1. Ministry of Science and Education of Russian Federation
2. Peter the Great St.Petersburg Polytechnic University
3. —
4. Institute of cubersecurity and information security

**LABORATORY WORK № 2**

1. **«UML Diagram»**
2. course «OOP»
3. Student
4. Gr. 4851003/80802 Bahir M. N.

<*signature*>

1. Instructor

Chernov A.Y

<*signature*>

1. Saint-Petersburg
2. 2020

**Table of contents**

[Tasks 3](#_Toc63421879)

[Theory and results 3](#_Toc63421880)

[Working process 3](#_Toc63421881)

[Conclusion 7](#_Toc63421882)

[APPENDIX 1 7](#_Toc63421883)

[APPENDIX 2 7](#_Toc63421884)

[APPENDIX 3 7](#_Toc63421885)

[APPENDIX 4 7](#_Toc63421886)

# **Tasks**

Examine theoretical material. According to the variant of the task, select entities related to the specified subject area. For each entity, fill the table. Build a UML class diagram. Self-Study the UML notation for the type of chart specified in the variant. Fill in the table that characterizes the aspect reflected in this diagram.

Develop a UML class diagram for an existing C project. Implement a program in C ++ that reflects the developed class diagram and preserves the functionality of the project under study in C.

# **Theory and results**

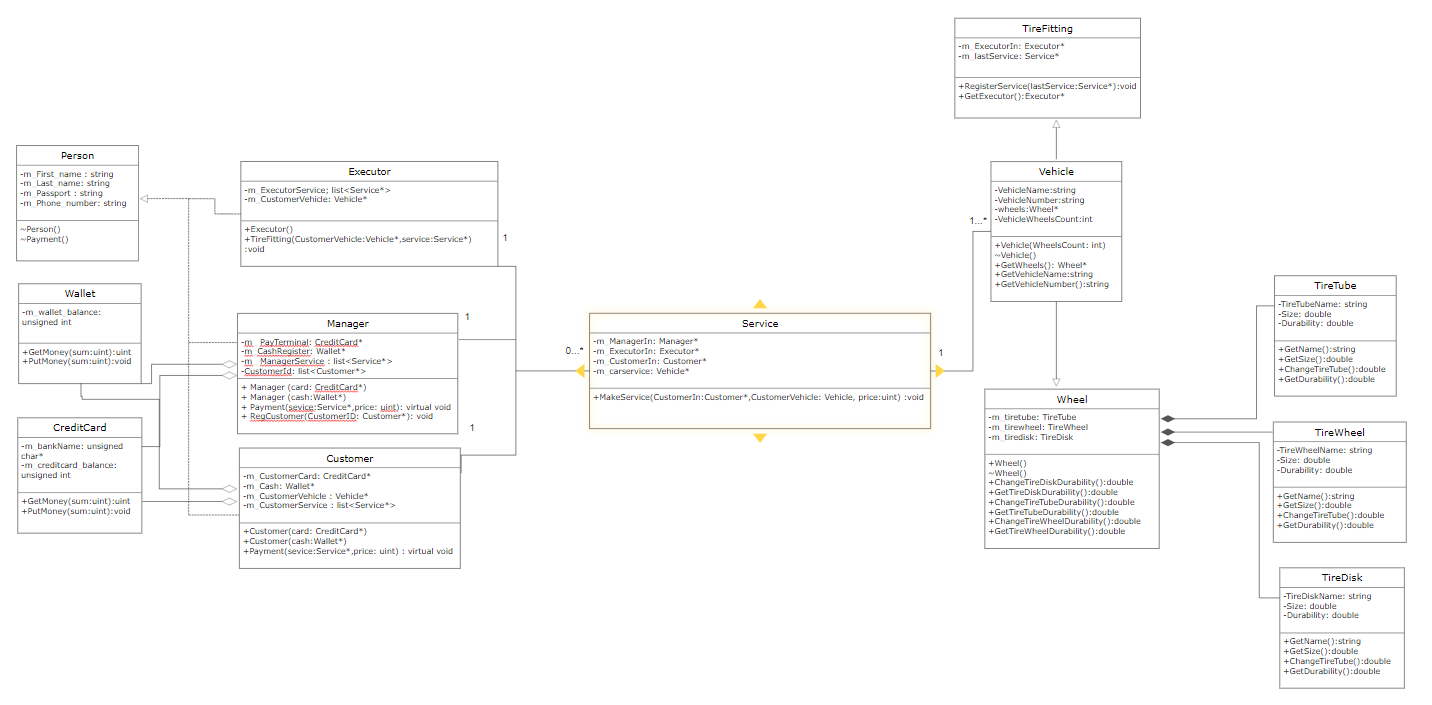
# **Working process**

Subject area - "Pit-stops" tire fitting.

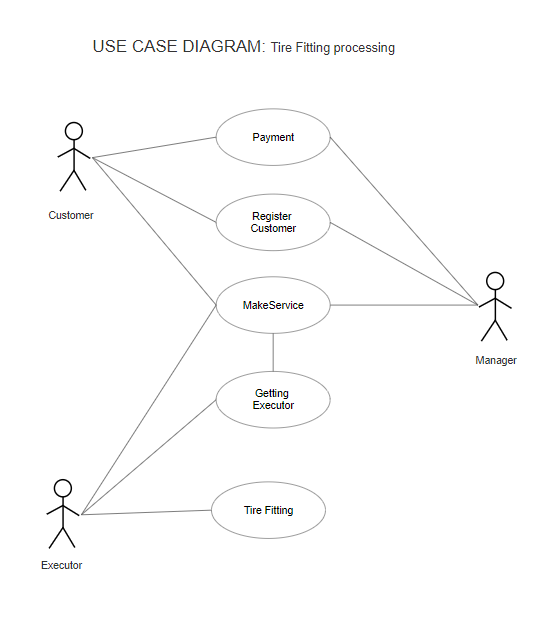
According to the variant of the task, I have identified 10 entities related to the specified subject area:

