СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА № 1
Создание классов и объектов

Цель: научиться создавать классы и объекты этих классов, вызывать методы и обращаться к данным через методы этих объектов, передавать объекты других классов через методы, понимать влияние модификаторов доступа на видимость данных и методов.

Теоретическая информация

Понятия класса и объекта

Объект – это сущность, выделяемая из словаря предметной области, которая характеризуется набором свойств и событийным поведением. Последнее обычно представляется в виде множества событий, из которых образуется жизненный цикл объекта. В более распространенной терминологии для именования этих двух аспектов описания объекта используют термины *атрибут* и *метод*. Атрибутами могут выступать как переменные стандартных типов данных, так и объекты других классов.

Класс – фундаментальное понятие объектно-ориентированного подхода. Это абстракция, обозначающая некоторое подмножество объектов, имеющих сходные черты как в наборе свойств, так и в поведении. На C++ создание (описание) класса выполняется с помощью следующей конструкции:

```
тип_класса имя_класса{список_членов_класса};
```

где `тип_класса` – одно из служебных слов `class`, `struct`, `union`;
`имя_класса` – идентификатор;
`список_членов_класса` – определения атрибутов и методов класса.

Пример:

```
class Window {
     int x;               // координаты левого верхнего угла
     int y;               //
     int Width, Height;   // ширина и высота
public:
     void Hide(){/*реализация*/};        // скрыть
     void Show(){/*реализация*/};        // показать
     void Paint(){/*реализация*/};       // прорисовать
} ;
```

Создание объектов

Для создания объекта используется конструкция, аналогичная определению переменной:

```
имя_класса имя_объекта;
```

Пример:

```
car my_car, car2;
car *carp = &car2;        // указатель на объект
car cars[30];             // массив объектов
car &c = my_car;          // ссылка на объект
```

Второй способ создавать указатель на объект с помощью операции `new`:

```
Имя_класса * имя_указателя = new
Имя_конструктора(параметры_конструктора);
```

Пример:

```
Window *w1 = new Window();
```

Обращаться к данным (атрибутам) объекта и вызывать методы объекта, если они отмечены как `public`, можно двумя способами. Первый способ – с помощью конструкций:

```
имя_объекта.имя_данного
имя_объекта.имя_функции
```

Пример:

```
Window w1, w2;
w1.x = 3.4;
w1.y = 0.3;
w1.Show();
w2.Paint();
```

Второй способ – используя указатель на объект:

```
указатель_на_объект->имя_данного
указатель_на_объект->имя_функции
```

Пример:

```
Window *w1 = new Window();
w1->x = 3.4;              // возможно, если x public
w1->y = 0.3;              // возможно, если y public
w1->Show();
```

Видимость атрибутов и методов

Для изменения видимости атрибутов и методов в определении класса можно использовать спецификаторы доступа: `public`, `private`, `protected`.

Общедоступные (`public`) – доступны в любой части программы. Они могут использоваться любой функцией как внутри данного класса, так и вне его.

Собственные (`private`) атрибуты и методы могут использоваться только в классе и недоступны извне. Они могут использоваться также функциями – «друзьями» данного класса. Если модификатор доступа не указан, то члены класса по умолчанию считаются `private`.

Защищенные (`protected`) компоненты доступны внутри класса и в классах-наследниках.

Считается, что данные (атрибуты) класса должны быть скрыты от внешнего доступа, т. е. описаны как `private`. Доступ к таким данным осуществляется через методы (`get`, `set`), которые описываются в классе как `public`.

Пример:

```
class Text {
AnsiString s;              // private по умолчанию
public:
AnsiString& getText(void);              // доступно извне
};
```

Перегрузка методов

В одном классе могут быть реализованы *несколько методов с одинаковым именем* (перегрузка). Такое возможно при соблюдении одного требования: набор параметров методов должен быть разным.

Задание

1. Создать приложение согласно выбранному варианту.
2. В приложении должно быть описано 2 класса (основной и дополнительный), в каждом из которых должны быть закрытые (`private`) данные и открытые (`public`) методы.
3. Хотя бы в одном из классов должно быть несколько методов с одинаковыми именами, но с разным набором параметров для инициализации сразу нескольких атрибутов класса.
4. При этом в основном классе должен быть массив объектов дополнительного класса.

5. Для добавления в этот массив должен использоваться метод, которому в параметрах должен передаваться объект дополнительного класса.

6. В теле основной программы необходимо создать объект основного класса, заполнить его данными через методы, заполнить массив объектами дополнительного класса, вывести содержимое основного объекта, включая весь массив дополнительных объектов.

7. Сделать выводы.

Пример упрощенного варианта реализации:

```
#include "pch.h"
#include <iostream>
#include <string>

using namespace std;

class Article {                 // дополнительный класс
      string name;              // закрытые данные
      string author;
public:
      string getName() {        // открытые методы
            return name;
      }
      void setName(string name) {
            this->name = name;
      }
      string getAuthor() {
            return author;
      }
      void setAuthor(string author) {
            this->author = author;
      }
      void setProperties() {          // метод с одинаковым именем 1
            string str;
            cout << "   Name:" << endl;
            getline(cin, str);
            this->setName(str);
            cout << "   Author:" << endl;
            getline(cin, str);
            this->setAuthor(str);
      }
      void setProperties(string str1, string str2) {              // метод
                                                  // с одинаковым именем 2
            this->setName(str1);
            this->setAuthor(str2);
      }
};

class Newspaper {               // основной класс
```

```
	string name;			// закрытые данные
	Article arr[10];		// массив объектов дополнительного
					// класса
public:
	string getName() {		// открытые методы
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	void setArticle(Article art, int i) {	// метод для добавления
					// объекта дополнительного класса в массив
		arr[i] = art;
	}
	Article getArticle(int i) {
		return arr[i];
	}
};


int main()
{
	Newspaper np;			// создание объекта основного класса
	string str1, str2;
	cout << "Newspaper:" << endl;
	getline(cin, str1);
	np.setName(str1);

	for (int i = 0; i < 5; i++) {
		cout << " Article " << i << ":" << endl;
		Article art;
		art.setProperties();		// заполнение методом 1
		np.setArticle(art, i);
	}
	for (int i = 5; i < 10; i++) {
		cout << " Article " << i << ":" << endl;
		Article art;
		cout << "   Name:" << endl;
		getline(cin, str1);
		cout << "   Author:" << endl;
		getline(cin, str2);
		art.setProperties(str1, str2);	// заполнение методом 2
		np.setArticle(art, i);
	}
	cout << endl << "Print Newspaper:" << endl;	// вывод данных
							// на экран
	cout << np.getName() << endl;
	for (int i = 0; i < 10; i++) {
		cout << "   Name:" << np.getArticle(i).getName() << "   Author:"
			<< np.getArticle(i).getAuthor() << endl;
	}
}
```

Варианты

Вариант 1

Тема проекта: приложение «Журнал регистрации корреспонденции».

В приложении должно быть реализовано 2 класса: *Журнал регистрации* и *Корреспонденция*. Класс *Журнал* содержит *номер*, *имя*. Класс *Корреспонденция* соержит *дату*, *тему*, *отправителя*.

Вариант 2

Тема проекта: приложение «Учет денежных средств».

В приложении должно быть реализовано 2 класса: *Журнал учета* и *Платеж*. Класс *Журнал* содержит *карту*, *имя*. Класс *Платеж* содержит *дату*, *сумму*, *описание*.

Вариант 3

Тема проекта: приложение «Энциклопедия «История в лицах».

В приложении должно быть реализовано 2 класса: *Энциклопедия* и *Личность*. Класс *Энциклопедия* содержит *название*, *год издания*. Класс *Личность* содержит *имя*, *род деятельности*, *описание*.

Вариант 4

Тема проекта: приложение «Касса кинотеатра».

В приложении должно быть реализовано 2 класса: *Кинотеатр* и *Сеанс*. Класс *Кинотеатр* содержит *название*, *адрес*. Класс *Сеанс* содержит *дату*, *время*, *название*.

Вариант 5

Тема проекта: приложение «Календарь планирования мероприятий».

В приложении должно быть реализовано 2 класса: *Календарь* и *Мероприятие*. Класс *Календарь* содержит *название помещения*, *адрес*. Класс *Мероприятие* содержит *дату*, *название*, *описание*.

Вариант 6

Тема проекта: приложение «Каталог кинофильмов».

В приложении должно быть реализовано 2 класса: *Каталог* и *Фильм*. Класс *Каталог* содержит *название*, *год создания*. Класс *Фильм* содержит *название*, *жанр*, *описание*.

Вариант 7

Тема проекта: приложение «Библиотека».

В приложении должно быть реализовано 2 класса: *Библиотека* и *Книга*. Класс *Библиотека* содержит *название*, *адрес*. Класс *Книга* содержит *название*, *жанр*, *автор*.

Вариант 8

Тема проекта: приложение «Отдел кадров».

В приложении должно быть реализовано 2 класса: *Отдел* и *Работник*. Класс *Отдел* содержит *название*, *телефон*. Класс *Работник* содержит *имя*, *должность*, *дату приема*.

