Стефани Арабаджиева
ф.н. 81888
30.05.2020 г.
https://github.com/sarabadjieva/Ticket-Booth-Managment

Проект „Билети”

курс Обектно-ориентирано програмиране
за специалности Информатика, Информационни системи и Компютърни науки
летен семестър 2019/2020 г.

1. Увод

Проектът представлява разработка на функционираща информационна система за управление на билетна каса. За изпълнението на проекта са използвани *Visual Studio* и езикът за програмиране C++, контейнери от стандартната библиотека с шаблони, библиотека за работа с файлове и за работа с потоци. Данните се запазват в текстов файл. Датите се изписват в международния стандарт за обмен на дати и данни за време *ISO 8601.*

Целта на проекта е да се реализира система, позволяваща улеснено управление на билетна каса чрез различни функции. Поддържат се операции за работа с билети, добавяне на събития, извеждане на справки и други.

Задачата на проекта е да предостави на потребителя различни възможности за работа с файлове, т.е. да може да се избира с кои файлове да се работи, дали новите данни да бъдат изтрити(close), запазени в същия файл(save) или друг(save as), чието име е въведено от потребителя. Другата част от задачата е проектиране на функции за работа с данните, намиращи се във файла. Те представляват: добавяне на представление; купуване, запазване на билети, както и отмяна на резервирано място; проверяване за валидност на билет; търсене на свободни места; извеждането на справка за закупени или запазени билети в даден период.

В документацията ще бъде направен преглед на предметната област (основните дефиниции, концепции и алгоритми, които са използвани; проблеми и сложност на поставената задача и методи за решаването им). Следващото нещо, което ще се разгледа, е проектирането на задачата, където са включени основните класове и връзката между тях. Приложена е таблица с името на класовете, съответните им член-данни и по-важните методи. После се преминава към реализацията и тестването на проекта, приложени са части код, обяснени са в дълбочина по-важните функции и работата на програмата. Създадени са и няколко примера, показващи некоректно въвеждане на информацията. В заключителната част са дадени идеи за бъдещо развитие на проекта и подобрението на потребителския интерфейс.

2. Преглед на предметната област

Подходът за реализиране на проекта е чрез обектно-ориентирано програмиране. Различните части от задачата се представят като обекти, чиито класове включват в себе си данни и методи(конструктори, селектори, мутатори и др.). Използвани са основните принципи абстракция, капсулация, както и рекурсия. Всички класове са дефинирани глобално. В програмата са включени алгоритми за търсене, рекурсия, динамично програмиране.

Един от проблемите е създаването на четим файл и въвеждането на информация в него. За да е разбираем за потребителя това няма как да се осъществи чрез използването на двоичен файл. В класа *Event* и *Ticket* са предефинирани операторите за вход и изход, които позволяват добре структуриран текстов файл, в който ясно са въведени данните.

Различните възможности за запазване на файл са програмирани чрез създаване на временен файл, в който се копира първоначалното съдържание. В края на програмата този файл се изтрива, а информацията в него може да се презапише в друг файл (оригиналния или нов) или да не се запазва.

Системата е удобна и скалируема, защото позволява да се работи с неограничен брой билети, зали и събития. Всеки потребител може да работи със системата без значение неговите права и роля. Той трябва да отговаря на различни въпроси или пише операции за промяна на данните, като въвежда всичко в терминала от клавиатурата. Проектът изисква обектно-ориентиран стил, за да може да се чете по-лесно и да могат да се правят промени бързо.

3. Проектиране

В проекта се съдържат четири основни класа: *Manager, Event, Ticket, Date.* Във файла *Manager.h* е реализиран запис *Hall,* който служи за представяне на данните за всяка зала – име; брой редове и брой места на всеки ред; дати, на които е заета. Класът мениджър съдържа два контейнера с обекти – един за залите и един за събитията. Всяко представление има член-данна динамичен масив от билети, където се запазват закупените и запазените такива за всяка една дата, на която се провежда това мероприятие. Билетите се идентифицират с код, в който е включено името на представлението, дата, зала, ред и номер на място. Класът дата е основен за всички функции, извършващи се от програмата.

В таблицата на следващата страница са представени накратко всички класове в проекта - техните член-данни и по-важни функции. Кратко обяснение се съдържа в полето, отдясно на тях. Стрелките посочват пътя на създаване на програмата, като най-отдолу е класът *Date,* който помага за създаването на билети и по-нататъшни операции. Над него е класът за билети, който е ключов за класа *Event.* Мениджър класа е частта, до която „достига” обикновеният потребител и работи с вход и изход на данни.