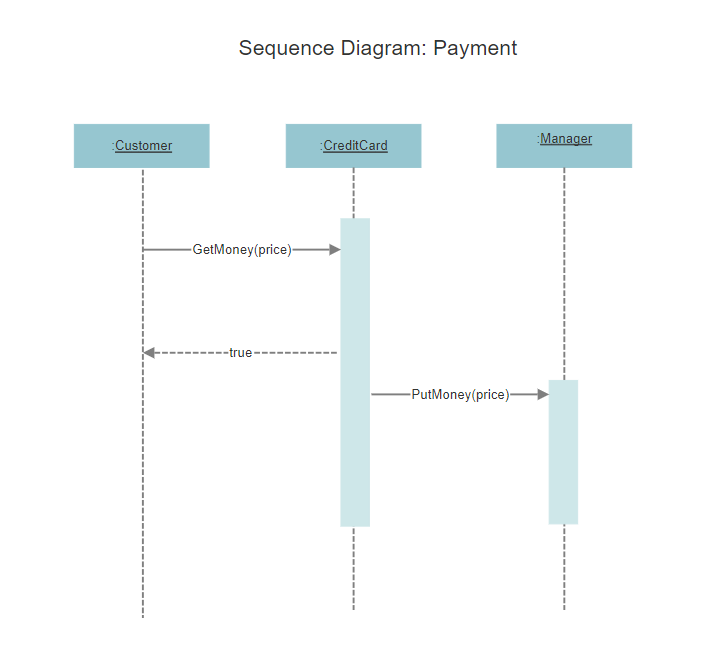
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Type of entity | Role | Attributes | Operations |
| Person | Interface | Represents the definition of a person | -m\_First\_name : string  -m\_Last\_name: string  -m\_Passport : string  -m\_Phone\_number: string | ~Person()  ~Payment() |
| Customer | Class | Pretends to be the one who came for a tire service | -m\_CustomerCard: CreditCard\*  -m\_Cash: Wallet\*  -m\_CustomerVehicle : Vehicle\*  -m\_CustomerService : list<Service\*> | +Customer(card: CreditCard\*)  +Customer(cash:Wallet\*)  +Payment(sevice:Service\*,price: uint) : virtual void |
| Manager | Class | The one who offers the service | -m\_ PayTerminal: CreditCard\*  -m\_CashRegister: Wallet\*  -m\_ ManagerService : list<Service\*>  -CustomerId: list<Customer\*> | + Manager (card: CreditCard\*)  + Manager (cash:Wallet\*)  +Payment(sevice:Service\*,price: uint): virtual void  + RegCustomer(CustomerID: Customer\*): void |
| Credit Card | Class | Means of payment | -m\_bankName: unsigned char\*  -m\_creditcard\_balance: unsigned int | +GetMoney(sum:uint):bool  +PutMoney(sum:uint):void |
| Wallet | Class | Means of payment | -m\_wallet\_balance: unsigned int | +GetMoney(sum:uint):bool  +PutMoney(sum:uint):void |
| Service | Class | Service delivery process | -m\_ManagerIn: Manager\*  -m\_ExecutorIn: Executor\*  -m\_CustomerIn: Customer\*  -m\_carservice; Vehicle\* | +MakeService(CustomerIn:Customer\*,CustomerVehicle: Vehicle, price:uint) :void |
| Executor | Class | The one who performs the tire service | -m\_ExecutorService; list<Service\*>  -m\_CustomerVehicle: Vehicle\* | +Executor()  +TireFitting(CustomerVehicle:Vehicle\*,service:Service\*) :void |
| Vehicle | Class | The car owned by the one who came for the service and what needs to be provided a tire service | -VehicleName:string  -VehicleNumber:string  -wheels:Wheel\*  -VehicleWheelsCount:int | +Vehicle(WheelsCount: int)  ~Vehicle()  +GetWheels(): Wheel\*  +GetVehicleName:string  +GetVehicleNumber():String |
| Wheel | Class | The main parts that need to be served, what the vehicle consists of | -m\_tiretube: TireTube  -m\_tirewheel: TireWheel  -m\_tiredisk: TireDisk | +Wheel()  ~Wheel()  +ChangeTireDiskDurability():double  +GetTireDiskDurability():double  +ChangeTireTubeDurability():double  +GetTireTubeDurability():double  +ChangeTireWheelDurability():double  +GetTireWheelDurability():double |
| TireFitting | Class | The service itself to be provided | -m\_ExecutorIn: Executor\*  -m\_lastService: Service\* | +RegisterService(lastService:Service\*):void  +GetExecutor():Executor\* |
| TireTube | Class | The main components of the wheel to be rendered service | -TireTubeName: string  -Size: double  -Durability: double | +GetName():string  +GetSize():double  +ChangeTireTube():double  +GetDurability():double |
| TireDisk | Class | The main components of the wheel to be rendered service | -TireDiskName: string  -Size: double  -Durability: double | +GetName():string  +GetSize():double  +ChangeTireTube():double  +GetDurability():double |
| TireWheel | Class | The main components of the wheel to be rendered service | -TireWheelName: string  -Size: double  -Durability: double | +GetName():string  +GetSize():double  +ChangeTireTube():double  +GetDurability():double |

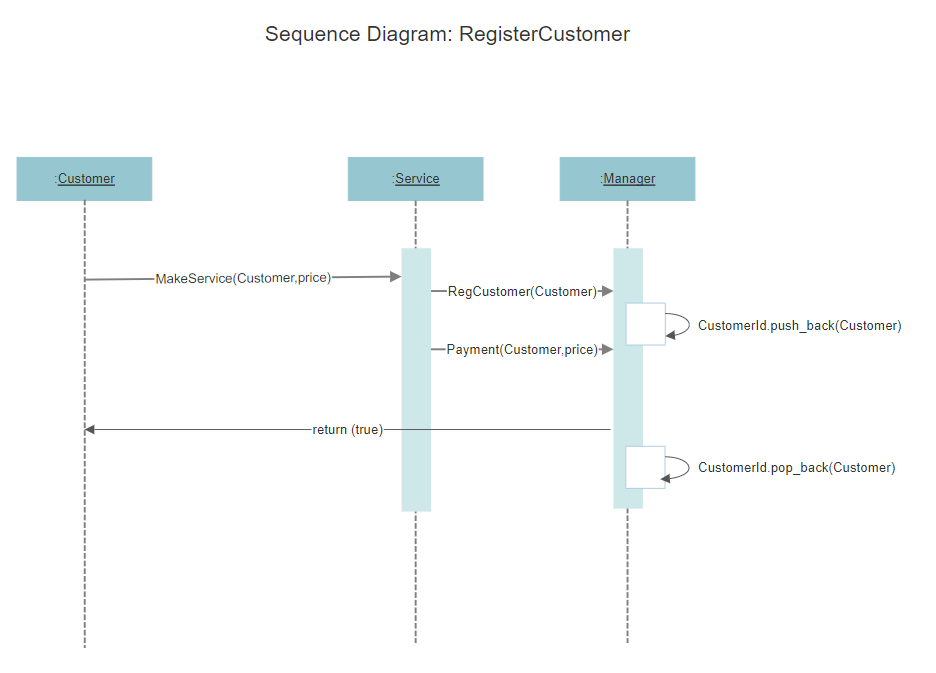
Built a UML class diagram using 4 or more kinds of relationships. Specified the access modifiers as well as the cardinality of the class relationships.