Вариант 9

Тема проекта: приложение «Управление пользователями».

В приложении должно быть реализовано 2 класса: *Приложение* и *Пользователь*. Класс *Приложение* содержит *название*, *требуемый объем* (Мб). Класс *Пользователь* содержит *логин*, *пароль*, *роль*.

Вариант 10

Тема проекта: приложение «Расписание занятий».

В приложении должно быть реализовано 2 класса: *Расписание* и *Занятие*. Класс *Расписание* содержит *номер учебного заведения*, *адрес*. Класс *Занятие* содержит *название*, *день недели*, *время*.

Вариант 11

Тема проекта: приложение «Журнал учителя».

В приложении должно быть реализовано 2 класса: *Журнал* и *Урок*. Класс *Журнал* содержит *название предмета*, *имя учителя*. Класс *Урок* содержит *дату*, *тему*, *номер класса*.

Вариант 12

Тема проекта: приложение «Учет клиентов».

В приложении должно быть реализовано 2 класса: *Журнал регистрации* и *Клиент*. Класс *Журнал регистрации* содержит *название организации*, *телефон*. Класс *Клиент* содержит *имя*, *адрес*, *номер договора*.

Вариант 13

Тема проекта: приложение «Тестирование знаний».

В приложении должно быть реализовано 2 класса: *Тест* и *Вопрос*. Класс *Тест* содержит *название предмета*, *сложность*. Класс *Вопрос* содержит *текст вопроса*, *ответ*, *количество баллов*.

Вариант 14

Тема проекта: приложение «Организация экскурсий».

В приложении должно быть реализовано 2 класса: *Журнал регистрации* и *Экскурсия*. Класс *Журнал регистрации* содержит *Название организации*, *контактный телефон*. Класс *Корреспонденция* содержит *дату*, *имя экскурсовода*, *описание*.

Вариант 15

Тема проекта: приложение «Телефонный справочник».

В приложении должно быть реализовано 2 класса: *Справочник* и *Контакт*. Класс *Справочник* содержит *название*, *имя владельца*. Класс *Контакт* содержит *имя*, *телефон*, *адрес*.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Приведите листинг программы.
3. Разместите скриншоты результатов выполнения программы.
4. Сделайте краткие пояснения к алгоритму работы.
5. Сформулируйте выводы.

Дополнительные вопросы

1. Можно ли определить несколько методов с одинаковым именем в одном классе?
2. Какие модификаторы доступа существуют и как они влияют на видимость данных и методов?
3. Как правильно организовать доступ к данным объекта?
4. Как обращаться к данным и методам объекта?
5. Можно ли объекту одного класса передать объект другого класса?

ЛАБОРАТОРНАЯ РАБОТА № 2
Конструкторы и деструкторы

Цель: научиться описывать в классе конструкторы различных видов (конструктор по умолчанию, конструктор с параметрами, конструктор копирования), деструктор; научиться создавать объекты при помощи различных конструкторов; понимать время жизни объекта.

Теоретическая информация

Конструктор. Виды конструкторов

Конструктор – выделяет память для объекта и инициализирует данные-члены класса:

– имя конструктора совпадает с именем класса;

– конструктор не имеет возвращаемого значения (даже `void`);

– классу без конструктора предоставляется конструктор по умолчанию;

– при явно описанном конструкторе конструктор по умолчанию не генерируется;

– конструкторы могут быть перегружены;

– не могут быть описаны с ключевыми словами `virtual`, `static`, `const`, `mutable`, `volatile`;

– не могут явно вызываться в программе.

Конструктор представляет собой обычную функцию, имя которой совпадает с именем класса, в котором он объявлен и используется. Количество и имена фактических параметров в описании функции конструктора чаще зависят от числа полей, которые будут инициализированы при объявлении объекта (экземпляра) данного класса. Кроме указанной формы записи конструктора в программах на C++ можно встретить и следующую форму записи конструктора:

```
kls(int A, int B, int C) :  a(A), b(B), c(C) { }
```

В этом случае после двоеточия перечисляются инициализируемые данные, а в скобках – инициализирующие их значения (точнее, через запятую перечисляются конструкторы объектов соответствующих типов). Возможна комбинация отмеченных форм.

Наряду с перечисленными выше формами записи конструктора существует конструктор, либо не имеющий параметров, либо все аргументы которого заданы по умолчанию – *конструктор по умолчанию*:

```
kls(){ }  для примера выше, аналогично  kls() :  a(), b(),
c() { }
kls(int=0, int=0, int=0){ } аналогично kls() :  a(0),
b(0), c(0) { }
```

Каждый класс может иметь только один конструктор по умолчанию. Конструктор по умолчанию используется при создании объекта без его инициализации, а также незаменим при создании массива объектов. Если при этом конструкторы с параметрами в классе есть, а конструктора по умолчанию нет, то компилятор зафиксирует синтаксическую ошибку.

В классе может быть объявлено (и определено) несколько конструкторов. Их объявления должны различаться списками параметров. Такие конструкторы по аналогии с функциями называются перегруженными (или совместно используемыми). Транслятор различает перегруженные конструкторы по спискам параметров. В этом смысле конструктор не отличается от обычной функции-члена класса:

```
ComplexType(double rePar, double imPar);
/*Объявление… */
ComplexType(double rePar, double imPar) {   /*…*/   }
/*Определение…*/
```

Конструктор вызывается не для объекта класса, как другие функции-члены, а для области памяти с целью её преобразования («превращения») в объект класса.

Деструктор

Противоположным по отношению к конструктору является деструктор – функция, приводящая к разрушению объекта соответствующего класса и возвращающая системе область памяти, выделенную конструктором. Деструктор имеет имя, аналогичное имени конструктора, но перед ним ставится знак ~:

```
~kls(void){}   или   ~kls(){}        // функция-деструктор
```

Деструктор вызывается автоматически при разрушении объекта, его задача – освободить память и корректно уничтожить объект:

– имя деструктора также совпадает с именем класса но предваряется символом тильда «~» (`~имя_класса`);

– деструктор не имеет никакого возвращаемого значения;

– не может быть описан как `static`, `const`, `mutable`, `volatile`;

– без явного задания генерируется деструктор по умолчанию;

– деструкторы могут быть описаны как `virtual` – у всех производных классов деструкторы автоматически будут виртуальными;

– могут явно вызываться.

Конструктор копирования

Частным случаем конструктора является *конструктор копирования*, у которого в аргументах передается ссылка на другой объект того же класса.

Необходимость использования конструктора копирования вызвана тем, что объекты наряду со статическими могут содержать и динамические данные. В то же время, например, при передаче объекта в качестве параметра функции в ней создается локальная (в пределах функции), копия этого объекта. При этом указатели обоих объектов будут содержать один и тот же адрес области памяти. При выводе локального объекта из поля видимости функции для его разрушения вызывается деструктор. В функцию деструктора входит также освобождение динамической памяти, адрес которой содержит указатель. При окончании работы программы (при вызове деструкторов), производится повторная попытка освободить уже освобожденную ранее память. Это приводит к ошибке. Для устранения этого в класс необходимо добавить конструктор копирования, который в качестве единственного параметра получает ссылку на объект класса. Общий вид конструктора копирования имеет следующий вид:

```
имя_класса (const имя_класса &);
```

В этом конструкторе выполняется выделение памяти и копирование в нее информации из объекта, получаемого по ссылке. Таким образом, указатели для обоих объектов содержат разные адреса памяти, и при вызове деструктора выполняется освобождение памяти, соответствующей каждому из объектов.

```
#include <iostream.h>
#include <stdlib.h>
#define n 3
// ------------------     объявление     konstr_copy.h
class cls
{  char *str;
```

```
    int dl;
          // другие данные класса
  public:
    cls ();          // конструктор по умолчанию
    cls(cls &);      // копирующий конструктор
    ~cls();          // деструктор
         // другие методы класса
};
// ------------------    реализация    konstr_copy.cpp
#include "konstr_copy.h"
cls::cls ()
{ dl=10;
   str=new char[dl];
}
cls::cls(cls & obj1)  // копирующий конструктор из obj1 в obj
{ dl=obj1.dl;         // копирование длины строки
   str=new char[dl];   // выделение памяти "под" строку длиной dl
   strcpy(str,obj1.str);  // копирование строки
}
cls::~cls()
{ delete [] str;
   cout<<"деструктор"<<endl;
}

void fun(cls obj1)
{  // код функции
    cout<<"выполняется функция"<<endl;
}

void main(void)
{ cls obj;
  //   .  .  .
   fun(obj);
  //  .  .  .
}
```

Если для класса конструктор копирования явно не описан, то компилятор сгенерирует его. При этом значения компоненты-данного одного объекта будут скопированы в компоненту-данное другого объекта. Это допустимо для объектов простых классов и недопустимо для объектов, имеющих динамические компоненты-данные (конструируются с использованием операторов динамического выделения памяти). Таким образом, даже если в классе не используются динамические данные, желательно явно описывать конструктор копирования.

