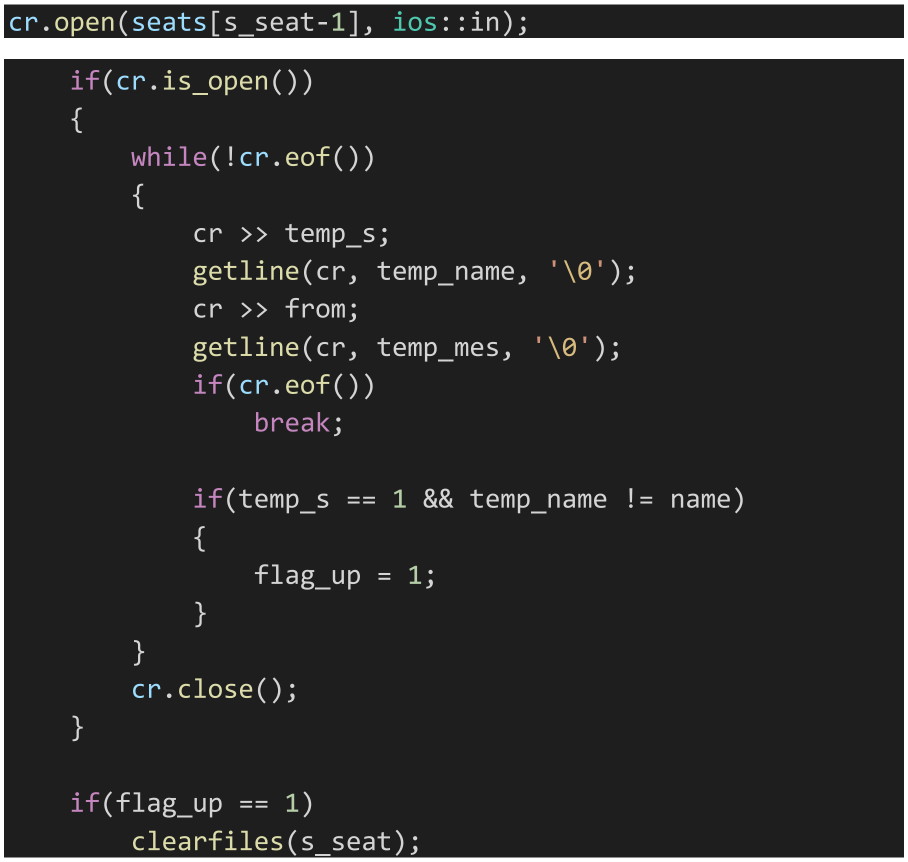
To ensure low consumption of memory, time is playing an important role in clearing those files. There are basically two functions which are working on the clearing files, named as “clearfiles_support()” and second as “clearfiles(int)”, the support function is activated automatically whenever a new user signed in for the seat, it can happens sometime that a user maybe signed out and then signed in again, the support function will check if the new signed in user is either same as previous ones or not, if it is the previous ones, the chat history of that user will not be remove or delete. If the user is not same, it will send a flagged high to the clear function which it will then clear all the file. Here is the little glimpse of clearing function,

```
 ofstream clr;

    clr.open(seats[check_s-1], ios::out | ios::trunc);
    clr.close();
    actions();
```

Where **seats** is an array which is containing all the names of files.

Here is basic structure or working of the support function where it is checking if the files needs to clear or not. If the file needs to be cleared, it will then flag the clear function to do its work.

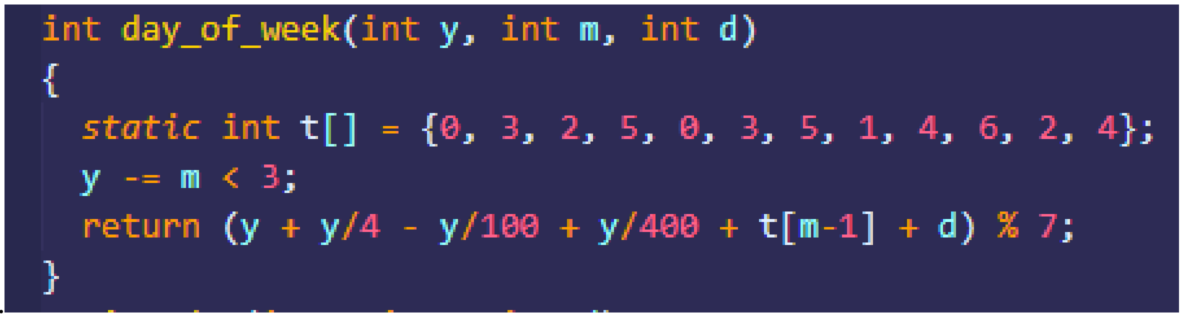
This method has been working effectively with the chat system to fully feel the working process of a real life chatting system.



