Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное образовательное учреждение высшего образования

«Пермский национальный исследовательский политехнический университет»

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы» направление подготовки: 09.03.04 – «Программная инженерия»

Лабораторная работа №8.

" Программа, управляемая событиями "

15 вариант

Выполнил студент гр. РИС-24-2б

Иванова Елена Олеговна

Проверил:

Доц. Каф. ИТАС

Ольга Андреевна Полякова

(оценка) (подпись)

(дата)

г. Пермь, 2025

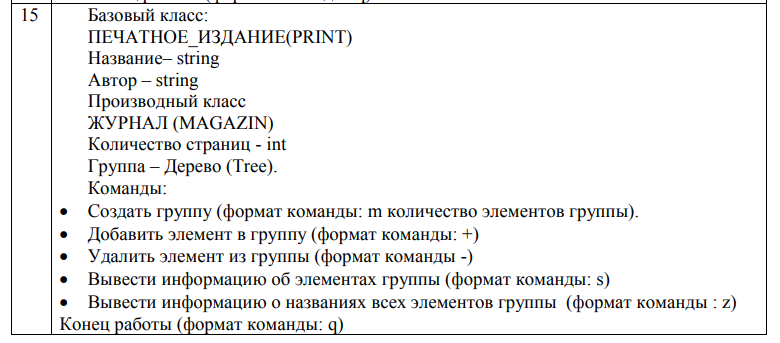
Цель задания:

1. Создание консольного приложения, состоящего из нескольких файлов в системе программирования Visual Studio.
2. Разработка программы, управляемой событиями.

Постановка задачи:

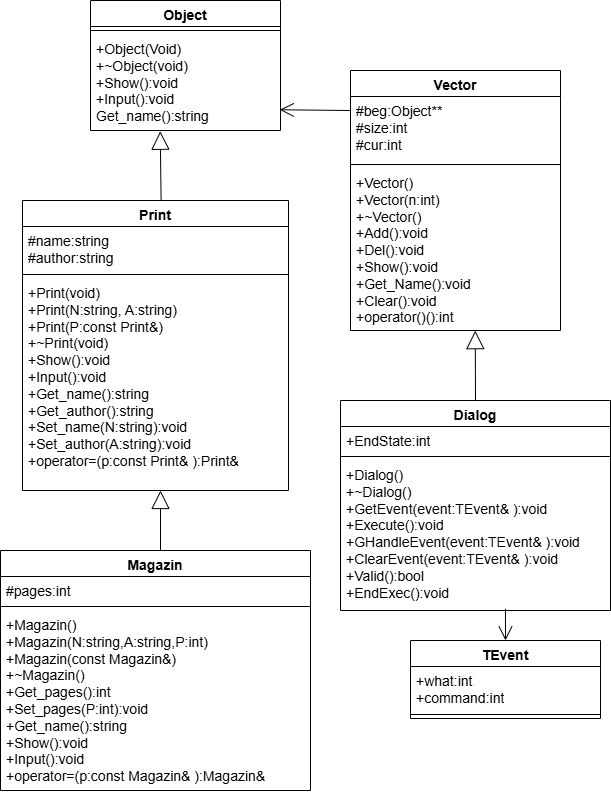
1. Определить иерархию пользовательских классов (см. лабораторную работу №5). Во главе иерархии должен стоять абстрактный класс с чисто виртуальными методами для ввода и вывода информации об атрибутах объектов.
2. Реализовать конструкторы, деструктор, операцию присваивания, селекторы и модификаторы.
3. Определить класс-группу на основе структуры, указанной в варианте.
4. Для группы реализовать конструкторы, деструктор, методы для добавления и удаления элементов в группу, метод для просмотра группы, перегрузить операцию для получения информации о размере группы.
5. Определить класс Диалог – наследника группы, в котором реализовать методы для обработки событий.
6. Добавить методы для обработки событий группой и объектами пользовательских классов.
7. Написать тестирующую программу.
8. Нарисовать диаграмму классов и диаграмму объектов.

Задача:

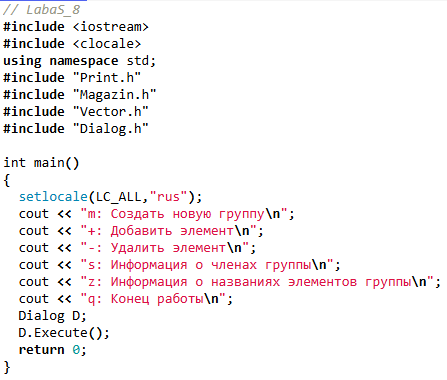
Аналил задачи.