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 1.
2. Определить в классах следующие конструкторы: без параметров, с параметрами, копирования.
3. Определить в классе деструктор.
4. В каждом конструкторе и деструкторе выдавать сообщение, показывающее, какой именно конструктор или деструктор был вызван.
5. Дополнить основной класс методом с параметрами, внутри которого должен создаваться объект дополнительного класса с помощью конструктора с параметрами и добавляться в массив объектов дополнительного класса.
6. Дополнить основной класс методом с двумя параметрами (ссылка на объект дополнительного класса и количество), внутри которого должны создаваться объекты дополнительного класса с помощью конструктора копирования (создается необходимое количество копий указанного объекта) и заноситься в массив объектов дополнительного класса.
7. Дополнить/модифицировать основное тело программы таким образом, чтобы продемонстрировать использование всех конструкторов.
8. Сделать выводы.

Пример упрощенного варианта реализации

```
#include "pch.h"
#include <iostream>
#include <string>

using namespace std;

class Article {
	string name;
	string author;
public:
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	string getAuthor() {
		return author;
	}
	void setAuthor(string author) {
		this->author = author;
	}
```

```
	Article() {		// конструктор без параметров (по умолчанию)
		name = "No name";
		author = "No author";
		cout << "Article: Default constructor" << endl;
	}
	Article(string _name, string _author) {		// конструктор
												// с параметрами
		name = _name;
		author = _author;
		cout << "Article: Constructor with param" << endl;
	}
	Article(Article &art) {		// конструктор копирования
		name = art.name;
		author = art.author;
		cout << "Article: Copy constructor" << endl;
	}
	~Article() {		// деструктор
		cout << "Article: Destructor" << endl;
	}
};

class Newspaper {
	string name;
	Article arr[10];
public:
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	void setArticle(Article art, int i) {
		arr[i] = art;
	}
	Article getArticle(int i) {
		return arr[i];
	}
	Newspaper() {		// конструктор без параметров (по умолчанию)
		name = "No name";
		cout << "Newspaper: Default constructor" << endl;
	}
	Newspaper(string _name) {		// конструктор с параметрами
		name = _name;
		cout << "Newspaper: Newspaper: Constructor with param" << endl;
	}
	Newspaper(Newspaper &np) {		// конструктор копирования
		name = np.name;
		cout << "Newspaper: Copy constructor" << endl;
	}
	~Newspaper() {		// деструктор
```

```
            cout << "Newspaper: Destructor" << endl;
      }
      void setNewArticle(string _name, string _author, int i) {
                                                      // новый метод
            Article art(_name, _author);           // создание объекта
                                          // конструктором с параметрами
            arr[i] = art;                 // добавление объекта в массив
      }
      void setCopyArticle(Article &art, int k) {     //новый метод 2
            for (int i = 0; i < k || i < 10; i++) {
                  Article newart(art);    // создание объекта
                                          // конструктором копирования
                  arr[i] = newart;        // добавление объекта в массив
            }
      }
};

int main()
{
      cout << "-=-=-=-=-=-=-=-=-=-= Step 1" << endl;
      Newspaper np1;
      Newspaper np2("Vestnik");
      Newspaper np3(np2);

      cout << "-=-=-=-=-=-=-=-=-=-= Step 2" << endl;
      Article art1;
      Article art2("Top news", "Ivanov");

      cout << "-=-=-=-=-=-=-=-=-=-= Step 3" << endl;
      np1.setNewArticle("First news", "Popov", 1);
      np2.setCopyArticle(art2,10);
      cout << "-=-=-=-=-=-=-=-=-=-= End" << endl;

}
```

Варианты

Варианты распределяются аналогично лабораторной работе № 1.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Приведите листинг программы.
3. Разместите скриншоты результатов выполнения программы.
4. Сделайте краткие пояснения к алгоритму работы.
5. Сформулируйте выводы.

Дополнительные вопросы

1. Какие конструкторы класса существуют?
2. Как создать конструктор с параметрами?
3. В чем особенность передачи объекта методу в качестве параметра?
4. Когда будет разрушен объект, созданный внутри метода?
5. Когда разрушаются объекты, созданные динамически?

ЛАБОРАТОРНАЯ РАБОТА № 3
Наследование

Цель: научиться создавать иерархию классов, создавать объекты базовых классов и классов-наследников, вызывать методы базового класса и класса-наследника, понимать влияние модификаторов доступа при наследовании, понимать порядок вызова конструкторов и деструкторов при наследовании.

Теоретическая информация

Понятие наследования

Наследование является фундаментальным понятием объектно-ориентированного программирования и представляет собой формализацию такого механизма манипулирования абстракциями, как обобщение. Фактически это отношение, которое связывает два класса, один из которых выступает в роли обобщения для другого. При этом отпадает необходимость дублировать в обоих классах код и данные, которые мы выносим в *обобщенный класс*. В более распространенной терминологии обобщенный класс называют *суперклассом* или *родительским классом*, а детализированный класс – *подклассом*, *производным* или *дочерним классом*.

Производный класс наследует описание родительского класса, но он может быть изменен путем добавлением новых членов, изменением кода существующих методов и изменением прав доступа. С помощью наследования может быть создана *иерархия классов*. Наследуемые методы и атрибуты не перемещаются в производный класс, а остаются в базовых классах.

Допускается множественное наследование – возможность для некоторого класса наследовать несколько никак не связанных между собой классов. В иерархии классов соглашение относительно доступности членов класса следующее:

– `private` – член класса может использоваться только функциями-членами данного класса и функциями-«друзьями» своего класса. В производном классе он недоступен;

– `protected` – то же, что и `private`, но дополнительно член класса с данным атрибутом доступа может использоваться функциями-членами и функциями-«друзьями» классов, производных от данного;

– public – член класса может использоваться любой функцией, которая является членом данного или производного класса, а также к public-членам возможен доступ извне через имя объекта.

Следует иметь в виду, что объявление друга (friend) не является атрибутом доступа и не наследуется.

Синтаксис определения производного класса:

```
class имя_класса : список_базовых_классов
{список_компонентов_класса};
```

В производном классе унаследованные компоненты получают статус доступа:

– private, если новый класс определен с помощью ключевого слова class;

– public, если новый класс определен с помощью ключевого слова struct.

Явно изменить умалчиваемый статус доступа при наследовании можно с помощью атрибутов доступа private, protected и public, которые указываются непосредственно перед именами базовых классов.

Конструкторы и деструкторы производных классов

Поскольку конструкторы не наследуются, при создании производного класса наследуемые им данные-члены должны инициализироваться конструктором базового класса. Конструктор базового класса вызывается автоматически и выполняется до конструктора производного класса. Параметры конструктора базового класса указываются в определении конструктора производного класса. Таким образом происходит передача аргументов от конструктора производного класса конструктору базового класса.

Пример:

```
class Basic {
      int i;
public:
      Basis(int x){ i=x; }
};

class Inherit: public Basis {
      int count;
```

```
public:
      Inherit(int x,int c): Basis(x){ count=c; }
};
```

Объекты класса конструируются снизу вверх: сначала базовый, потом компоненты-объекты (если они есть), а потом сам производный класс. Таким образом, объект производного класса содержит в качестве подобъекта объект базового класса. Уничтожаются объекты в обратном порядке: сначала производный, потом его компоненты-объекты, а потом базовый объект.

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 2.
2. Создать несколько классов-наследников для дополнительного класса (т. е. дополнительный класс станет базовым для новых создаваемых классов). Каждый класс-наследник должен включать в себя атрибуты, отличающие его от других классов.
3. Определить в новых классах конструкторы и деструктор.
4. В каждом конструкторе и деструкторе выдавать сообщение, показывающее, какой именно конструктор или деструктор и какого класса был вызван.
5. Определить в новых классах методы для ввода и вывода данных как самого класса-наследника, так и базового класса. Метод вывода данных должен иметь то же имя, что и метод вывода данных в базовом классе.
6. Дополнить основное тело программы созданием объектов базового класса, объектов классов-наследников, инициализацией данных и выводом на экран этих объектов без занесения их в массив объектов в основном классе из лабораторной работы № 1 (в этой работе данный класс не задействован).
7. Создать массив объектов, тип массива – базовый класс. Поместить в массив как объекты базового, так и объекты классов-наследников.
8. В цикле пройти по всем элементам созданного в п. 7 массива, вызвать метод вывода данных на экран.
9. Создать массив объектов. Тип массива – один из классов-наследников. Поместить в массив объекты этого класса-наследника.
10. В цикле пройти по всем элементам созданного в п. 9 массива, вызвать метод вывода данных на экран.
11. Сделать выводы.