Manager
file: fstream temp: fstream hallsFile: fstream filename: char array tempName: char array hallsFileName: char array events: vector<Event*> halls: vector<Hall*>
callManager(): void readHalls(): void readEvents(): void writeEvents(): void workwithfile(): void

Класът *Manager* служи за контролиране на всички процеси и управление на потребителския вход. В него се запазват множеството от зали и представления, с които се работи. Те биват прочетени от файлове чрез функциите *readHalls(), readEvents().* При всяка промяна на данните се изтрива информацията от началния файл и се копира тази, запазена във временния файл. Той ”живее” само по време на работа с програмата. При нейното спиране автоматично се изтрива.

Event
eventname: char* shows: map<Date, char*> tickets: Ticket* ticketsSize: unsigned integer ticketsCapacity: unsigned integer
addShow(const Date&, char*): bool bookTicket(Ticket&): bool buyTicket(Ticket&): bool unbookTikcet(const Ticket&): void report(const Date&, const Date&, const char*): void

Класът *Event* представя различните представления като име; асоциативен контейнер на шоута, където ключ е датата, а стойност името на залата; динамичен масив от билети, в който се съдържат и запазените и закупените такива, големина на този масив и капацитет. Методите в този клас позволяват различна работа с горепосочените, като при добавяне на билет се правят проверки за големината и капацитета на масива.

Ticket
code: char* note: char* booked: bool bought: bool
setCodeviaParam(const Date&, const char*, const char*, const size_t, const size_t): void

Класът *Ticket* служи за създаването на обект билет. Членът *code* е уникален за всеки обект, създаден по шаблона „събитие$дата$зала$ред$място”.
Проверява се валидността на кода, дали са билетите са закупени или запазени. Създадени са функции, работещи върху низа код, които позволяват да се взима цялата нужна информация от билета.

Date
year: unsigned integer month: unsigned integer date: unsigned integer validDate: bool
daysInMonth(unsigned integer, unsigned integer): unsigned integer nextDate(): Date operator==(Date): bool operator<(const Date&): bool operator>(Date): bool getDatesBetween(Date, Date): vector<Date>

Класът *Date* съдържа основните операции, които могат да се извършват с дати и биха били полезни. Член-данна е булева, която съдържа информация относно валидността на въведената дата, т.е. дали месецът е между 0 и 12, дали денят отговаря на истинска дата. В случай, когато датата е създадена без да се подават параметри, новосъздадената дата е днешната.

4. Реализация, тестване

При стартиране на програмата потребителят трябва да избере дали иска да работи с файловете по подразбиране или с други, създадени вече. Програмата очаква вход от потребителя и след получаване на такъв - информацията се обработва. Според въведените данни се избира кой конструктор на класа *Manager* да се използва. В конструктора се извиква функция *readHalls()*, която прочита от файл предварително въведените зали и ги запазва в контейнер в обекта мениджър. След това се влиза в цикъл, позволяващ на потребителя да избере по какъв начин иска да работи с файла.

```
void Manager::callManager()
{
	std::cout << "Choose what you want to do:\nopen, close, save, saveas, help,
exit.\n";
	char choice[20];
	std::cin >> choice;
	std::cin.clear();
	std::cin.ignore();
	if (strcmp(choice, "open") == 0)
	{
		open();
		workwithfile();
	}
	else if (strcmp(choice, "close") == 0)
	{
		close();
	}
	else if (strcmp(choice, "save") == 0)
	{
		save();
		callManager();
	}
	else if (strcmp(choice, "saveas") == 0)
	{
		saveas();
		callManager();
	}
	else if (strcmp(choice, "help") == 0)
	{
		help();
		callManager();
	}
	else if (strcmp(choice, "exit") == 0)
	{
		exitFunction();
	}
	else callManager();
}
```

При избор на *open* се отваря избраният файл за работа с билети, като се прави негово копие във временен текстов файл (*“temp.txt”*). Ако при отварянето на някой файл се появи грешка, програмата принтира съобщение и извиква функцията *callManager()*.Ако няма проблеми с файловете се извиква се функция *workwithfile()*, с чиято помощ се извършват операции върху данните, прочетени от файла. При потребителски вход *close* програмата затваря файла с данните, без да запазва създадените промени по време на процеса на работа. *Save* запазва направените промени в същия файл, от който са били прочетени данните, а *saveas* ги записва в файл, чието име трябва да се укаже от потребителя. Информацията се извежда с помощта на *help*, а *exit* излиза от програмата.