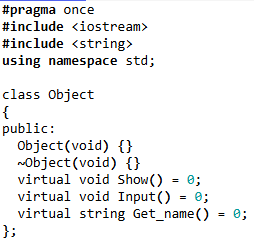
1. Создать абстрактный класс
2. Написать базовый класс на основе абстрактного
3. Написать производный класс на основе базового
4. Создать вектор и структуру с событиями
5. Создать класс Диалог на основе класса вектор.
6. Реализовать команды, данные в варианте

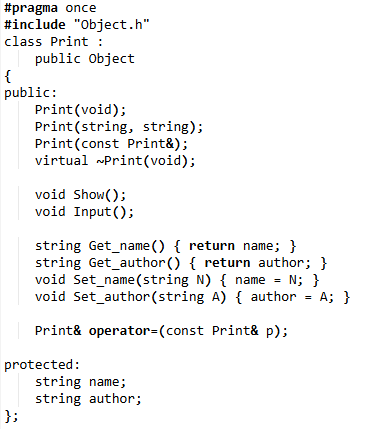
Диаграмма.

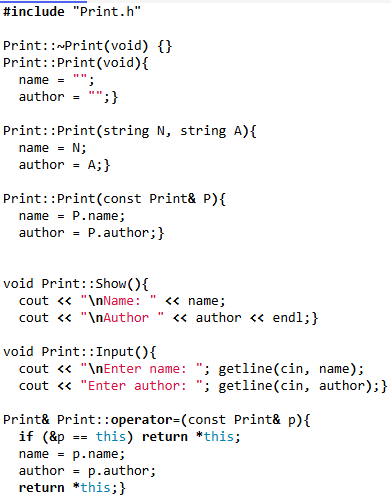


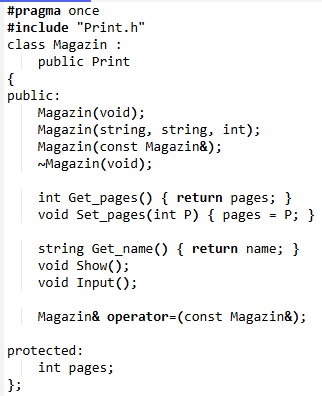
Код.