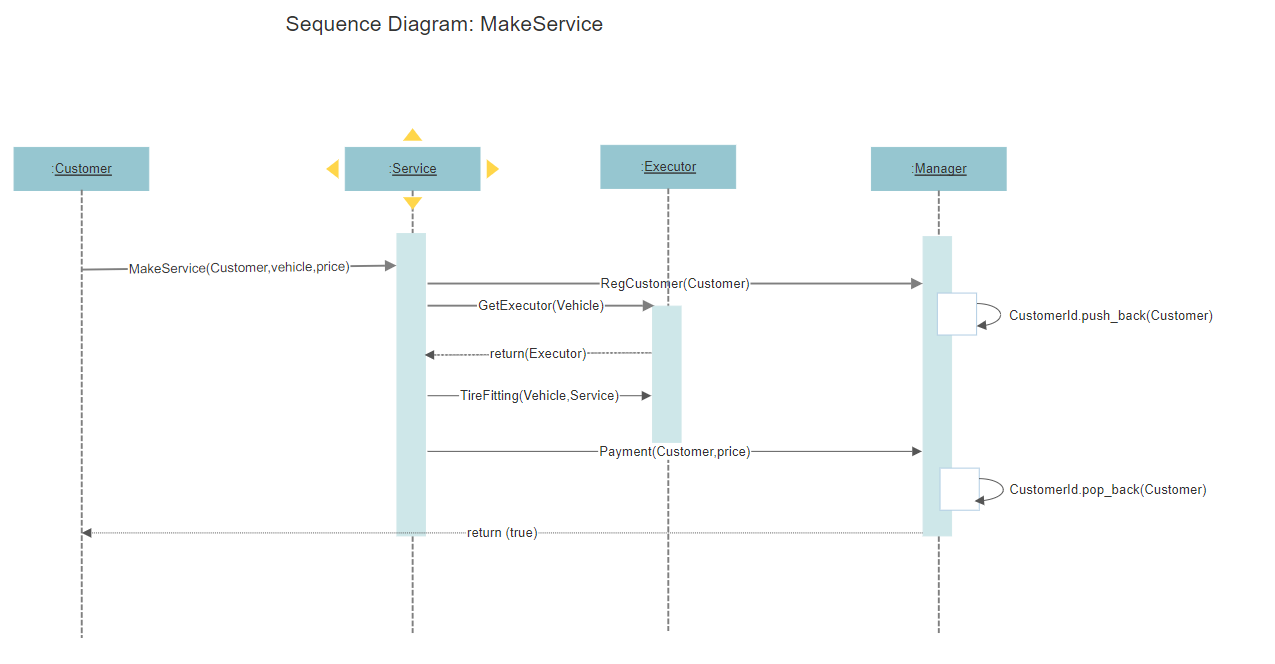


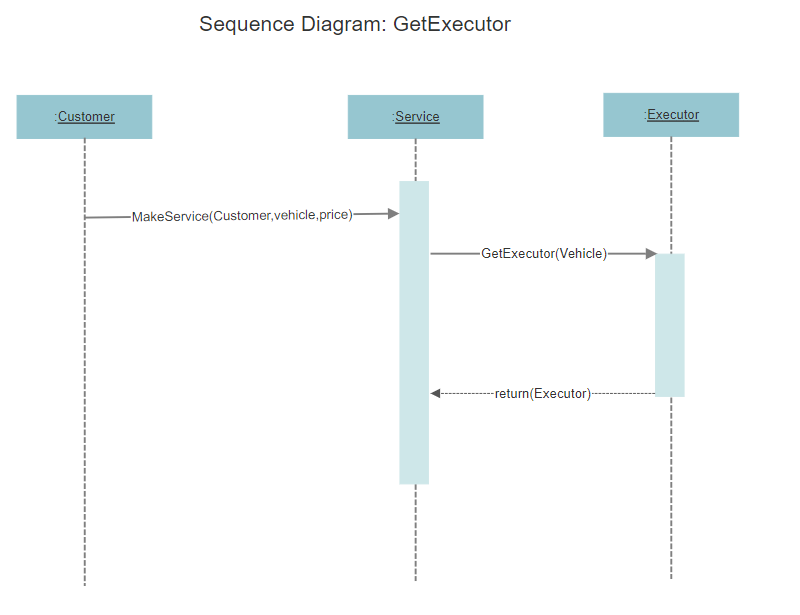
Pit-Stops Tire Mounting Logic: We have a Client (Customer) who wants to receive a tire fitting service, for which he pays money either from a wallet or by credit card. There is a Manager who registers the Client and carries out the function of payment for the service. There is an Executor, the same worker who performs the tire fitting service. There is a client's car (Vehicle), which is used for tire fitting. The machine consists of wheels, and inherits all the methods of Wheels. Wheels, in turn, consist of TireDisk, TireTube, TireWheel, which in turn give strength to each, the same name or brand of the manufacturer. There is a Tire Service class, this is our service that requires a Contractor and returns a pointer to it. There is a MakeService Class in which everything happens. First, the manager registers the Client (Customer), after which his Vehicle is registered for tire fitting. Next, an Executor is needed, for this service for this Vehicle, for which the method is called to get the Executor. The executor carries out tire fitting, restoring wheel parts. After that, the Manager makes the payment and puts the money either in the cashier or on a bank card. The code of the developed program is given in Appendix 1.

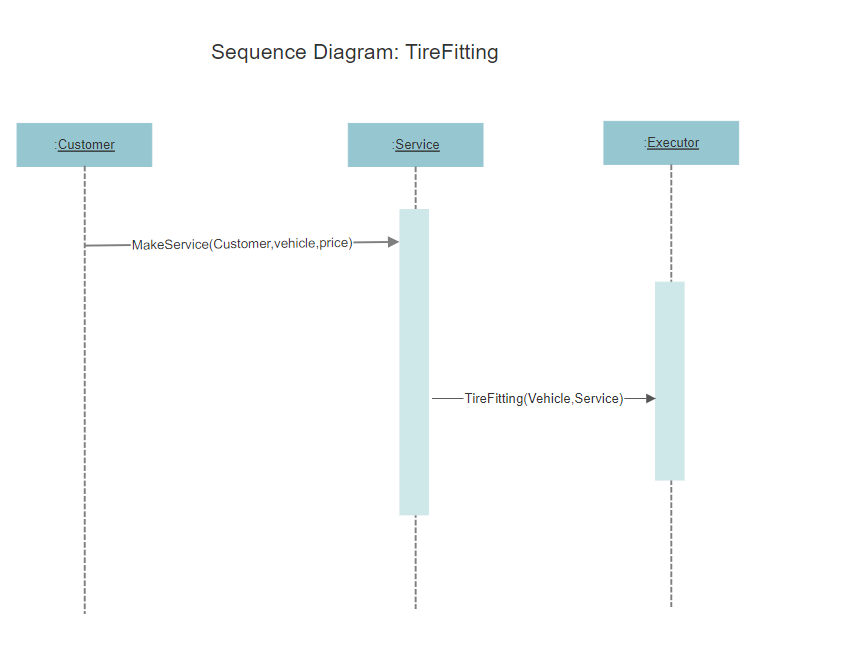




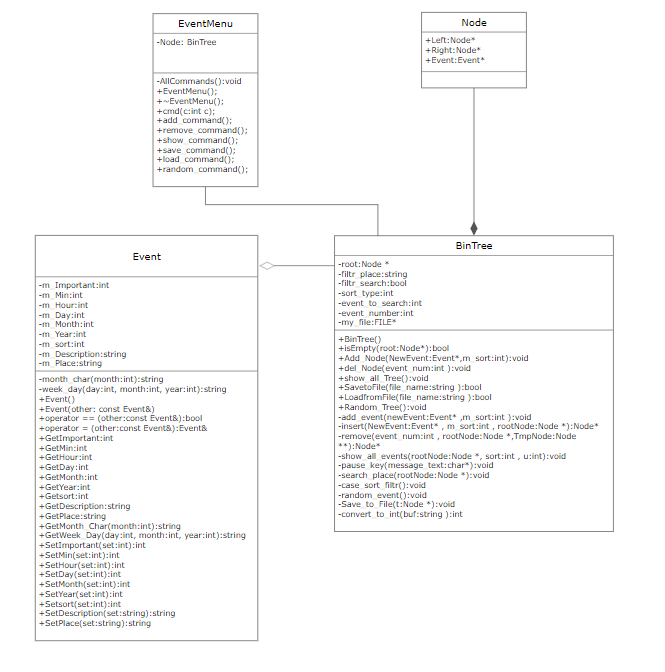








As a selected project in C, the program developed in the previous courses "Events" or the Diary. This is a binary tree program. Has the functions of adding an event, deleting an event, sorting by importance, by date, by location. Also filling the diary with random events. Also saving and loading from file.



The code of the developed program is given in Appendix 2

Screenshots, that program is works.



Figure 1. Main menu

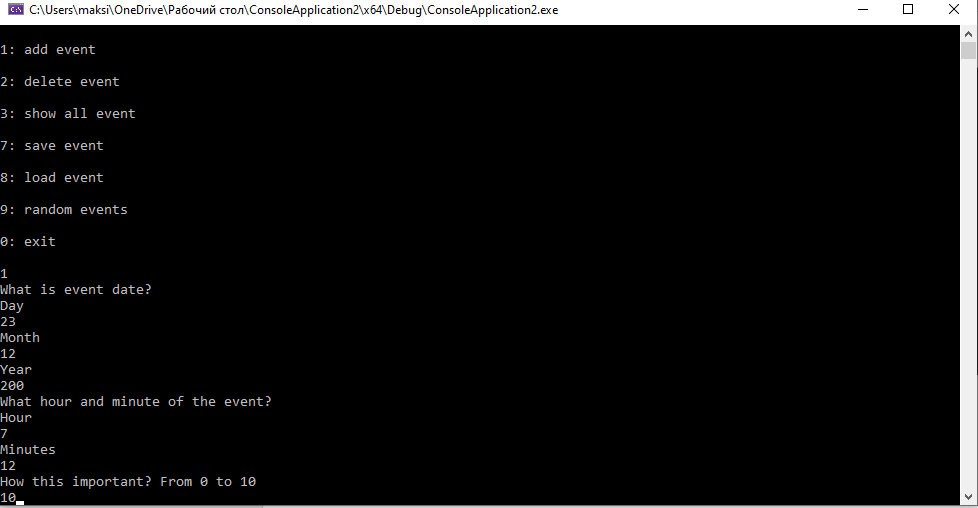


Figure Add Event



Figure 3 Menu "Show all events" (sort)