Пример упрощенного варианта реализации

```
#include "pch.h"
#include <iostream>
#include <string>

using namespace std;

class Article {
	string name;
	string author;
public:
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	string getAuthor() {
		return author;
	}
	void setAuthor(string author) {
		this->author = author;
	}
	Article() {
		name = "No name";
		author = "No author";
		cout << "Article: Default constructor" << endl;
	}
	Article(string _name, string _author) {
		name = _name;
		author = _author;
		cout << "Article: Constructor with param" << endl;
	}
	Article(Article &art) {
		name = art.name;
		author = art.author;
		cout << "Article: Copy constructor" << endl;
	}
	~Article() {
		cout << "Article: Destructor" << endl;
	}
	void print() {                          // метод вывода
		cout << "Article -=-=- Name: " << getName() << " Author: "
                                              << getAuthor() << endl;
	}
};

class Scientific: public Article {          // класс-наследник 1
	string science;
```

```
public:
	string getScience() {
		return science;
	}
	void setScience(string science) {
		this->science = science;
	}
	Scientific() {
		science = "No science";
		cout << "Scientific: Default constructor" << endl;
	}
	Scientific(string _name, string _author, string science):
Article(_name, _author){
		this->science = science;
		cout << "Scientific: Constructor with param" << endl;
	}
	~Scientific() {
		cout << "Scientific: Destructor" << endl;
	}
	void print() {                          // метод вывода
		cout << "Scientific -=-=- Name: " << getName() << " Author: "
			<< getAuthor() << " Science: " << getScience() << endl;
	}
};

class News : public Article {               // класс-наследник 2
	string area;
public:
	string getArea() {
		return area;
	}
	void setArea(string area) {
		this->area = area;
	}
	News() {
		area = "No area";
		cout << "News: Default constructor" << endl;
	}
	News(string _name, string _author, string area) {
		this->setName(_name);
		this->setAuthor(_author);
		this->area = area;
		cout << "News: Constructor with param" << endl;
	}
	~News() {
		cout << "News: Destructor" << endl;
	}
	void print() {                          // метод вывода
```

```
            cout << "News -=-=- Name: " << getName() << " Author: " <<
                getAuthor() << " Area: " << getArea() << endl;
        }
};

int main()
{
        cout << "-==-==-==-==-= Step1" << endl;
        Article art1;
        Article art2("First article", "Ivanov");

        Scientific sc1;
        Scientific sc2("Biology article", "Petrov", "ecology");

        News n1;
        News n2("Top news", "Sidorov", "Politics");

        cout << "-==-==-==-==-= Step2" << endl;
        Article arr1[6];                        // массив базового класса
        arr1[0] = art1;
        arr1[1] = art2;
        arr1[2] = sc1;
        arr1[3] = sc2;
        arr1[4] = n1;
        arr1[5] = n2;

        for (int i = 0; i < 6; i++) {
                arr1[i].print();
        }

        cout << "-==-==-==-==-= Step3" << endl;
        Scientific arr2[2];             // массив наследников
        arr2[0] = sc1;
        arr2[1] = sc2;

        for (int i = 0; i < 2; i++) {
                arr2[i].print();
        }
}
```

Варианты

Вариант 1

Тема проекта: приложение *Журнал регистрации корреспонденции*.
Добавить классы: *Электронное письмо*, *Почтовое письмо*.

Вариант 2

Тема проекта: приложение *Учет денежных средств*.
Добавить классы: *Доходы*, *Расходы*.

Вариант 3

Тема проекта: приложение *Энциклопедия «История в лицах»*.
Добавить классы: *Живописцы*, *Писатели*.

Вариант 4

Тема проекта: приложение *Касса кинотеатра*.
Добавить классы: *Бронирование*, *Оплаченные билеты*.

Вариант 5

Тема проекта: приложение *Календарь планирования мероприятий*.
Добавить классы: *Разовое мероприятие*, *Повторяющееся мероприятие*.

Вариант 6

Тема проекта: приложение *Каталог кинофильмов*.
Добавить классы: *Избранное*, *Заблокированное*.

Вариант 7

Тема проекта: приложение *Библиотека*.
Добавить классы: *Основной зал*, *Читальный зал*.

Вариант 8

Тема проекта: приложение *Отдел кадров*.
Добавить классы: *Работники*, *Уволенные*.

Вариант 9

Тема проекта: приложение *Управление пользователями*.
Добавить классы: *Администратор*, *Гость*.

Вариант 10

Тема проекта: приложение *Расписание занятий*.
Добавить классы: *Лабораторная работа*, *Лекция*.

Вариант 11

Тема проекта: приложение *Журнал учителя*.
Добавить классы: *Зачет*, *Экзамен*.

Вариант 12

Тема проекта: приложение *Учет клиентов*.
Добавить классы: *Физические лица*, *Юридические лица*.

Вариант 13

Тема проекта: приложение *Тестирование знаний*.
Добавить классы: *Тест по теме*, *Контрольный тест*.

Вариант 14

Тема проекта: приложение *Организация экскурсий*.
Добавить классы: *Местные экскурсии*, *Международные экскурсии*.

Вариант 15

Тема проекта: приложение *Телефонный справочник*.
Добавить классы: *Данные юридических и физических лиц*.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Приведите листинг программы.
3. Разместите скриншоты результатов выполнения программы.
4. Сделайте краткие пояснения к алгоритму работы.
5. Сформулируйте выводы.

Дополнительные вопросы

1. Как создать иерархию классов?
2. Как влияют модификаторы доступа на видимость данных и методов при наследовании?
3. Можно ли обращаться к методам базового класса через объекты класса-наследника?
4. В каком порядке вызываются конструкторы и деструкторы при наследовании?
5. Можно ли поместить объекты базового класса и классов-наследников в один массив?

ЛАБОРАТОРНАЯ РАБОТА № 4
Полиморфизм и виртуальные методы

Цель: научиться создавать абстрактные классы и определять виртуальные методы внутри этих классов, переопределять виртуальные методы внутри классов-наследников, создавать списки из объектов различных классов, понимать механизм позднего связывания.

Теоретическая информация

Виртуальные функции

Виртуальные функции предоставляют механизм *позднего* (*отложенного*), или *динамического связывания*. Любая нестатическая функция базового класса может быть сделана виртуальной, для чего используется ключевое слово `virtual`:

```
virtual тип имя_функции (список_формальных_параметров);
```

Виртуальность наследуется. После того как функция определена как виртуальная, ее повторное определение в производном классе (с тем же самым прототипом) создает в этом классе новую виртуальную функцию, причем спецификатор `virtual` может не использоваться.

Конструкторы не могут быть виртуальными в отличие от деструкторов. Практически каждый класс, имеющий виртуальную функцию, должен иметь виртуальный деструктор.

Абстрактные классы

Абстрактным называется класс, в котором есть хотя бы одна чистая (пустая) виртуальная функция. *Чистой виртуальной функцией* называется функция, которая имеет следующее определение:

```
virtual тип имя_функции (список_формальных_параметров) = 0;
```

Чистая виртуальная функция ничего не делает и недоступна для вызовов. Ее назначение – служить основой для подменяющих ее функций в производных классах. Абстрактный класс в C++ – это конструкция, которая в других объектно-ориентированных языках программирования и в UML фиксируется в таком понятии, как *интерфейс*. В UML введен стереотип «Interface» для изображения класса.

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 3.
2. Определить в базовом классе (он же дополнительный класс из лабораторной работы № 1) чисто виртуальную функцию.
3. Перегрузить в классах-наследниках (классы, созданные в лабораторной работе № 3) виртуальную функцию: реализовать вывод всех данных как базового класса, так и класса-наследника.
4. В основном теле программы создать массив объектов, тип массива – базовый класс. Поместить в массив объекты всех классов-наследников.
5. В цикле пройти по всем элементам созданного в п. 4 массива, вызвать метод вывода данных на экран. Сравнить с результатом лабораторной работы № 3.
6. Модифицировать основной класс (из лабораторной работы № 1) таким образом, чтобы его внутренний массив мог содержать объекты любых классов-наследников (из лабораторной № 3), и добавить метод вывода всех данных (включая массив) на экран.
7. В основном теле программы продемонстрировать работу с объектом основного класса.
8. Сделать выводы.