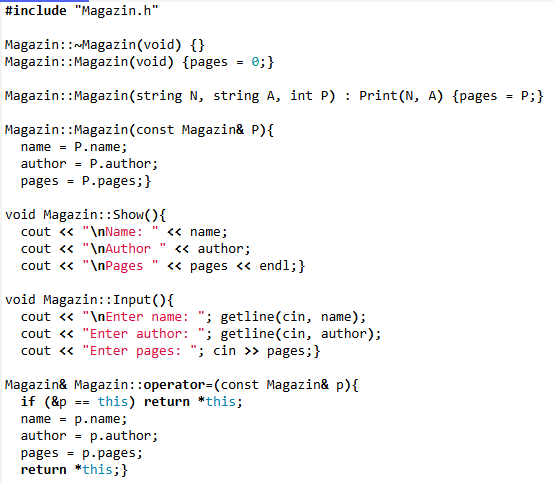


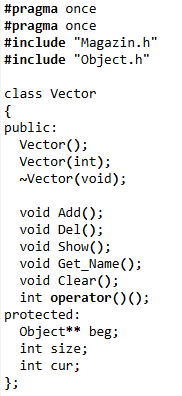


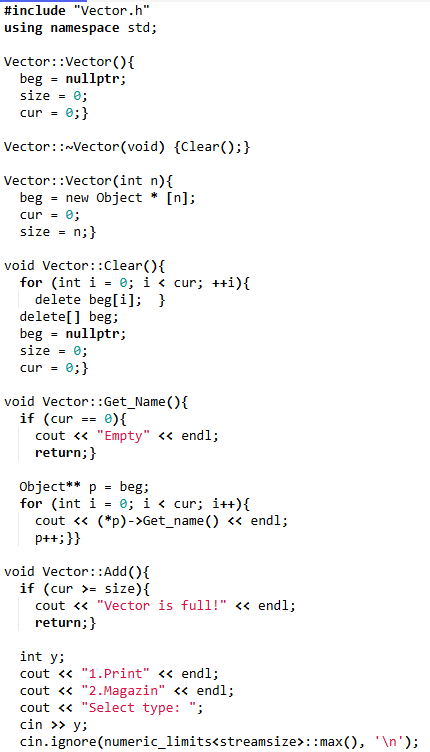


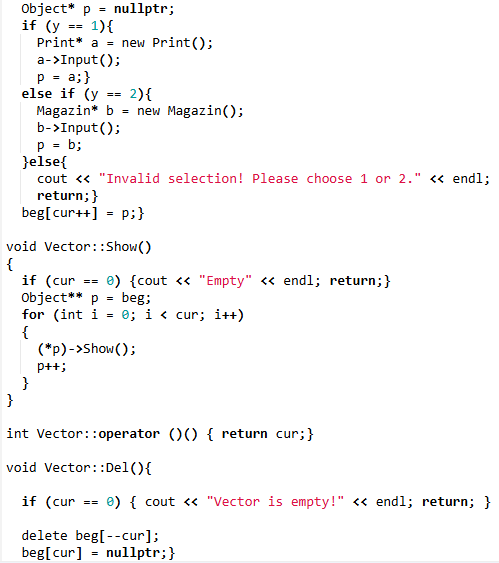


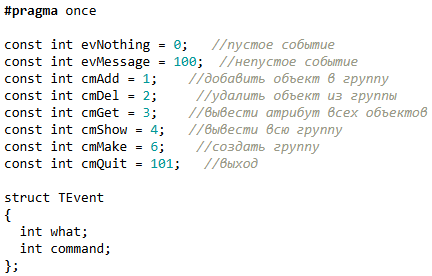


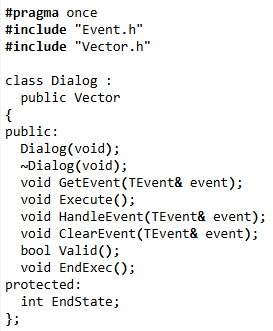


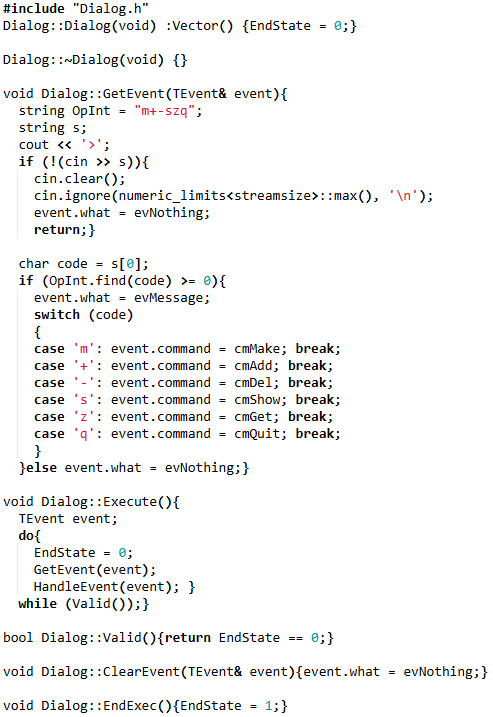


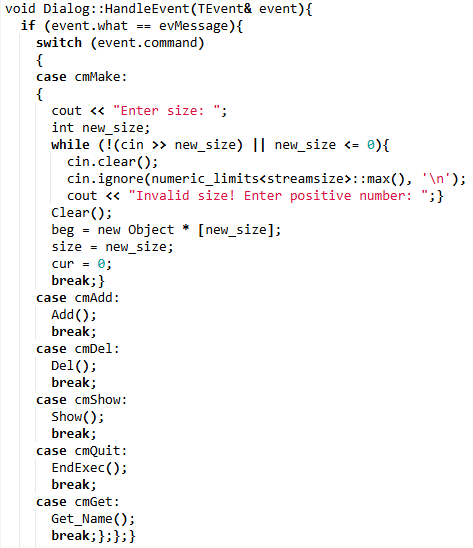




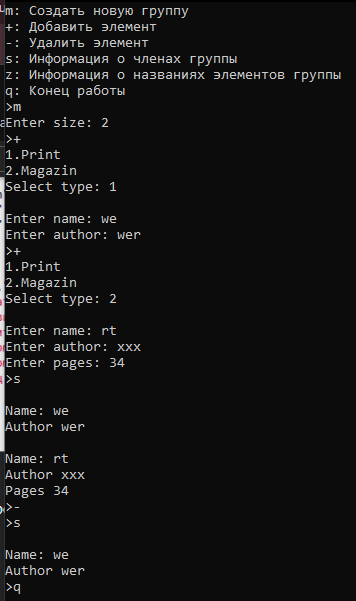




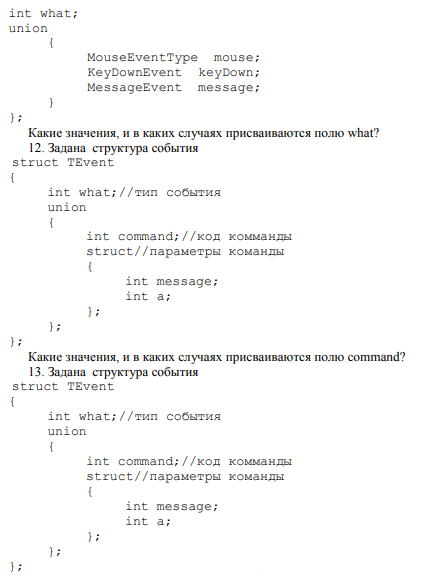
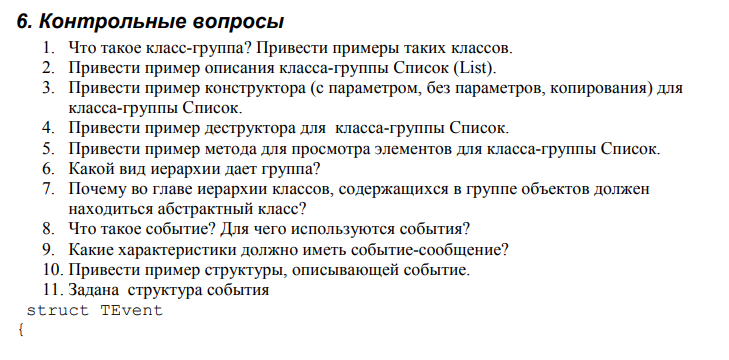


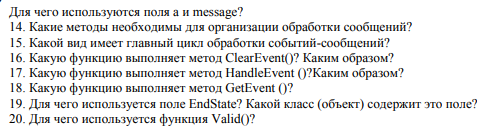


Вывод.



Контрольные вопросы.





1. Группа это объект, в который включены другие объекты. Объекты, входящие в группу, называются элементами группы. Элементы группы, в свою очередь, могут быть группой. В отличие от контейнера мы понимаем группу как класс, который не только хранит объекты других классов, но и обладает собственными свойствами, не вытекающими из свойств его элементов. Группа дает второй вид иерархии - иерархию объектов (иерархию типа целое/часть), построенную на основе агрегации, первый вид иерархия классов, построенная на основе наследования.

Пример:

class Vector {

public:

Vector(int);

~Vector(void);

void Add();

void Del();

void Show();

int operator()();

protected:

Object\*\* beg;

1. struct Node {

Node\* prev = nullptr, \* next = nullptr;

Object\* data; };

class List {

public:

List();

List(int);

List(List&);

~List(void);

void puchback();

void popback();

void Show();

protected:

Node\* head = nullptr, \* tail = nullptr;

int size; };

1. Без параметров:

List::List() {

size = 0;

head = nullptr;

tail = nullptr;

}

С параметрами:

List::List(int s) {

for (int i = 0; i < size ; i++)

pushback();

}

Копирования:

List::List(const List& a) {

Node\* nodeA = a.head;

List newList(a.size);

Node\* newNode = newList.head;

while (nodeA != nullptr) {

newNode->data = nodeA->data;

nodeA = nodeA->next;

newNode = newNode->next;

}}

1. List::~List() {

Node\* curr = head;

while (curr != nullptr) {

head = curr->next;

delete curr->data;

delete curr;

curr = head;}}

1. void List::Show() {

if (size == 0) cout << "Empty" << endl;

Node\* p = head;

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

p->data->Show();

p = p->next; }}

1. Группа дает второй вид иерархии - иерархию объектов(иерархию типа целое/часть), построенную на основе агрегации, первый вид иерархия классов, построенная на основе наследования.
2. Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах.

Таким образом, можно создать функцию, параметром которой является указатель на абстрактный класс. На место этого параметра при выполнении программы может передаваться указатель на объект любого производного класса. Это позволяет создавать полиморфные функции, работающие с объектом любого типа в пределах одной иерархии.

Именно поэтому во главе иерархии классов должен находится абстрактный класс.

1. События лучше всего представить себе как пакеты информации, которыми обмениваются объекты и которые создаются объектно-ориентированной средой в ответ на те или иные действия пользователя. Нажатие на клавишу или манипуляция мышью порождают событие, которое передается по цепочке объектов, пока не найдется объект, знающий, как обрабатывать это событие.