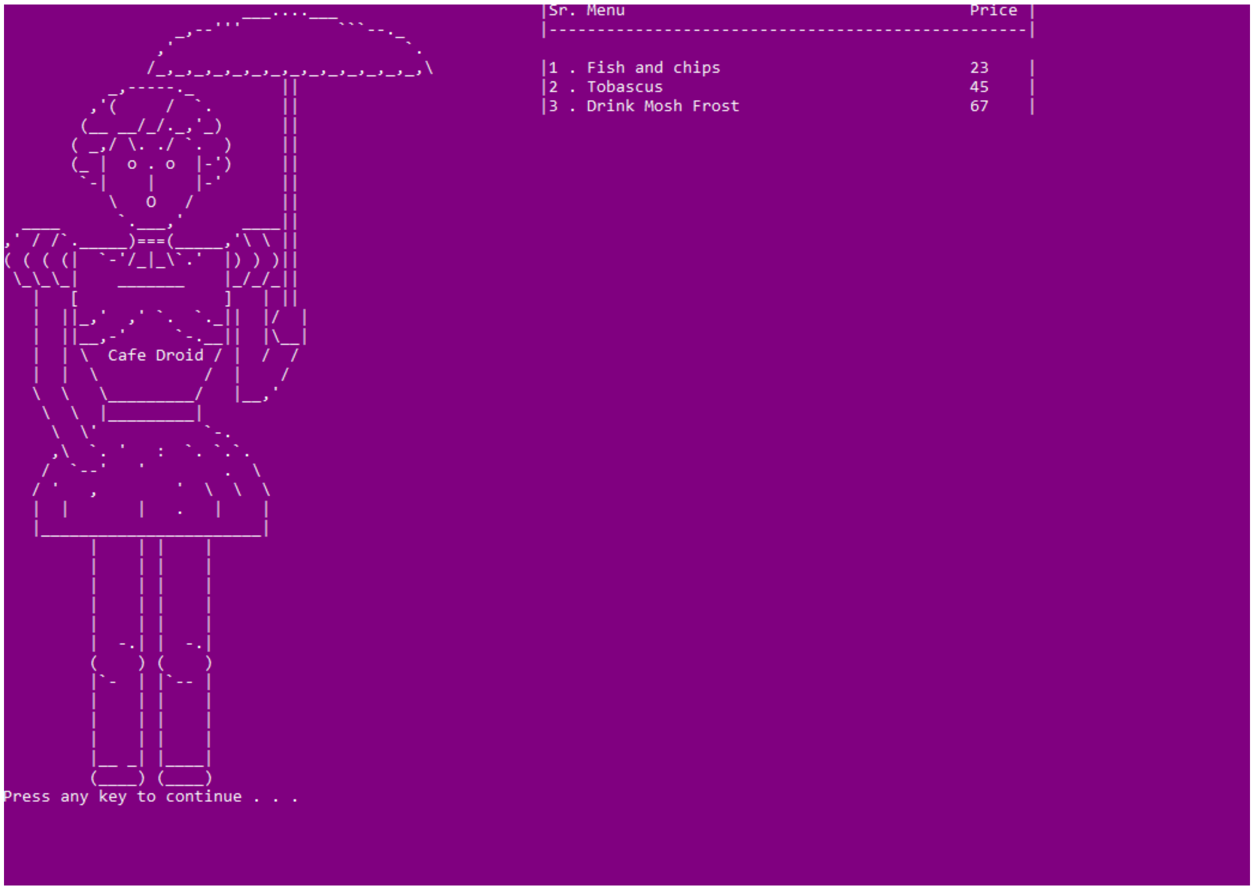
```cpp
cr.open(seats[s_seat-1], ios::in);

    if(cr.is_open())
    {
        while(!cr.eof())
        {
            cr >> temp_s;
            getline(cr, temp_name, '\0');
            cr >> from;
            getline(cr, temp_mes, '\0');
            if(cr.eof())
                break;

            if(temp_s == 1 && temp_name != name)
            {
                flag_up = 1;
            }
        }
        cr.close();
    }

    if(flag_up == 1)
        clearfiles(s_seat);
```

Usage of filing while designing food menu & schedule:

Special day function for accuracy of leap year etc.