Пример упрощенного варианта реализации:

```
#include "pch.h"
#include <iostream>
#include <string>

using namespace std;

class Article {
	string name;
	string author;
public:
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	string getAuthor() {
		return author;
	}
```

```
	void setAuthor(string author) {
		this->author = author;
	}
	Article() {
		name = "No name";
		author = "No author";
		cout << "Article: Default constructor" << endl;
	}
	Article(string _name, string _author) {
		name = _name;
		author = _author;
		cout << "Article: Constructor with param" << endl;
	}
	~Article() {
		cout << "Article: Destructor" << endl;
	}

	virtual void print() = 0;			// чисто виртуальный метод
};

class Scientific : public Article {		// класс-наследник 1
	string science;
public:
	string getScience() {
		return science;
	}
	void setScience(string science) {
		this->science = science;
	}
	Scientific() {
		science = "No science";
	}
	Scientific(string _name, string _author, string science):
Article(_name, _author){
		this->science = science;
	}
	~Scientific() {
	}
	void print() {				// метод вывода
		cout << "Scientific -=-=- Name: " << getName() <<
		 " Author: " << getAuthor() << " Science: " <<
		 getScience() << endl;
	}
};

class News : public Article {			// класс-наследник 2
	string area;
public:
	string getArea() {
		return area;
	}
	void setArea(string area) {
```

```
		this->area = area;
	}
	News() {
		area = "No area";
	}
	News(string _name, string _author, string area) {
		this->setName(_name);
		this->setAuthor(_author);
		this->area = area;
	}
	~News() {
	}
	void print() {                          // метод вывода
		cout << "News -=-=- Name: " << getName() << " Author: " <<
			getAuthor() << " Area: " << getArea() << endl;
	}
};

class Newspaper {                           // основной класс
	string name;
	Article* arr[4];
public:
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	void setArticle(Article* art, int i) {
		arr[i] = art;
	}
	Article* getArticle(int i) {
		return arr[i];
	}
	void show() {                           // метод вывода
		cout << "-=-=- Newspaper -=-=- Name: " << getName() << endl;
		for (int i = 0; i < 4; i++) {
                                arr[i]->print();
		}
	}
};


int main()
{
	cout << "-=-=-=-=-=-= Step1" << endl;
	Scientific* sc1 = new Scientific();
	Scientific* sc2 = new Scientific("Biology article", "Petrov",
	"ecology");

	News* n1 = new News();
```

```
    News* n2 = new News("Top news", "Sidorov", "Politics");

    cout << "-==========-=-= Step2" << endl;
    Article* arr1[4];            // массив указателей базового класса
    arr1[0] = sc1;
    arr1[1] = sc2;
    arr1[2] = n1;
    arr1[3] = n2;

    for (int i = 0; i < 4; i++) {
        arr1[i]->print();
    }

    cout << "-==========-=-= Step3" << endl;
    Newspaper np;                 // объект основного класса
    np.setName("Vestnik");
    for (int i = 0; i < 4; i++) {
        np.setArticle(arr1[i], i);
    }
    np.show();

    delete sc1;
    delete sc2;
    delete n1;
    delete n2;

}
```

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Приведите листинг программы.
3. Разместите скриншоты результатов выполнения программы.
4. Сделайте краткие пояснения к алгоритму работы.
5. Сформулируйте выводы.

Дополнительные вопросы

1. Как описать абстрактный класс?
2. Для чего применяют виртуальные функции?

3. Как переопределить виртуальные функции?
4. Что произойдет, если в классе-наследнике не переопределить чисто виртуальную функцию?
5. Можно ли создавать списки объектов абстрактного класса?

ЛАБОРАТОРНАЯ РАБОТА № 5
Статические члены класса

Цель: научиться определять статические данные и методы в классе, обращаться к статическим данным и методам, в том числе не создавая объекты класса, понимать различия между статическими и нестатическими данными и методами класса.

Теоретическая информация

Статические члены класса определены в классе с ключевым словом `static`. Статические данные классов не дублируются при создании объектов. Доступ к статическому компоненту возможен только после его инициализации. Для инициализации используется конструкция

```
тип имя_класса :: имя_данного = инициализирующее_значение;
```

Это предложение должно быть размещено в глобальной области после определения класса. К статическим данным и коду можно обращаться и тогда, когда объект класса еще не существует. Доступ к статическим членам возможен не только через имя объекта, но и через имя класса:

```
имя_класса : : имя_компонента
```

Однако так можно обращаться только к `public`-членам.

Обращение к `private`-статической компоненте извне возможно с помощью статических методов.

Пример:

```
#include <iostream.h>
class Car {
     int speed;
     static int CarCount;    // статический атрибут –
                             // количество машин
public:
Car(int s){ speed = s; CarCount++; }
static int& count(){ return CarCount; };
}

int Car::CarCount = 0;       // инициализация статического
                             // атрибута

void main(void) {
```

```
Car c1(20); Car c2(30);
cout << ” Количество машин = “ << Car::count();
}
```

Указатель `this` является дополнительным скрытым параметром каждого нестатического метода. При входе в тело принадлежащего классу метода `this` инициализируется значением адреса того объекта, для которого вызвана функция. В результате этого объект становится доступным внутри этой функции.

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 4.
2. Определить в базовом классе закрытый (`private`) статический счетчик объектов.
3. Определить статический метод для получения информации о количестве созданных объектов.
4. Модифицировать конструкторы и деструкторы базового класса: увеличивать счетчик в конструкторе и уменьшать в деструкторе.
5. В основном классе определить закрытый статический счетчик объектов, помещенных во внутренний массив и соответствующий ему метод для получения информации о количестве объектов в массиве.
6. В основном теле программы вывести количество объектов, не создавая ни одного объекта базового класса или классов-наследников.
7. Создать объекты классов-наследников и поместить их во внутренний массив объекта основного класса.
8. Снова вывести количество созданных объектов и количество объектов во внутреннем массиве объекта основного класса.
9. Сделать выводы.

Пример упрощенного варианта реализации:

```
#include "pch.h"
#include <iostream>
#include <string>

using namespace std;

class Article {
	string name;
	string author;
```

```
	static int countArticle;				// статический счетчик
public:
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	string getAuthor() {
		return author;
	}
	void setAuthor(string author) {
		this->author = author;
	}
	Article() {
		countArticle++;					// увеличение счетчика
		name = "No name";
		author = "No author";
	}
	Article(string _name, string _author) {
		countArticle++;					// увеличение счетчика
		name = _name;
		author = _author;
	}
	~Article() {
		countArticle--;					// уменьшение счетчика
	}
	virtual void print() = 0;
	static int getCount() {				//статический метод
		return countArticle;
	}
};

int Article::countArticle = 0;

class Scientific : public Article {
	string science;
public:
	string getScience() {
		return science;
	}
	void setScience(string science) {
		this->science = science;
	}
	Scientific() {
		science = "No science";
	}
		this->science = science;
	}
	~Scientific() {
	}
```

```
	void print() {
		cout << "Scientific -=-=- Name: " << getName() <<
		" Author: " << getAuthor() << " Science: " <<
		getScience() << endl;
	}
};

class News : public Article {
	string area;
public:
	string getArea() {
		return area;
	}
	void setArea(string area) {
		this->area = area;
	}
	News() {
		area = "No area";
	}
	News(string _name, string _author, string area) {
		this->setName(_name);
		this->setAuthor(_author);
		this->area = area;
	}
	~News() {
	}
	void print() {
		cout << "News -=-=- Name: " << getName() << " Author: " <<
		getAuthor() << " Area: " << getArea() << endl;
	}
};

class Newspaper {                              // основной класс
	string name;
	Article* arr[4];
	static int countArr;
public:
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	void setArticle(Article* art, int i) {
		arr[i] = art;
		countArr++;
	}
	Article* getArticle(int i) {
		return arr[i];
	}
```

```
	void show() {
		cout << "-=-=- Newspaper -=-=- Name: " << getName() << endl;
		for (int i = 0; i < 4; i++) {
			arr[i]->print();
		}
	}
	static int getCountArray() {
		return countArr;
	}
};

int Newspaper::countArr = 0;

int main()
{
	cout << "-=-=-=-=-=-= Step1" << endl;
	cout << "Create objects: " << Article::getCount() << endl;
	cout << "Objects in array: " << Newspaper::getCountArray() <<
	endl;

	cout << "-=-=-=-=-=-= Step2" << endl;

	Scientific* sc1 = new Scientific();
	Scientific* sc2 = new Scientific("Biology article", "Petrov",
	"ecology");

	News* n1 = new News();
	News* n2 = new News("Top news", "Sidorov", "Politics");

	cout << "Create objects: " << Article::getCount() << endl;
	cout << "Objects in array: " << Newspaper::getCountArray() <<
	endl;

	cout << "-=-=-=-=-=-= Step3" << endl;
	Newspaper np;                          // объект основного класса
	np.setName("Vestnik");
	np.setArticle(sc1, np.getCountArray());
	np.setArticle(sc2, np.getCountArray());
	np.setArticle(n1, np.getCountArray());
	np.setArticle(n2, np.getCountArray());

	cout << "Create objects: " << Article::getCount() << endl;
	cout << "Objects in array: " << Newspaper::getCountArray() <<
	endl;

	np.show();

	delete sc1;
	delete sc2;
	delete n1;
	delete n2;

}
```

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Приведите листинг программы.
3. Разместите скриншоты результатов выполнения программы.
4. Сделайте краткие пояснения к алгоритму работы.
5. Сформулируйте выводы.

Дополнительные вопросы

1. В чем особенность статических данных класса? В чем их отличие от нестатических?
2. В чем особенность статических методов класса?
3. Могут ли статические методы работать с нестатическими данными класса?
4. Как работать со статическими данными или методами класса, не создавая объекта этого класса?
5. Для чего могут применяться статические данные или методы?

ЛАБОРАТОРНАЯ РАБОТА № 6
Перегрузка операторов

Цель: научиться перегружать бинарные и унарные операторы для пользовательского класса, применять перегруженные операторы для операций над объектами классов.