События сами по себе не производят никаких действий в программе, но в ответ на событие могут создаваться новые объекты, модифицироваться или уничтожаться существующие, что и приводит к изменению состояния программы. Иными словами все действия по обработке данных реализуются объектами, а события лишь управляют их работой.

События предназначены для того, чтобы иметь возможность предусмотреть реакцию программы на то или иное действие.

1. Сообщение передаваемое от одних объектов другим имеет, как правило, следующие характеристики:

* код класса сообщения, отличающий сообщения объектов одного класса от объектов другого класса;
* адрес объекта, которому предназначено сообщение (м.б. не задан, тогда сообщение могут прочитать все объекты);
* информационное поле.

1. const int evNothing = 0; //пустое событие

const int evMessage = 100; //непустое событие

const int cmAdd = 1; // добавить элемент в группу

const int cmDel = 2; // удалить элемент из группы

const int cmShow = 3; // вывести информацию об элементах группы

const int cmMake = 4; // создать группу

const int cmQuit = 101; // выйти из программы

struct TEvent {

int what; //наличие события

int command; //тип команды};

1. Поле what задает тип события, определяющий источник данного события. Оно может принимать следующие значения:

⦁ evNothing - это пустое событие, которое означает, что ничего делать не надо. Полю what присваивается значение evNothing, когда событие обработано каким-либо объектом.

⦁ evMessage - событие-сообщение от объекта (непустое событие).

1. Полю command присваиваются коды различных определённых команд. При получении того или иного сообщения, поле command принимает одно из кодов команд (см. пример из вопроса 10).
2. Поля a и message необходимы в случае, если программа получает сообщение с параметром. Тогда поле message будет отвечать за то, какая команда поступила, а поле a будет являться параметром команды.
3. ⦁ GetEvent – формирование события;

⦁ Execute реализует главный цикл обработки событий. Он постоянно получает событие путем вызова GetEvent и обрабатывает их с помощью HandleEvent. Этот цикл завершается, когда поступит событие «конец».

⦁ HandleEvent – обработчик событий. Обрабатывает каждое событие нужным для него образом. Если объект должен обрабатывать определенное событие (сообщение), то его метод HandleEvent должен распознавать это событие и реагировать на него должным образом. Событие может распознаваться, например, по коду команды (поле command).

⦁ ClearEvent очищает событие, когда оно обработано, чтобы оно не обрабатывалось далее.

⦁ Valid - проверяет, завершена ли работа.

⦁ EndExec - завершение обработки событий (после вызова этого метода цикл обработки событий заканчивается).

1. Главный цикл обработки событий реализуется в методе Execute главной группы-объекта “прикладная программа” по следующей схеме:

int TMyApp::Execute() {

do {

endState=0;

GetEvent(event); //получить событие

HandleEvent(event); //обработать событие

//событие осталось не обработано

if(event.what!=evNothing)

EventError(event);

} while(!Valid());

return endState;}

1. ClearEvent очищает событие, присваивая полю event.what значение evNothing.
2. Получив событие (структуру типа TEvent), обработчик событий для класса TDerivedClass обрабатывает его по следующей схеме:

void TDerivedClass::HandleEvent(TEvent& event) {

//Вызов обработчика событий базового класса

TBaseClass::HandleEvent( event );

if (event.what == evCommand) { // Если обработчик базового

// событие не обработал

switch( event.message.command ) {

case cmCommand1:

// Обработка команды cmCommand1;

СlearEvent( event );

break;

case cmCommand2:

// Обработка команды cmCommand2;

СlearEvent( event );

break; …

case cmCommandN:

// Обработка команды cmCommandN

СlearEvent( event );

break;

default: // событие не обработано

break;} }; }

1. Метод GetEvent() выполняет функцию формирования события на основе полученного сообщения
2. Поле EndState используется в классе Dialog и отвечает за состояние работы диалога (идёт/завершён).

class Dialog :

public Vector {

public:

Dialog(void);

~Dialog(void);//деструктор

void GetEvent(TEvent& event);//получить событие

void Execute();//главный цикл обработки событий

void HandleEvent (TEvent& event); //обработчик

void ClearEvent (TEvent & event);//очистить событие

bool Valid();//проверка атрибута EndState

void EndExec();//обработка события «конец работы»

protected:

int EndState;};

1. Функция Valid проверяет состояние поля EndState. Она возвращает значение True, если программа не завершена, False - если завершена. Используется в главном обработчике событий Execute.

bool Dialog::Valid() {

return EndState == 0;}

void Dialog::Execute() {

TEvent event;

do {

EndState = 0;

GetEvent(event); //получить событие

HandleEvent(event); //обработать событие

} while (Valid()); }