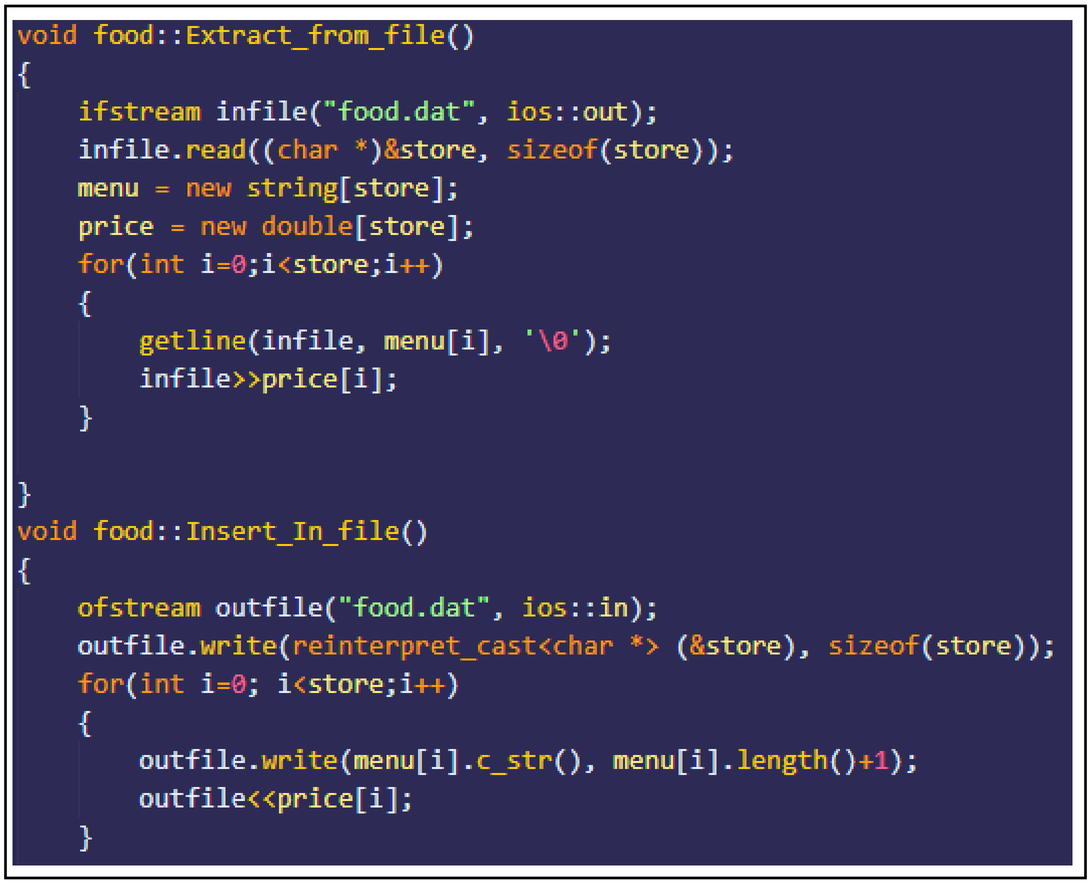
```cpp
int day_of_week(int y, int m, int d)
{
  static int t[] = {0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4};
  y -= m < 3;
  return (y + y/4 - y/100 + y/400 + t[m-1] + d) % 7;
}
```



```
|Sr. Menu                                        Price |
|------------------------------------------------------|

|1 . Fish and chips                              23    |
|2 . Tobascus                                    45    |
|3 . Drink Mosh Frost                            67    |

Press any key to continue . . .
```

The filing procedure works in both cases in the sense that it takes the array element, then saves the string/integer/characters subsequently according to the exact amount of spaces of the array element. While filing we can first pick up the elemental digit indication, then use a for loop to pick up all of the data, this in turn ensures that no extra data is picked up at the time of use.

For example:



```
void food::Extract_from_file()
{
    ifstream infile("food.dat", ios::out);
    infile.read((char *)&store, sizeof(store));
    menu = new string[store];
    price = new double[store];
    for(int i=0;i<store;i++)
    {
        getline(infile, menu[i], '\0');
        infile>>price[i];
    }

}
void food::Insert_In_file()
{
    ofstream outfile("food.dat", ios::in);
    outfile.write(reinterpret_cast<char *> (&store), sizeof(store));
    for(int i=0; i<store;i++)
    {
        outfile.write(menu[i].c_str(), menu[i].length()+1);
        outfile<<price[i];
    }
```

4. Future work/ Conclusive Statement

We would have loved to have worked on external libraries to make our Management System look more aesthetically pleasing and professional or to have made something creative like a game. As a gaming café would be more interesting if it provided the ability to play actual games rather than just being a management system. But we wanted to stick very close to the core Object Oriented Programming concepts therefore we had to sacrifice some of our creative vision. However, in the future we would love to work on something innovative and new whilst incorporating our core OOP concepts.

5. Appendix

All headers used in Project:

```
Project.h
Manager.h
RegularStaff.h
Password.h
Complex_ID.h
Person.h
Foodmenu.h
ScheduleWorker.h
Visuals.h
Seat.h
oneTimeTicket.h
recurringTicket.h
TicketingSystem.h
TicketMachine.h
TicketManagement.h
Date.h
Time.h
Impartial_Tech.h
RegularEquipment.h
CybernetComputer.h
ticketInfo.h
Accountant.h
FeedbackSystem.h
Customer.h
chatsystem.h
Tech_Staff.h
OneTimeCustomer.h
```