При влизане във функцията *workwithfile()*, потребителят трябва да въведе желаната функция за работа с данните. *Workwithfile()* работи на същия принцип като *callManager()* – очаква се вход от потребителя, който се конвертира в поток; програмата брои въведените думи и сравнява първата от тях с възможните операции. В края на функцията програмата пита потребителя дали иска да продължи да работи с данните от файла, ако не отново се вика *callManager()*. Приложен е част от кода, показващ главната функционалност на *workwithfile()*, като са пропуснати различните *if* проверки, които четат първата дума от потребителския вход.

```
void Manager::workwithfile()
{
	std::cout << "\nPlease, enter function with parameters:";
	char function[1000];
	std::cin.getline(function, 1000);
	std::stringstream ss(function);

	char word[MAX_NOTE_LENGTH];
	int wordCount = 0;
	while (ss >> word)
	{
		wordCount++;
	}
	if (wordCount == 0)
	{
		std::cerr << "Please write something!\n";
		workwithfile();
	}

	ss.clear();
	ss.seekg(0, std::ios::beg);
	char functionName[20];
	ss.getline(functionName, 20, DELIMITER);

	if (strcmp(functionName, "addevent") == 0)
	{
		if (wordCount != 4)
		{
			std::cout << "Wrong parameters!\n";
			workwithfile();
		}

		Date d;
		char eName[MAX_EVENT_NAME_LENGTH];
		char hall[MAX_HALL_NAME_LENGTH];

		ss >> d;
		ss.get();
		ss.getline(hall, MAX_EVENT_NAME_LENGTH, DELIMITER);
		ss.getline(eName, MAX_HALL_NAME_LENGTH);

		addevent(d,hall, eName);
		writeEvents();
		readEvents();
	}

...
```

```
else
	{
		std::cout << "Wrong input!\n\n";
	}

	std::cout << "\nDo you want to work with the same file more(y/n)\n";
	char answer;
	std::cin >> answer;
	std::cin.clear();
	std::cin.ignore();
	if (answer == 'y')
	{
		workwithfile();
	}
	else callManager();
}
```

Работи по следния начин: след прочитане на първата дума, а именно името на функцията, програмата отчита валидността на следващите подадени параметри, като се проверява дали е изпълнено условието за брой на думи (например на *addevent* трябва да се подадат още три думи освен името на функцията, за да има достатъчен брой данни, с които да се работи). Четат се параметрите от останалата част от потока, който вече е с една дума по-малко (първата, защото е взета за проверка), и се вика съответната операция, която се изисква.

При избор на *addevent* първо се чете датата, после два масива от букви – един за името на залата и друг за представлението. Самата функция *addevent(const Date&, const char*, const char*)* прави проверка за правилно изписана бъдеща дата, за съществуваща зала и дали тя е свободна на тази дата. Ако всички тези са изпълнени, програмата продължава напред и търси съответствие в контейнера от представления с името на подаденото. Ако бъде намерено, се извиква функцията *addShow(...)* от класа *Event*. В случай че не бъде намерено такова, се създава ново с представление на подадената дата в подадената зала.

freeseats(const Date&, const char name)* извежда справка за свободните места за представлението, чието име е подадено като параметър, на желаната дата. В случай че такова представление не съществува или датата не е валидна или минала, програмата изписва грешка. В кода са създадени два начина за принтиране на местата, като единият е поставен в коментар.

book(unsigned short row, unsigned short seat, const Date&, const char eName, const char* note)* проверява дали датата е вярно изписана и е бъдеща, ако е – програмата извиква функцията *getFreeSeatsManager()*, която проверява за съществуващо представление и свободните места за него на подадената дата. Ако датата не е валидна, желаното място е заето или такова шоу не съществува, програмата връща грешка. Функцията *book* продължава с цикъл за търсене на подаденото представление във вектора, член-данна на класът мениджър. При неговото намиране се създава билет с желаните параметри с функцията *setCodeviaParam(...)* от класа за билет и се извиква булевата такава *bookTicket(...)* от класа *Event*. При успешно/неуспешно запазване се извежда съобщение.

unbook(unsigned short, unsigned short, const Date&, const char)* отменя резервация на запазен билет. Проверява се дали датата е правилно написана, след което се търси дали името

на представлението съвпада с някое от съществуващите. Ако то бъде намерено се извиква *unbookTicket(…)* от *Event.*

buy(unsigned short row, unsigned short seat, const Date&, const char eName)* работи като *book(…)* с разликата, че билетът се запазва чрез функцията *bookTicket(…)* от класа *Event.* В този случай не се изисква бележка, подадена от потребителя.

bookings(const Date&, const char name)* и *bookings(const char*)* извеждат справка за закупените, но неплатени билети за въведеното представление на съответната дата. Втората функция служи в случай, че едно от двете е пропуснато. При нея се проверява първо дали въведеният параметър е име на представление. Ако бъде намерено се извежда информация за всички дати. В противен случай низът се превръща в поток и се пробва да се запише като дата. При успех се извежда информация за всички представления на дадената дата. Ако нито един от двата случая не се изпълни, се извежда съобщение за грешка.