Теоретическая информация

Перегрузка операций

Перегрузка операций – это возможность использовать знаки стандартных операций для записи выражений как для встроенных, так и для абстрактных типов данных. В языке C++ для перегрузки операций используется ключевое слово `operator`, с помощью которого определяется специальная операция-функция (`operator function`). Ее формат:

```
тип_возвр_значения operator знак_операции
(специф_параметров)
{операторы_тела_функции}
```

Перегрузка унарных операций

Любая унарная операция ⊕ может быть определена двумя способами: либо как компонентная функция без параметров, либо как глобальная функция с одним параметром. В первом случае выражение `⊕Z` означает вызов `Z.operator⊕ ()`, во втором – вызов `operator⊕(Z)`. Унарные операции, перегружаемые в рамках определенного класса, могут перегружаться только через нестатическую функцию без параметров. Вызываемый объект класса автоматически воспринимается как операнд. Унарные операции, перегружаемые вне области класса, должны иметь один параметр типа класса. Передаваемый через этот параметр объект воспринимается как операнд.

Пример:

```
class audio_player {
      int records_count;
      int current_record;
      ...
public:
      play(int i);
      void init(void){ current_record = 0; }
      void operator++ (){ //++a
```

```
            current_record++;
            if(current_record>records_count) current_record = 0;
        }
        void operator++ (int i){ //a++
            current_record++;
            if(current_record>records_count) current_record = 0;
        }
        void operator-(){ //--a
            current_record--;
            if(current_record<0) current_record =
records_count - 1;
        }
} ;
        void main(){
            audio_plajer a;
            a.init(); ++a; a++;
}
```

Перегрузка бинарных операций

Любая бинарная операция ⊕ может быть определена двумя способами: либо как функция-член класса с одним параметром, либо как глобальная функция с двумя параметрами. В первом случае `x⊕y` означает вызов `x.operator⊕ (y)`, во втором – вызов `operator⊕ (x,y)`.

Операции, перегружаемые внутри класса, могут перегружаться только нестатическими функциями с параметрами. Вызываемый объект класса автоматически воспринимается в качестве первого операнда. Операции, перегружаемые вне области класса, должны обладать двумя операндами, один из которых обязан иметь тип класса.

Пример:

```
class Car{
        int model; int year; int color;
public:
        bool __fastcall operator== (Car& c){
          return (model==c.model) && (year==c.year) && (color==c.color);
        }
} ;
```

Перегрузка операции присваивания

Операция отличается тремя особенностями:

1) операция не наследуется;

2) операция определена по умолчанию для каждого класса в качестве операции поразрядного копирования объекта, стоящего справа от знака операции, в объект, стоящий слева.

3) операция может перегружаться только в области определения класса. Это гарантирует, что первым операндом всегда будет леводопустимое выражение.

Формат перегруженной операции присваивания:

```
имя_класса& operator=(имя_класса& объект);
```

Отметим две важные особенности функции `operator=`. Во-первых, в ней используется параметр-ссылка. Это необходимо для предотвращения создания копии объекта, передаваемого через параметр по значению. Во-вторых, функция `operator=()` возвращает не объект, а ссылку на него. Смысл этого тот же, что и при использовании параметра-ссылки: избежать создания временного объекта, в качестве возвращаемого значения используется ссылка.

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 5.
2. Определить в основном классе оператор + для добавления объекта одного из классов-наследников во внутренний массив.
3. Определить в основном классе операторы ++ префиксный и постфиксный для добавления объектов по умолчанию в массив (созданных с помощью конструктора по умолчанию).
4. Определить в основном классе оператор `[]` для доступа к элементу массива по индексу.
5. Определить глобальный оператор << для вывода данных на экран.
6. В основном теле программы продемонстрировать применение всех операторов.
7. Сделать выводы.

Пример упрощенного варианта реализации:

```
#include "pch.h"
#include <iostream>
#include <string>

using namespace std;
```

```
class Article {
	string name;
	string author;
public:
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	string getAuthor() {
		return author;
	}
	void setAuthor(string author) {
		this->author = author;
	}
	Article() {
		name = "No name";
		author = "No author";
	}
	Article(string _name, string _author) {
		name = _name;
		author = _author;
	}
	~Article() {
	}
	virtual void print() = 0;
};

class Scientific : public Article {
	string science;
public:
	string getScience() {
		return science;
	}
	void setScience(string science) {
		this->science = science;
	}
	Scientific() {
		science = "No science";
	}
	Scientific(string _name, string _author, string science): Arti-
cle(_name, _author){
		this->science = science;
	}
	~Scientific() {
	}
	void print() {
		cout << "Scientific -=-=- Name: " << getName() << " Author: "
			<< getAuthor() << " Science: " << getScience() << endl;
	}
};
```

```
class News : public Article {
	string area;
public:
	string getArea() {
		return area;
	}
	void setArea(string area) {
		this->area = area;
	}
	News() {
		area = "No area";
	}
	News(string _name, string _author, string area) {
		this->setName(_name);
		this->setAuthor(_author);
		this->area = area;
	}
	~News() {
	}
	void print() {
		cout << "News -=-=- Name: " << getName() << " Author: "
		<< getAuthor() << " Area: " << getArea() << endl;
	}
};

class Newspaper {                              // основной класс
	string name;
	Article* arr[4];
	static int countArr;
public:
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	void setArticle(Article* art, int i) {
		arr[i] = art;
		countArr++;
	}
	Article* getArticle(int i) {
		return arr[i];
	}
	void show() {
		cout << "-=-=- Newspaper -=-=- Name: " << getName() << endl;
		for (int i = 0; i < getCountArray(); i++) {
			arr[i]->print();
		}
	}
	static int getCountArray() {
```

```
            return countArr;
      }
      void operator+ (Article* art) {
            this->setArticle(art, getCountArray());
      }
      Article* operator[](int i) {
            return this->arr[i];
      }
};

ostream& operator << (ostream &os, Newspaper &np)   // глобальная
                                                    // перегрузка оператора <<
{
      return os << np.getName();
}

int Newspaper::countArr = 0;

int main()
{
      Scientific* sc1 = new Scientific();
      Scientific* sc2 = new Scientific("Biology article", "Petrov",
      "ecology");

      News* n1 = new News();
      News* n2 = new News("Top news", "Sidorov", "Politics");

      Newspaper np;
      np.setName("Vestnik");
      np + sc1;
      np + sc2;
      np.show();

      np + n1;
      np + n2;
      np.show();

      cout << "-=-=- Operator [] -=-=- " << endl;
      np[3]->print();

      cout << "-=-=- Operator << -=-=- " << endl;
      cout << np;

      delete sc1;
      delete sc2;
      delete n1;
      delete n2;
}
```

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Приведите листинг программы.
3. Разместите скриншоты результатов выполнения программы.
4. Сделайте краткие пояснения к алгоритму работы.
5. Сформулируйте выводы.

Дополнительные вопросы

1. Чем отличаются бинарные и унарные операторы?
2. Как реализовать префиксные и постфиксные унарные операторы?
3. Можно ли несколько раз перегрузить один и тот же оператор в одном классе?
4. Как применить перегруженный оператор для операций с объектами своего класса?
5. Чем отличается перегрузка операторов внутри класса и перегрузка оператора как глобальной функции?

ЛАБОРАТОРНАЯ РАБОТА № 7
Шаблоны классов

Цель: научиться создавать шаблоны классов, применять шаблоны как для встроенных типов данных, так и для пользовательских классов, понимать особенности применения пользовательских классов в шаблонах.

Теоретическая информация

Понятие шаблона

В языке C++ возможно и параметрическое программирование. Параметризованные компоненты обладают свойством адаптивности к конкретной ситуации, в которой такой компонент используется, что позволяет разрабатывать достаточно универсальные и в то же время высокоэффективные компоненты программ (в частности, объекты). Параметрическое программирование в языке C++ реализовано с помощью *шаблонов* (*template*). В C++ определено два вида шаблонов: *шаблоны-классы* и *шаблоны-функции*.

Шаблоны-классы могут использоваться различными способами, но наиболее очевидным является их использование в качестве адаптивных объектов памяти. *Шаблоны-функции* могут использоваться для определения параметризованных алгоритмов. Основное отличие шаблона-функции от шаблона-класса в том, что не нужно сообщать компилятору, к параметрам каких типов применяется функция, он сам может определить это по типам ее формальных параметров. Возможность параметрического программирования на языке C++ обеспечивается стандартной библиотекой шаблонов STL (Standard Template Library).

Шаблоны функций

Объявление шаблона функции начинается с заголовка, состоящего из ключевого слова `template`, за которым следует список параметров шаблона:

```
template <class X> X min (X a, X b) {    // Описание
                                         // шаблона функции
return a<b ? a : b;
}
```

Ключевое слово `class` в описании шаблона означает тип, идентификатор в списке параметров шаблона `X` означает имя любого типа.