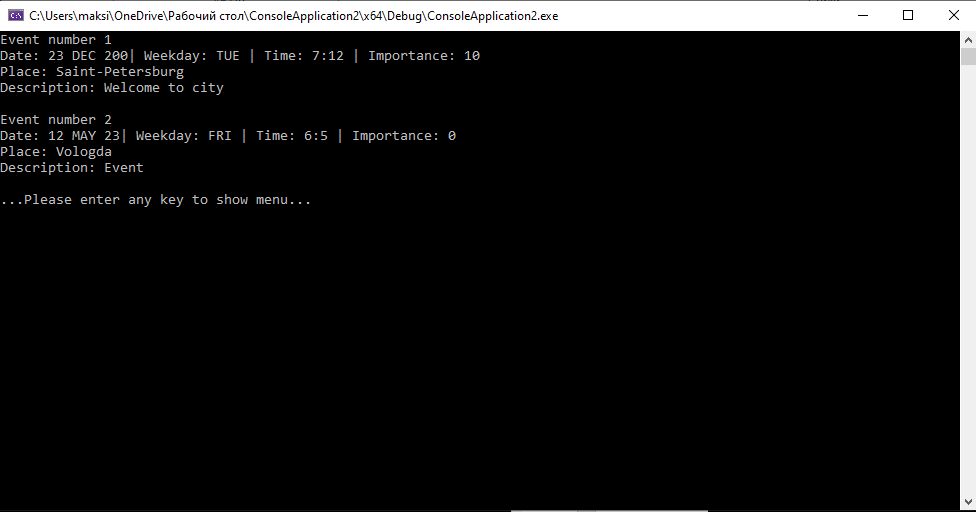


Figure 4 Function Show all Events(increase date sort)



Figure 5 Function Delete Event (part 1)

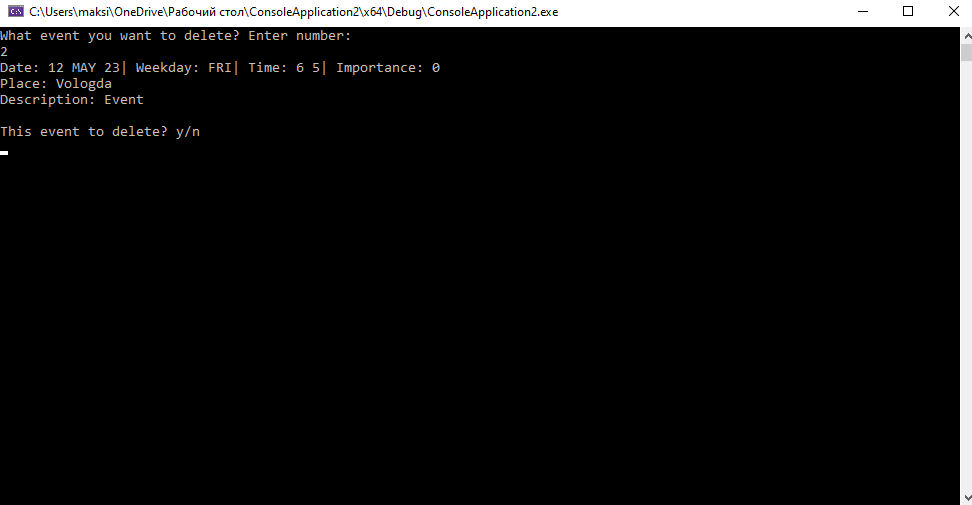


Figure 6 Function Delete Event (part 2)



Figure Function Random Events

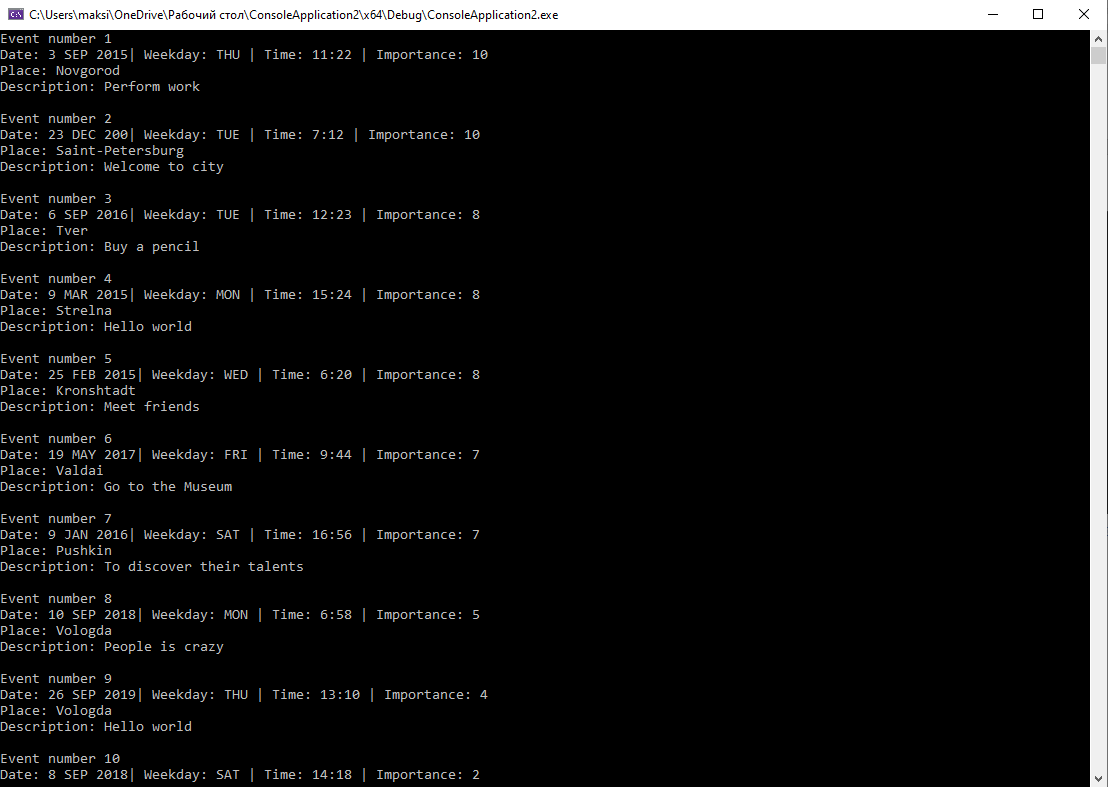


Figure 8 Show all Events after Random (increase important)

# **Conclusion**

I studied the types of relationships between objects and classes, got acquainted with the basic elements of defining, representing, designing and modeling software systems using the UML language, and gained the skill of developing UML diagrams for applied applications. A demo in C ++ was also developed for each type of relationship used, taking into account the subject area. And the existing project implemented in the C language was redesigned.