check(const char code)* прави се проверка за валидност на билет с функцията *validCode(…),* дефинирана в *Ticket.cpp*. В случай че цялата нужна информация е въведена, се проверява дали представление с това име съществува и дали има шоу на конкретната дата във въведената зала. Ако всичко това е изпълнено, се проверява дали билетът е закупен, запазен или нито едно от двете и се принтира в зависимост от конкретната стройност.

report(const Date&, const Date&, const char hall = nullptr)* извежда информация за всички закупени билети в дадената зала за даден период от време, който интервал се намира с функцията *getDatesBetween(…)* от класа дата. Ако залата не въведена, се извежда информация за всички зали.

mostwatched() извежда статистика за най-гледаните представления, като ги принтира в намаляващ ред относно броят продадени билети за целия период от време.

Ще бъдат разгледани накратко споменатите функции от други класове.

bool Event::addShow(const Date&, const char)* проверява дали има шоу от това представление на тази дата. Ако има – не се случва нищо и функцията връща стойност *false.* В противен случай се добавя ново шоу в асоциативният контейнер *shows* на желаната дата и върнатата стойност е *true.*

За функциите *bool Event::bookTicket(Ticket&)* и *bool Event::buyTicket(Ticket&)* е създадена *private* булева такава, добавяща билет за съществуващо представление на тази дата. В класа за билети е предефиниран оператора за равенство и има възможност да се сравняват по код. Така може да направим бърза проверка дали билетът, който искаме да запазим/купим вече го има. В случай че това е така, функцията връща *false,* а иначе *true.* В зависимост от повиканата функция билетът се задава да е запазен или закупен. След това го добавяме в динамичния масив от билети чрез помощната.

Event::unbookTicket(Ticket&) проверява дали билетът, чиято резервация искаме да премахнем, съществува и е запазен. Ако тези условия не са изпълнени се връща съобщение за грешка. В противен случай мястото му в паметта се трие, намаля се *ticketsSize* от член-данните на класа и се принтира съобщение за успешно изпълнение на задачата.

Event::report(const Date&, const Date&, const char hall = nullptr)* правят се проверка за датите дали са правилно въведени и една след друга. След това се проверява за всяко шоу в този период колко билета са закупени. Ако залата бъде указана, то програмата проверява специфично за нея. В противен случай извежда информация за всички зали.

vector<pair<size_t, size_t> > Event::getFreeSeats(const Date&, map<size_t, size_t> places) функцията приема като аргументи дата и асоциативен контейнер, който представлява местата във всяка зала – ключ е редът, а стойността му е броят места на него. Създава се вектор от всички възможни двойки ред-място. Проверяваме всички билети, запазени/закупени за това представление, и при намиране на билет за подадената дата – премахваме реда и мястото му от вектора със свободни места. Връща се векторът с останалите свободни места.

Всеки клас съдържа в себе си конструктор и деструктор, а всички без мениджър класа съдържат и конструктор за копиране и присвояване. В класовете са дефинирани и селектори и мутатори. Деструкторите позволяват освобождаване на заетата памет по време на работенето с програмата. Създадени са глобални променливи, представители на типовете *char* и *int,* за по-лесно програмиране и четене на файла от други хора.

Тестовете, които трябва да се извършат, за да се провери коректността на програмата са различни. Един от тях е да се въведат по-малък на брой думи, отколкото функцията изисква и следователно липсват данни, с които да се работи. Валидността на датата също е от съществено значение за обработване на информацията. Въвеждане на данните в грешен ред. Примерите по-долу показват грешно въведени данни от потребителя.

$addevent 2020-02-30 hall name – грешно въведени данни (във февруари има 28/29 дена)

$addevent hall name (2020-02-02) –неправилна подредба или липса на информация

Датите се изписват в международния стандарт за обмен на дати и данни за време *ISO 8601.* Ако потребителят се опита да ги въведе в друг формат, програмата ще отчете датата за невалидна.

5. Заключение

При работа с програмата потребителят има възможност да изпълнява всички функционалности при въвеждане на валидни данни. Програмата може лесно да бъде модифицирана с цел скалируемост. Освен че може да работи с неограничен брой зали и представления, лесно могат да бъдат добавени данни за цена на билет. Това би дало множество възможности – справка кое представление колко печели; вариант за даване на отстъпка за учащи, деца, хора в неравностойно положение и други. Друг плюс би бил включен списък на артистите за всяко представление, което би помогнало на клиента да избере за коя дата иска да закупи билет.

За улеснение на потребителя е ключово и създаването на графичен потребителски интерфейс, в който командите не трябва да се пишат директно от потребителя, а ще са предоставени елементи за управление във вид на графични изображения.