В описании заголовка функции этот же идентификатор означает тип возвращаемого функцией значения и типы параметров функции:

```
int m = min (1, 2);       // Использование шаблона функции
```

Экземпляр шаблона функции породит компилятор

```
int min (int a, int b) { return a<b ? a : b; }
```

В списке параметров шаблона слово `class` может также относиться к обычному типу данных. Шаблоны иногда называют *параметризованными типами*. Приведем описание шаблона функции:

```
template <class T> T toPower (T base, int exponent) {
T result = base;
if (exponent==0) return (T)1;
if (exponent<0) return (T)0;
while (--exponent) result *= base;
return result;
}
```

Переменная `result` имеет тип `T`, так что, когда передаваемое в программу значение есть 1 или 0, оно сначала приводится к типу `T`, чтобы соответствовать объявлению шаблона функции. Типовой аргумент шаблона функции определяется согласно типам данных, используемых в вызове этой функции:

```
int i = toPower (10, 3);
long l = toPower (1000L, 4);
double d = toPower (1e5, 5);
```

В первом примере `T` становится типом `int`, во втором – `long`. Наконец, в третьем примере `T` становится типом `double`. Следующий пример приведет к ошибке компиляции, так как в нем используются разные типы данных:

```
int i = toPower (1000L, 4);
```

Требования к фактическим параметрам шаблона

Шаблон функции `toPower()` может быть использован практически для любого типа данных. Предостережение «практически» проистекает из характера операций, выполняемых над параметром `base` и переменной `result` в теле функции `toPower()`. Какой бы тип мы не использовали в функции `toPower()`, эти операции для нее должны быть определены.

Вот список действий, выполняемых в функции `toPower()` с переменными `base` и `result`:

```
1) T result = base;
2) return (T)1;
3) return (T)0;
4) result *= base;
5) return result;
```

Все эти действия определены для встроенных типов. Однако если вы создадите функцию `toPower()` для какого-либо классового типа, то в этом случае такой класс должен будет включать общедоступные принадлежащие функции, которые обеспечивают следующие возможности:

1) действие 1 инициализирует объект типа `T` таким образом, что класс `T` должен содержать конструктор копирования;

2) действия 2 и 3 преобразуют значения типа `int` в объект типа `T`, поэтому класс `T` должен содержать конструктор с параметром типа `int`, поскольку именно таким способом в классах реализуется преобразование к классовым типам;

3) действие 4 использует операцию `*=` над типом `T`, поэтому класс должен содержать собственную функцию-operator `*=()`;

4) действие 5 предполагает, что в типе `T` предусмотрена возможность построения безопасной копии возвращаемого объекта (см. конструктор копирования).

Схема такого класса выглядит следующим образом:

```
class T {
public:
T (const T &base);        // конструктор копирования
T (int i); //приведение int к T
operator *= (T base);     // ... прочие методы
} ;
```

Используя классы в шаблонах функций, убедитесь в том, что вы знаете, какие действия с ними выполняются в шаблоне функции, и определены ли для класса эти действия. Если вы не снабдили класс необходимыми функциями, возникнут различные невразумительные сообщения об ошибках.

Шаблоны классов

Вы можете создавать шаблоны и для классов, что позволяет столь же красиво работать с любыми типами данных. Классическим примером являются контейнерные классы, например множества. Используя шаблон, можно создавать параметрический класс для множеств, после чего порождать конкретные множества цветов, строк и т. д.

Давайте сначала рассмотрим вполне тривиальный пример, просто чтобы привыкнуть к используемому синтаксису. Рассматриваемый далее пример – класс, который хранит пару значений. Принадлежащие функции этого класса передают минимальное и максимальное значения, а также позволяют определить, являются ли два значения одинаковыми. Еще раз повторю, что раз перед нами шаблон класса, то тип значения может быть практически любым:

```
template <class T> class Pair {
        T a, b;
public:
        Pair (T t1, T t2);
        T Max();
        T Min ();
        int isEqual ();
};
```

Пока все выглядит также изящно, как и для шаблонов функций. Единственная разница состоит в том, что вместо описания функции используется объявление класса. Шаблоны классов становятся все более сложными, когда вы описываете принадлежащие функции класса. Вот, например, описание принадлежащей функции `Min()` класса `Pair`:

```
template <class T> T Pair <T>::Min() { return a < b ? a : b; }
```

Чтобы понять эту запись, давайте вернемся немного назад. Если бы `Pair` был обычным классом (а не шаблоном класса) и `T` был бы некоторым конкретным типом, то функция `Min` класса `Pair` была бы описана таким образом:

```
T Pair::Min() { return a < b ? a : b; }
```

Для случая шаблонной версии нам необходимо, во-первых, добавить заголовок шаблона `template <class T>`. Затем нужно дать имя классу. Помните, что на самом деле мы описываем множество классов – семейство `Pair`. Повторяя синтаксис префикса (заголовка) шаблона, экземпляр класса `Pair` для целых типов можно назвать `Pair<int>`, экземпляр для типа

double – Pair<double>, для типа Vector – Pair<Vector>. Однако в описании принадлежащей функции нам необходимо использовать имя класса Pair<T>. Это имеет смысл, так как заголовок говорит нам, что T означает имя любого типа.

Приведем текст остальных методов класса Pair. Описания методов помещаются в заголовочный файл, так как они должны быть видимы везде, где используется класс Pair.

```
// конструктор
template <class T> Pair <T>::Pair (T t1, T t2) : a(t1),
b(t2) {}
// метод Max
template <class T> T Pair <T>::Max() { return a>b ? a : b; }
// метод isEqual
template <class T> int Pair <T>::isEqual() {
       if (a==b) return 1;
       return 0;
}
```

Ранее уже отмечалось, что шаблоны функций могут работать только для тех (встроенных) типов данных или классов, которые поддерживают необходимые операции. То же самое справедливо и для шаблонов классов. Чтобы создать экземпляр класса Pair для некоторого типа, например для класса X, этот класс должен содержать следующие общедоступные функции:

```
X (X &); // конструктор копирования
int operator == (X);
int operator < (X);
```

Три указанные функции необходимы, так как они реализуют операции, выполняемые над объектами типа T в методах шаблона класса Pair.

Если вы собираетесь использовать некоторый шаблон класса, как узнать какие операции требуются? Если шаблон класса снабжен документацией, то эти требования должны быть в ней указаны. В противном случае придется читать первичную документацию – исходный текст шаблона.

Шаблоны классов: не только для типов

Параметризовать некоторый класс так, чтобы он работал для любого типа данных – это только половина того, что шаблоны обеспечивают для классов. Другой аспект состоит в том, чтобы дать возможность задания числовых параметров. Это позволяет Вам, например, создавать объекты типов «Вектор

из 20 целых», «Вектор из 1000 целых» и т. п. Основная идея проста, хотя используемый синтаксис может показаться сложным. Давайте в качестве примера рассмотрим некоторый обобщенный класс `Vector`. Как и класс `Pair`, класс `Vector` содержит функции `Min()`, `Max()`, `isEqual()`. Однако в нем может быть любое количество участников, а не два. В классе `Pair` число участников фиксировано и задаются они в качестве аргументов конструктора. В шаблоне `Vector` вместо этого используется второй параметр заголовка шаблона:

```
template <class T, int n> class Vector  {
       T *coord;
       int current;
public:
       Vector();
       ~Vector() {delete[] coord;}
       void newCoord (T x);
       T Max ();
       T Min();
       int isEqual();                    };
```

Значение `n`, заданное в заголовке шаблона не используется в описании класса, но применяется в описании его методов. Конструктор `Vector`, использующий значение `n` для задания размера массива, выглядит так:

```
// конструктор
template <class T, int n> Vector <T, n>::Vector() {
       coord = new T[n]; current = 0;
}
// метод Max
template <class T, int n> T Vector <T, n>::Max() {
       T result (coord[0]); // *
       for (int i=0; i<n; i++)
             if (result < coord[i]) // **
                   result = coord[i]; // ***
}
```

В конструкторе задается список инициализаций, устанавливающих начальные значения для двух элементов класса. Элемент `coord` инициализируется адресом динамически размещенного массива размером `n` и состоящего из элементов типа `T`, а элемент `current` инициализируется значением `0`.

Опять заметим, что в качестве `T` может выступать практически любой тип. Однако и в этом случае успешная реализация возможна лишь при определенных

условиях: для объектов, чей тип передается в шаблон в качестве параметра, должны быть определены следующие операции:

```
1) конструктор копирования (*),
2) оператор < (**), и > для метода Max(),
3) оператор = (***).
```

Имеется несколько вариантов использования шаблонов с параметрами-значениями для динамического размещения массивов различных размеров. Например, можно передать размер массива конструктору. Указание размеров объекта во время конструирования или путем обращения к некоторому методу действительно обеспечивает задание размера, однако такой способ не позволяет создать отдельный тип для каждого отдельного размера. Подход с использованием шаблона гарантирует, что размер становится частью типа.

Хотите ли вы, чтобы различные размеры были различными типами, зависит от ваших нужд. Если сравнение двух векторов с четырьмя и пятью координатами не имеет особого смысла, то было бы неплохо сделать их различными типами. Вместе с тем, в случае классов для матриц, вы, возможно, не захотите иметь особый тип для каждого размера матриц, так как, например, умножение, работает с матрицами различных размеров. Если вы столкнетесь с подобной проблемой, то вам потребуются только разумные проверки времени выполнения, а не контроль типов при компиляции.