APPENDIX 1

CreditCard.h:

#pragma once

#include <iostream>

class CreditCard {

public:

bool GetMoney(unsigned int sum) {

if (m\_creditcard\_balance >= sum) {

m\_creditcard\_balance -= sum;

return true;

}

else return false;

}

void PutMoney(unsigned int sum) {

m\_creditcard\_balance += sum;

}

private:

unsigned char\* m\_bankName;

unsigned int m\_creditcard\_balance;

};

Customer.h:

#pragma once

#include <iostream>

#include "Person.h"

#include "CreditCard.h"

#include "Wallet.h"

#include "Vehicle.h"

#include "Service.h"

class Customer : Person {

public:

Customer(CreditCard\* card) {

m\_CustomerCard = card;

}

Customer(Wallet\* cash) {

m\_Cash = cash;

}

virtual void Payment(Service\* service, unsigned int price)

{

Vehicle\* m\_CustomerVehicle;

if (m\_CreditCardOrWalletPay) {

service->MakeService(this, m\_CustomerVehicle, m\_CustomerCard->GetMoney(price));

}

else {

service->MakeService(this, m\_CustomerVehicle, m\_Cash->GetMoney(price));

}

m\_CustomerService.push\_back(service);

}

private:

CreditCard\* m\_CustomerCard;

Wallet\* m\_Cash;

Vehicle\* m\_CustomerVehicle;

std::list<Service\*> m\_CustomerService;

};

Executor.h:

#pragma once

#include <iostream>

#include "Person.h"

#include "CreditCard.h"

#include "Wallet.h"

#include "Vehicle.h"

#include "Service.h"

class Executor : Person {

public:

Executor() {

}

void TireFitting(Vehicle\* CustomerVehicle, Service\* service) {

m\_CustomerVehicle = CustomerVehicle;

Wheel\* wheels = CustomerVehicle->GetWheels();

int VehicleWheelsCount = CustomerVehicle->GetWheelsCount();

for (int i = 0; i < VehicleWheelsCount; i++) {

if (wheels[i].GetTireDiskDurability() < 100.0) {

wheels[i].ChangeTireDiskDurability();

}

if (wheels[i].GetTireTubeDurability() < 100.0) {

wheels[i].ChangeTireDiskDurability();

}

if (wheels[i].GetTireWheelDurability() < 100.0) {

wheels[i].ChangeTireDiskDurability();

}

m\_ExecutorService.push\_back(service);

}

}

private:

std::list<Service\*> m\_ExecutorService;

Vehicle\* m\_CustomerVehicle;

};

Manager.h:

#pragma once

#include "Person.h"

#include "Wallet.h"

#include "CreditCard.h"

#include "Service.h"

class Manager : Person {

public:

Manager(CreditCard\* card) {

m\_PayTerminal = card;

}

Manager(Wallet\* cash) {

m\_CashRegister = cash;

}

void RegCustomer(Customer\* CustomerID) {

CustomerId.push\_back(CustomerID);

}

virtual void Payment(Service\* service, unsigned int price)

{

if (m\_CreditCardOrWalletPay) {

m\_PayTerminal->PutMoney(price);

}

else {

m\_CashRegister->PutMoney(price);

}

m\_ManagerService.push\_back(service);

CustomerId.pop\_back();

}

private:

CreditCard\* m\_PayTerminal;

Wallet\* m\_CashRegister;

std::list<Service\*> m\_ManagerService;

std::list<Customer\*> CustomerId;

};

Person.h:

#pragma once

#include <iostream>

#include <list>

#include <string>

class Person {

public:

bool m\_CreditCardOrWalletPay;

virtual ~Person() {}

virtual void Payment()=0;

private:

std::string m\_First\_name;

std::string m\_Last\_name;

std::string m\_Passport;

std::string m\_Phone\_number;

};

Service.h:

#pragma once

#include <iostream>

#include "Vehicle.h"

#include "Customer.h"

#include "Executor.h"

#include "Manager.h"

class Service {

public:

void MakeService(Customer\* CustomerIn, Vehicle\* CustomerVehicle, unsigned int price){

m\_CustomerIn = CustomerIn;

m\_carservice = CustomerVehicle;

m\_ManagerIn->RegCustomer(m\_CustomerIn);

CustomerVehicle->RegisterService(this);

m\_ExecutorIn = CustomerVehicle->GetExecutor();

m\_ExecutorIn->TireFitting(CustomerVehicle,this);

m\_ManagerIn->Payment(this, price);

}

private:

Manager\* m\_ManagerIn;

Executor\* m\_ExecutorIn;

Customer\* m\_CustomerIn;

Vehicle\* m\_carservice;

};

TireFitting.h:

#pragma once

#include <iostream>

#include "Vehicle.h"

#include "Executor.h"

#include "Service.h"

class TireFitting {

public:

inline void RegisterService(Service\* lastService) {

m\_lastService = lastService;

}

inline Executor\* GetExecutor() {

return m\_ExecutorIn;

}

protected:

Executor\* m\_ExecutorIn;

Service\* m\_lastService;

};

Vehicle.h:

#pragma once

#include <iostream>

#include "Wheel.h"

#include "TireFitting.h"

class Vehicle : public TireFitting, public Wheel{

public:

Vehicle(int WheelsCount) {

VehicleWheelsCount = WheelsCount;

wheels = new Wheel[VehicleWheelsCount];

std::cout << "All vehicle wheels Created\n";

}

~Vehicle() {

delete[] wheels;

}

Wheel\* GetWheels() {

return wheels;

}

std::string GetVehicleName(){

return VehicleName;

}

std::string GetVehicleNumber() {

return VehicleNumber;

}

int GetWheelsCount() {

return VehicleWheelsCount;

}

private:

std::string VehicleName;

std::string VehicleNumber;

Wheel\* wheels;

int VehicleWheelsCount;

};

Wallet.h:

#pragma once

#include <iostream>

class Wallet {

public:

bool GetMoney(unsigned int sum) {

if (m\_wallet\_balance >= sum) {

m\_wallet\_balance -= sum;

return true;

}

else return false;

}

void PutMoney(unsigned int sum) {

m\_wallet\_balance += sum;

}

private:

unsigned int m\_wallet\_balance;

};

Wheel.h:

#pragma once

#include <iostream>

class Wheel {

public:

class TireTube;

class TireWheel;

class TireDisk;

Wheel() : m\_tiretube(), m\_tirewheel(),m\_tiredisk() {

std::cout << "Wheel Created\n";

};

class TireTube {

std::string TireTubeName;

double Size;

double Durability;

public:

TireTube() { std::cout << "TireTube Created\n"; }

std::string GetName() { return TireTubeName; }

double GetSize() { return Size; }

double ChangeTireTube() { Durability = 100.0; }

double GetDurability() { return Durability; }

};

class TireWheel {

std::string TireWheelName;

double Size;

double Durability;

public:

TireWheel() { std::cout << "TireWheel Created\n"; }

std::string GetName() { return TireWheelName; }

double GetDurability() { return Durability; }

double ChangeTireWheel() { Durability = 100.0; }

double GetSize() { return Size; }

};

class TireDisk {

std::string DiskName;

double Size;

double Durability;

public:

TireDisk() { std::cout << "TireDisk Created\n"; }

std::string GetName() { return DiskName; }

double ChangeTireDisk() { Durability = 100.0; }

double GetSize() { return Size; }

double GetDurability() { return Durability; }

};

double ChangeTireDiskDurability() { m\_tiredisk.ChangeTireDisk(); }

double GetTireDiskDurability() { return m\_tiredisk.GetDurability(); }

double ChangeTireTubeDurability() { m\_tiretube.ChangeTireTube(); }

double GetTireTubeDurability() { return m\_tiretube.GetDurability(); }

double ChangeTireWheelDurability() { m\_tirewheel.ChangeTireWheel(); }

double GetTireWheelDurability() { return m\_tirewheel.GetDurability(); }

private:

TireTube m\_tiretube;

TireWheel m\_tirewheel;

TireDisk m\_tiredisk;

double m\_Pressure;

};

APPENDIX 2

EventMenu.h:

#pragma once

#include <iostream>

#include "BinTree.h"

class EventMenu {

public:

EventMenu();

~EventMenu();

void cmd(int c);

void add\_command();

void remove\_command();

void show\_command();

void save\_command();

void load\_command();

void random\_command();

private:

void AllCommands();

BinTree Node;

};

Event.h:

#pragma once

#include <iostream>

#include <string>

class Event {

public:

Event() = default;

Event(const Event& other) {

\*this = other;

}

bool operator== (const Event& other) const {

return (m\_Description == other.m\_Description)

&& (m\_Place == other.m\_Place) && (m\_Important == other.m\_Important);

}

Event& operator= (const Event& other) {

m\_Place = other.m\_Place;

m\_Description = other.m\_Description;

m\_Day = other.m\_Day;

m\_Hour = other.m\_Hour;

m\_Min = other.m\_Min;

m\_Month = other.m\_Month;

m\_Year = other.m\_Year;

m\_Important = other.m\_Important;

return \*this;

}

int GetImportant() {

return m\_Important;

}

int GetMin() {

return m\_Min;

}

int GetHour (){

return m\_Hour;

}

int GetDay (){

return m\_Day;

}

int GetMonth (){

return m\_Month;

}

int GetYear (){

return m\_Year;

}

int GetSort (){

return m\_sort;

}

std::string GetDescription() {

return m\_Description;

}

std::string GetPlace() {

return m\_Place;

}

std::string GetMonth\_Char(int month) {

return month\_char(month);

}

std::string GetWeek\_Day(int day, int month, int year) {

return week\_day(day, month, year);

}

void SetImportant(int set) {

m\_Important = set;

}

void SetMin(int set) {

m\_Min = set;

}

void SetHour(int set) {

m\_Hour = set;

}

void SetDay(int set) {

m\_Day = set;

}

void SetMonth(int set) {

m\_Month = set;

}

void SetYear(int set) {

m\_Year = set;

}

void SetSort(int set) {

m\_sort = set;

}

void SetDescription(std::string set) {

m\_Description=set;

}

void SetPlace(std::string set) {

m\_Place = set;

}

private:

int m\_Important=0;

int m\_Min = 0;

int m\_Hour = 0;

int m\_Day = 0;

int m\_Month = 0;

int m\_Year = 0;

int m\_sort = 1;

std::string m\_Description;

std::string m\_Place;

std::string month\_char(int month)

{

std::string mc;

switch (month) {

case 1: mc = "JAN"; break;

case 2: mc = "FEB"; break;

case 3: mc = "MAR"; break;

case 4: mc = "APR"; break;

case 5: mc = "MAY"; break;

case 6: mc = "JUN"; break;

case 7: mc = "JUL"; break;

case 8: mc = "AUG"; break;

case 9: mc = "SEP"; break;

case 10: mc = "OKT"; break;

case 11: mc = "NOV"; break;

case 12: mc = "DEC"; break;

}

return mc;

}

std::string week\_day(int day, int month, int year)

{

std::string dnc;

int a = (14 - month) / 12;

int y = year - a;

int m = month + 12 \* a - 2;

int dn = (7000 + (day + y + y / 4 - y / 100 + y / 400 + (31 \* m) / 12)) % 7;

switch (dn) {

case 0: dnc = "SUN"; break;

case 1: dnc = "MON"; break;

case 2: dnc = "TUE"; break;

case 3: dnc = "WED"; break;

case 4: dnc = "THU"; break;

case 5: dnc = "FRI"; break;

case 6: dnc = "SAT"; break;

}

return dnc;

}

};

BinTree.h:

#pragma once

#include <iostream>

#include <string>

#include <conio.h>

#include "Event.h"

class BinTree : Event {

class Node;

public:

BinTree(){

root = NULL;

}

bool isEmpty(Node\* root) {

return(root == NULL);

}

void add\_Node(Event\* NewEvent, int m\_sort) {

add\_event(NewEvent, m\_sort);

SavetoFile("my\_notebook.tmp");

}

void del\_Node(int event\_num)

{

remove(event\_num, root,&root);

SavetoFile("my\_notebook.tmp");

}

void show\_all\_Tree() {

case\_sort\_filtr();

}

bool SavetoFile(std::string file\_name) {

my\_file = fopen(file\_name.c\_str(), "w");

if (my\_file == NULL) {

return false;

}

Save\_to\_File(root);

fclose(my\_file);

return true;

}

bool LoadfromFile(std::string file\_name) {

my\_file = fopen(file\_name.c\_str(), "r");

if (my\_file == NULL) {

return false;

}

Load\_from\_File(root);

SavetoFile("my\_notebook.tmp");

fclose(my\_file);

return true;

}

void Random\_Tree() {

random\_event();

SavetoFile("my\_notebook.tmp");

}

private:

void add\_event(Event\* newEvent,int m\_sort) {

int k = 1;

std::string Description;

std::string Place;

int day, month, year,imp,hour,min;

std::cout<<"What is event date?\n";

while (k != 0) {

std::cout << "Day\n";

std::cin >> day;

std::cout << "Month\n";

std::cin >> month;

std::cout << "Year\n";

std::cin >> year;

if ((day < 1 || day > 31) || (month < 1 || month > 12) || (year < 0 || year > 3000))

std::cout<<"Wrong input. Enter the date correctly\n";

else {

k = 0;

newEvent->SetDay(day);

newEvent->SetMonth(month);

newEvent->SetYear(year);

}

//printf("%d:%d:%d\n", newevent->day, newevent->month, newevent->year);

}

std::cout << "What hour and minute of the event?\n";

k = 1;

while (k != 0) {

std::cout << "Hour\n";

std::cin >> hour;

std::cout << "Minutes\n";

std::cin >> min;

if ((hour < 0 || hour > 23) || (min < 0 || min > 59))

std::cout << "Wrong time. Enter time correctly\n";

else {

k = 0;

newEvent->SetHour(hour);

newEvent->SetMin(min);

}

}

std::cout << "How this important? From 0 to 10\n";

k = 1;

while (k != 0) {

std::cin>>imp;

if (imp < 0 || imp > 10)

std::cout << "Wrong importance. Enter importance correctly\n";

else {

k = 0;

newEvent->SetImportant(imp);

}

}

std::cout << "What is event place?\n";

std::cin >> Place;

newEvent->SetPlace(Place);

std::cout << "What is event?\n";

std::cin.ignore();

getline(std::cin,Description);

newEvent->SetDescription(Description);

// save to tree

root = insert(newEvent, sort\_type, this->root);

}

Node\* insert(Event\* NewEvent, int m\_sort, Node \*rootNode) {

if (isEmpty(rootNode)) {

rootNode = new Node;

rootNode->Event = NewEvent;

rootNode->Left = NULL;

rootNode->Right = NULL;

//\*this = rootNode;

}

else {

if (m\_sort == 1) {//sort to date

if (rootNode->Event->GetYear() > NewEvent->GetYear())

rootNode->Right=insert(NewEvent, m\_sort, rootNode->Right);

else if (rootNode->Event->GetYear() < NewEvent->GetYear())

rootNode->Left = insert(NewEvent, m\_sort, rootNode->Left);

else if (rootNode->Event->GetMonth() > NewEvent->GetMonth())

rootNode->Right = insert(NewEvent, m\_sort, rootNode->Right);

else if (rootNode->Event->GetMonth() < NewEvent->GetMonth())

rootNode->Left = insert(NewEvent, m\_sort, rootNode->Left);

else if (rootNode->Event->GetDay() > NewEvent->GetDay())

rootNode->Right = insert(NewEvent, m\_sort, rootNode->Right);

else if (rootNode->Event->GetDay() < NewEvent->GetDay())

rootNode->Left = insert(NewEvent, m\_sort, rootNode->Left);

else if (rootNode->Event->GetHour() > NewEvent->GetHour())

rootNode->Right = insert(NewEvent, m\_sort, rootNode->Right);

else if (rootNode->Event->GetHour() < NewEvent->GetHour())

rootNode->Left = insert(NewEvent, m\_sort, rootNode->Left);

else if (rootNode->Event->GetMin() > NewEvent->GetMin())

rootNode->Right = insert(NewEvent, m\_sort, rootNode->Right);

else if (rootNode->Event->GetMin() < NewEvent->GetMin())

rootNode->Left = insert(NewEvent, m\_sort, rootNode->Left);

else if (rootNode->Event->GetImportant() > NewEvent->GetImportant())

rootNode->Right = insert(NewEvent, m\_sort, rootNode->Right);

else if (rootNode->Event->GetImportant() < NewEvent->GetImportant())

rootNode->Left = insert(NewEvent, m\_sort, rootNode->Left);

else std::cout << "Event exists. Please other parameters." << std::endl;

}

if (m\_sort == 2) {//sort to date

if (rootNode->Event->GetImportant() > NewEvent->GetImportant())

rootNode->Right = insert(NewEvent, m\_sort, rootNode->Right);

else if (rootNode->Event->GetImportant() < NewEvent->GetImportant())

rootNode->Left = insert(NewEvent, m\_sort, rootNode->Left);

else if (rootNode->Event->GetYear() > NewEvent->GetYear())

rootNode->Right = insert(NewEvent, m\_sort, rootNode->Right);

else if (rootNode->Event->GetYear() < NewEvent->GetYear())

rootNode->Left = insert(NewEvent, m\_sort, rootNode->Left);

else if (rootNode->Event->GetMonth() > NewEvent->GetMonth())

rootNode->Right = insert(NewEvent, m\_sort, rootNode->Right);

else if (rootNode->Event->GetMonth() < NewEvent->GetMonth())

rootNode->Left = insert(NewEvent, m\_sort, rootNode->Left);

else if (rootNode->Event->GetDay() > NewEvent->GetDay())

rootNode->Right = insert(NewEvent, m\_sort, rootNode->Right);

else if (rootNode->Event->GetDay() < NewEvent->GetDay())

rootNode->Left = insert(NewEvent, m\_sort, rootNode->Left);

else if (rootNode->Event->GetHour() > NewEvent->GetHour())

rootNode->Right = insert(NewEvent, m\_sort, rootNode->Right);

else if (rootNode->Event->GetHour() < NewEvent->GetHour())

rootNode->Left = insert(NewEvent, m\_sort, rootNode->Left);

else if (rootNode->Event->GetMin() > NewEvent->GetMin())

rootNode->Right = insert(NewEvent, m\_sort, rootNode->Right);

else if (rootNode->Event->GetMin() < NewEvent->GetMin())

rootNode->Left = insert(NewEvent, m\_sort, rootNode->Left);

else std::cout << "Event exists. Please other parameters." << std::endl;

}

}

return rootNode;

}

Node\* remove(int event\_num, Node\* rootNode, Node\*\* TmpNode) {

char ev;

if (event\_to\_search == 0 && isEmpty(rootNode))

return false;

if (!isEmpty(rootNode->Left)) remove(event\_num, rootNode->Left,&(\*TmpNode)->Left);

event\_to\_search++;

if (event\_to\_search == event\_num) {

std::cout << "Date: " << rootNode->Event->GetDay() << " " << rootNode->Event->GetMonth\_Char(rootNode->Event->GetMonth()) << " " << rootNode->Event->GetYear() << "| Weekday: " << rootNode->Event->GetWeek\_Day(rootNode->Event->GetDay(), rootNode->Event->GetMonth(), rootNode->Event->GetYear());

std::cout << "| Time: " << rootNode->Event->GetHour() << " " << rootNode->Event->GetMin() << "| Importance: " << rootNode->Event->GetImportant() << std::endl << "Place: " << rootNode->Event->GetPlace() << std::endl << "Description: " << rootNode->Event->GetDescription() << std::endl << std::endl; std::cout << "This event to delete? y/n"<<std::endl;

int num = 1;

while (num != 0) {

std::cin >> ev;

if (ev == 'y' || ev == 'Y') {

num = 0;

if (isEmpty(rootNode->Right)) {

\*TmpNode = rootNode->Left;

return rootNode;

}

else {

Node\* tempNode = rootNode->Right;

if (isEmpty(tempNode->Left)) {

tempNode->Left = rootNode->Left;

\*TmpNode = tempNode;

return rootNode;

}

else {

Node\* tempNode2 = tempNode->Left;

while (!(isEmpty(tempNode2->Left)))

{

tempNode = tempNode2;

tempNode2 = tempNode->Left;

}

tempNode->Left = tempNode2->Right;

tempNode2->Left = rootNode->Left;

tempNode2->Right = rootNode->Right;

\*TmpNode = tempNode2;

return rootNode;

}

}

return rootNode;

}

else if (ev == 'n' || ev == 'N') {

num = 0;

return false;

}

else std::cout << "Wrong input. Enter y or n" << std::endl;

}

}

if (!isEmpty(rootNode->Right)) remove(event\_num, rootNode->Right, & (\*TmpNode)->Right);

}

void show\_all\_events(Node\* rootNode, int sort, int u) {

if (isEmpty(rootNode)) { //if tree null to exit

//printf("Events not exist \n");

return;

}

if (sort == 1 || sort == 9 || sort == 3) { // 1, 3 - sort increase; 9 - firter output

if (!isEmpty(rootNode->Left)) show\_all\_events(rootNode->Left, sort, ++u);//Left

//print event to console

if (sort == 1 || sort == 3) {

event\_number++;

//printf("Event number %d | node numder %d\n", event\_number, u);

if (filtr\_place == rootNode->Event->GetPlace() || filtr\_place == "") {

std::cout<<"Event number "<< event\_number<<std::endl;

std::cout << "Date: " << rootNode->Event->GetDay() << " " << rootNode->Event->GetMonth\_Char(rootNode->Event->GetMonth()) << " " << rootNode->Event->GetYear() << "| Weekday: " << rootNode->Event->GetWeek\_Day(rootNode->Event->GetDay(), rootNode->Event->GetMonth(), rootNode->Event->GetYear());

std::cout << " | Time: " << rootNode->Event->GetHour() << ":" << rootNode->Event->GetMin() << " | Importance: " << rootNode->Event->GetImportant() << std::endl << "Place: " << rootNode->Event->GetPlace() << std::endl << "Description: " << rootNode->Event->GetDescription() << std::endl << std::endl;

}

}

else

if (sort == 9) {

event\_number++;

std::cout<< rootNode->Event->GetPlace()<<std::endl;

}

u--;

if (!isEmpty(rootNode->Right)) show\_all\_events(rootNode->Right, sort, ++u); //Right

} //end sort increase

else

if (sort == 2 || sort == 4) { // 2, 4 - sort decrease

if (!isEmpty(rootNode->Right)) show\_all\_events(rootNode->Right, sort, ++u);//Left

//print event to console

event\_number++;

if (filtr\_place == rootNode->Event->GetPlace() || filtr\_place == "") {

std::cout << "Event number " << event\_number << std::endl;

std::cout << "Date: " << rootNode->Event->GetDay() << " " << rootNode->Event->GetMonth\_Char(rootNode->Event->GetMonth()) << " " << rootNode->Event->GetYear() << "| Weekday: " << rootNode->Event->GetWeek\_Day(rootNode->Event->GetDay(), rootNode->Event->GetMonth(), rootNode->Event->GetYear());

std::cout << "| Time: " << rootNode->Event->GetHour() << " " << rootNode->Event->GetMin() << "| Importance: " << rootNode->Event->GetImportant() <<std::endl<< "Place: " << rootNode->Event->GetPlace() <<std::endl<< "Description: " << rootNode->Event->GetDescription() << std::endl << std::endl;

}

u--;

if (!isEmpty(rootNode->Left)) show\_all\_events(rootNode->Left, sort, ++u); //Left

} //end sort decrease

}

void pause\_key(char\* message\_text)

{

std::cout<<message\_text;

while (!(\_getch())) {

//any key

};

system("CLS");

}

void search\_place(Node\* rootNode) {

if ((event\_to\_search == 0) && (isEmpty(rootNode))) { //if tree null to exit

filtr\_search = false;

return;

}

if (!isEmpty(rootNode->Left)) search\_place(rootNode->Left);//Left

event\_to\_search++;

if (filtr\_place== rootNode->Event->GetPlace()) {

filtr\_search = true;

return;

}

if (!isEmpty(rootNode->Right)) search\_place(rootNode->Right); //Right

}

void case\_sort\_filtr() {

system("CLS");

int sort\_num;

bool st = true;

for (;;) {

if (st) {

st = false;

if (filtr\_place== "") std::cout<<"Filtering by place All"<<std::endl<<std::endl;

else std::cout << "Filtering by place "<<filtr\_place << std::endl << std::endl;

std::cout<<"Enter the sort type:\n\n1 - Increase date\n\n2 - Decrease date\n\n3 - Increase important\n\n4 - Decrease important\n\n9 - Filter place\n\n0 - Return to main menu\n\n"<<std::endl;

}

std::cin>>sort\_num;

if ((sort\_num > 0) && (sort\_num < 5)) {

system("CLS");

event\_number = 0;

if (sort\_num == 1 || sort\_num == 2) {

sort\_type = 1;

LoadfromFile("my\_notebook.tmp");

show\_all\_events(root, sort\_num, 0);

}

if (sort\_num == 3 || sort\_num == 4) {

sort\_type = 2;

LoadfromFile("my\_notebook.tmp");

show\_all\_events(root, sort\_num, 0);

}

pause\_key("...Please enter any key to show menu...");

st = true;

}

else

if (sort\_num == 9) {

system("CLS");

event\_number = 0;

show\_all\_events(root, sort\_num, 0);

std::cout<<std::endl<<"Enter filtering by place. All - all place."<<std::endl;

for (;;) {

std::cin>>filtr\_place;

event\_to\_search = 0;

if (filtr\_place=="All") {

filtr\_place="";

break;

}

filtr\_search = false;

search\_place(root);

if (filtr\_search) {

break;

}

else {

std::cout << "There is no such place "<<filtr\_place<<". Re-enter place\n";

}

}

system("CLS");

st = true;

delete root;

root = NULL;

}

else

if (sort\_num == 0) {

filtr\_place="";

system("CLS");

return;

}

else std::cout<<"Wrong input. Enter correctly\n\n";

}

}

void random\_event() {

srand(time(NULL));

int k = 1;

int rne = 0;

//Node\* node=new Node;

char\* d\_array[] = { "Hello world","People is crazy","Go to the Museum","Buy a pencil","Perform work",

"Paint the wall","Congratulate on the birthday of the city","To discover their talents",

"Pass laboratory works","Meet friends" };

char\* p\_array[] = { "Saint-Petersburg","Kronshtadt","Novgorod","Tver","Vologda",

"Valdai","Strelna","Petrodvorets","Pargolovo","Pushkin" };

std::cout<<"Enter the number of events 1-10000"<<std::endl;

while (k != 0) {

std::cin>>rne;

if (rne < 1 || rne > 10000)

std::cout<<"Wrong input. Enter the number correctly"<<std::endl;

else k = 0;

}

for (int i = 0; i < rne; i++) {

Event\* newEvent = new Event;

newEvent->SetHour(rand() % 24) ;

newEvent->SetMin(rand() % 60) ;

newEvent->SetImportant(rand() % 11) ;

newEvent->SetYear(rand() % 5 + 2015) ;

newEvent->SetMonth(rand() % 12 + 1) ;

if (newEvent->GetMonth()==2) {

newEvent->SetDay(rand() % 28 + 1) ;

}

else {

if (newEvent->GetMonth() == 4 || newEvent->GetMonth() == 6 || newEvent->GetMonth() == 9 || newEvent->GetMonth() == 11) {

newEvent->SetDay(rand() % 30 + 1);

}

else {

newEvent->SetDay(rand() % 31 + 1);

}

}

newEvent->SetDescription(d\_array[rand() % 10]);

newEvent->SetPlace(p\_array[rand() % 10]);

root=insert(newEvent,sort\_type,root);

}

}

void Save\_to\_File(Node\* t) {

if (t == NULL) { //if tree null to exit

return;

}

else

{

fprintf(my\_file, "%d\n%d\n%d\n%d\n", t->Event->GetDay(), t->Event->GetMonth() , t->Event->GetYear() , t->Event->GetHour());

fprintf(my\_file, "%d\n%d\n%s\n%s\n", t->Event->GetMin(), t->Event->GetImportant(), t->Event->GetPlace().c\_str(), t->Event->GetDescription().c\_str());

if (t->Left != NULL) Save\_to\_File(t->Left);//Left

if (t->Right != NULL) Save\_to\_File(t->Right); //Right

}

}

int convert\_to\_int(std::string buf) {

char conv[4] = "";

for (int k = 0; k <= sizeof(buf); k++) {

if (buf[k] != '\n') conv[k] = buf[k];

else { break; }

}

return atoi(conv);

}

void Load\_from\_File(Node\* t) {

char buf[1000] = "";

char desc[1000]="";

char place[100]="";

int i = 0;

int poz = 0;

delete root;

root = NULL;

Event\* newEvent = new Event;

while (!feof(my\_file))

{

buf[i] = fgetc(my\_file);

i++;

if ((buf[i - 1] == '\n') && (poz == 0)) {

i = 0; poz++;

newEvent->SetDay(convert\_to\_int(buf));

memset(buf, NULL, sizeof(buf));

}

if (buf[i - 1] == '\n' && poz == 1) {

i = 0; poz++;

newEvent->SetMonth(convert\_to\_int(buf));

memset(buf, NULL, sizeof(buf));

}

if (buf[i - 1] == '\n' && poz == 2) {

i = 0; poz++;

newEvent->SetYear(convert\_to\_int(buf));

memset(buf, NULL, sizeof(buf));

}

if (buf[i - 1] == '\n' && poz == 3) {

i = 0; poz++;

newEvent->SetHour(convert\_to\_int(buf));

memset(buf, NULL, sizeof(buf));

}

if (buf[i - 1] == '\n' && poz == 4) {

i = 0; poz++;

newEvent->SetMin(convert\_to\_int(buf));

memset(buf, NULL, sizeof(buf));

}

if (buf[i - 1] == '\n' && poz == 5) {

i = 0; poz++;

newEvent->SetImportant(convert\_to\_int(buf));

memset(buf, NULL, sizeof(buf));

}

if (buf[i - 1] == '\n' && poz == 6) {

i = 0; poz++;

memset(place, NULL, sizeof(place));

for (int k = 0; k < sizeof(buf); k++) {

if (buf[k] == '\n') { break; }

place[k] = buf[k];

}

newEvent->SetPlace(place);

memset(buf, NULL, sizeof(buf));

}

if (buf[i - 1] == '\n' && poz == 7) {

i = 0; poz++;

memset(desc, NULL, sizeof(desc));

for (int k = 0; k < sizeof(buf); k++) {

if (buf[k] == '\n') { break; }

desc[k] = buf[k];

}

newEvent->SetDescription(desc);

memset(buf, NULL, sizeof(buf));

}

if (poz == 8) {

poz = 0; i = 0;

Event\* newEvent2 = new Event;

newEvent2->SetDay(newEvent->GetDay());

newEvent2->SetMonth(newEvent->GetMonth());

newEvent2->SetYear(newEvent->GetYear());

newEvent2->SetImportant(newEvent->GetImportant());

newEvent2->SetHour(newEvent->GetHour());

newEvent2->SetMin(newEvent->GetMin());

newEvent2->SetPlace(newEvent->GetPlace());

newEvent2->SetDescription(newEvent->GetDescription());

root = insert(newEvent2, sort\_type, root);

}

}//end while

}

class Node {

public:

Node\* Left=NULL;

Node\* Right=NULL;

Event\* Event;

};

Node \*root;

std::string filtr\_place="";

bool filtr\_search = false;

int sort\_type = 1; // 1 - sort to date; 2 - sort to imp

int event\_to\_search = 0;

int event\_number = 0;

FILE\* my\_file;

};