Хотя числовые параметры шаблонов часто используются для задания размеров различных элементов, как это было показано для класса `Vector`, этим их применение не ограничивается. Например, с помощью числовых параметров можно задавать диапазоны значений графических координат в графическом классе.

Наследование в шаблонах классов

Шаблоны классов, как и классы, поддерживают механизм наследования. Все основные идеи наследования при этом остаются неизменными, что позволяет построить иерархическую структуру шаблонов, аналогичную иерархии классов. Рассмотрим совершенно тривиальный пример, на котором продемонстрируем, каким образом можно создать шаблон класса, производный из нашего шаблона класса `Pair`. Пусть это будет класс `Trio`, в котором к паре элементов `a` и `b` из `Pair` добавим еще один `c`:

```
template <class T> class Trio: public Pair <T> {
       T c;
```

```
        public:
        Trio (T t1, T t2, T t3);
        ...
};
template <class T>
        Trio<T>::Trio (T t1, T t2, T t3): Pair <T> (t1, t2),
c(t3) {}
        // Заметьте, что вызов родительского конструктора
        // также сопровождается передачей типа T
        // в качестве параметра
```

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 6.
2. Создать шаблон класса, в котором определить конструктор, в котором реализовано создание массива на заданное количество элементов `n` (количество передается в параметре конструктору).
3. В шаблоне определить метод добавления элемента массива и метод получения элемента массива по индексу.
4. В шаблоне определить функцию `T min()` и функцию `T max()`, которые возвращают минимальное и максимальное значения из объектов массива. Для работы этих функций потребуется перегрузка операторов сравнения в базовом классе. Сравнение объектов производить по любому из полей данных базового класса.
5. В основном теле программы применить шаблон для разных типов данных: `int`, `char`, указатель на базовый класс, каждый из классов-наследников.
6. Сделать выводы.

Пример упрощенного варианта реализации

```
#include "pch.h"
#include <iostream>
#include <string>

using namespace std;

template<class T>
class MyArray {
      T* arr;
      int count = 0;
      int size;
```

```
public:
	MyArray(int n) {
		arr = new T[n];
		size = n;
	}
	~MyArray() {
		delete [] arr;
	}
	void addItem(T obj) {
		if (count < size) {
			arr[count] = obj;
			count++;
		}
	}
	int findItem(T obj) {
		int index = -1;
		for (int i = 0; i < count; i++) {
			if (arr[i] == obj) {
				index = i;
				break;
			}
		}
		return index;
	}
};


class Article {
	string name;
	string author;
public:
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	string getAuthor() {
		return author;
	}
	void setAuthor(string author) {
		this->author = author;
	}
	Article() {
		name = "No name";
		author = "No author";
	}
	Article(string _name, string _author) {
		name = _name;
		author = _author;
	}
```

```
	~Article() {
	}
	virtual void print() = 0;
};

class Scientific : public Article {
	string science;
public:
	string getScience() {
		return science;
	}
	void setScience(string science) {
		this->science = science;
	}
	Scientific() {
		science = "No science";
	}
	Scientific(string _name, string _author, string science):
Article(_name, _author){
		this->science = science;
	}
	~Scientific() {
	}
	void print() {                    // метод вывода
		cout << "Scientific -=-=- Name: " << getName() << " Author: "
			<< getAuthor() << " Science: " << getScience() << endl;
	}
};

class News : public Article {
	string area;
public:
	string getArea() {
		return area;
	}
	void setArea(string area) {
		this->area = area;
	}
	News() {
		area = "No area";
	}
	News(string _name, string _author, string area) {
		this->setName(_name);
		this->setAuthor(_author);
		this->area = area;
	}
	~News() {
	}
	void print() {
		cout << "News -=-=- Name: " << getName() << " Author: " <<
		getAuthor() << " Area: " << getArea() << endl;
```

```
        }
};

int main()
{
        MyArray<int> intArr(3);
        intArr.addItem(12);
        intArr.addItem(22);
        intArr.addItem(31);
        cout << "Index of int: " << intArr.findItem(22) << endl;

        MyArray<char> charArr(3);
        charArr.addItem('a');
        charArr.addItem('b');
        charArr.addItem('c');
        cout << "Index of char: " << charArr.findItem('d') << endl;

        Scientific* sc1 = new Scientific();
        Scientific* sc2 = new Scientific("Biology article", "Petrov",
        "ecology");
        News* n1 = new News();
        News* n2 = new News("Top news", "Sidorov", "Politics");

        MyArray<Article*> myArr(4);
        myArr.addItem(sc1);
        myArr.addItem(sc2);
        myArr.addItem(n1);
        myArr.addItem(n2);
        cout << "Index of Article: " << myArr.findItem(n1) << endl;

        delete sc1;
        delete sc2;
        delete n1;
        delete n2;
}
```

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Приведите листинг программы.
3. Разместите скриншоты результатов выполнения программы.
4. Сделайте краткие пояснения к алгоритму работы.
5. Сформулируйте выводы.

Дополнительные вопросы

1. Как создать шаблон класса?
2. Можно ли применять один и тот же шаблон для разных типов данных?
3. Может ли быть несколько параметров-типов у шаблона класса?
4. Можно ли в качестве параметра-типа указать пользовательский класс?
5. Если в методах шаблона класса применяется, например, оператор <, то какие ограничения накладываются на параметр-тип?

ЛАБОРАТОРНАЯ РАБОТА № 8
Обработка исключений

Цель: научиться обрабатывать исключительные ситуации различных типов, устанавливать исключения, создавать классы исключений, выводить данные о возникшей ошибке.

Теоретическая информация

Исключения в C++

C++ обеспечивает встроенный механизм обработки ошибок, называемый *обработкой исключительных ситуаций*. Благодаря обработке исключительных ситуаций можно упростить управление и реакцию на ошибки времени исполнения. Обработка исключительных ситуаций в C++ строится с помощью трех ключевых слов: `try`, `catch`, `throw`. Операторы программы, во время выполнения которых вы хотите обеспечить обработку исключительных ситуаций, располагаются в блоке `try`. Если исключительная ситуация (т. е. ошибка) имеет место внутри блока `try`, она генерируется искусственно (с помощью `throw`). Перехватывается и обрабатывается исключительная ситуация с помощью ключевого слова `catch`. Любой оператор, который генерирует исключительную ситуацию, должен выполняться внутри блока `try`. Функции, которые вызываются внутри блока `try` также могут генерировать исключительную ситуацию. Любая исключительная ситуация должна перехватываться оператором `catch`, который следует непосредственно за блоком `try`, генерирующим исключительную ситуацию:

```
try { // блок try
}
catch (type1 arg1) {      // блок catch
}
catch (type2 arg2) {      // блок catch
}
...
catch (typeN argN) {      // блок catch
}
```

Блок `try` должен содержать ту часть программы, в которой вы хотите отслеживать ошибки. Это могут быть как несколько операторов внутри одной функции, так и все операторы функции `main()` (что, естественно, вызывает отслеживание ошибок во всей программе).

Когда исключительная ситуация возникает, она перехватывается соответствующим ей оператором `catch`, который ее обрабатывает. С блоком `try` может быть связано более одного оператора `catch`. То, какой конкретно оператор `catch` используется, зависит от типа исключительной ситуации, т. е., если тип данных, указанный в операторе `catch`, соответствует типу исключительной ситуации, то выполняется данный оператор `catch`, а все другие операторы блока `try` пропускаются. Если исключительная ситуация перехвачена, то аргумент `arg` получает ее значение. Можно перехватить любые типы данных, включая типы, создаваемые вами.

Оператор `throw` должен выполняться либо внутри блока `try`, либо в любой функции, которую этот блок вызывает. Здесь *исключительная_ситуация* – это исключительная ситуация, вызываемая оператором. Синтаксически выражение `throw` появляется в двух формах:

```
throw
throw выражение
```

Выражение `throw` устанавливает исключение. Самый внутренний блок `try`, в котором устанавливается исключение, используется для выбора оператора `catch`, обрабатывающего исключение. Выражение `throw` без аргумента повторно устанавливает текущее исключение. Обычно оно используется для дальнейшей обработки исключения вторым обработчиком, вызываемым из первого.

Рассмотрим пример работы исключительной ситуации:

```
void main(){
try{ throw 10; }
catch(int i) { cout << " error " << i << endl; }
return;
}
```

На экран выведется сообщение: `error 10`. Значение целого числа, выданное через `throw i`, хранится до завершения работы обработчика с целочисленной сигнатурой `catch(int)`. Это значение доступно для использования внутри обработчика в виде аргумента.

Пример переустановки исключения выглядит следующим образом:

```
catch(int n) { . . .
throw; // переустановка
}
```

Поскольку предполагается, что установленное выражение имело целый тип, переустановленное исключение также представляет собой постоянный целый объект, который обрабатывается ближайшим обработчиком, подходящим для этого типа. Концептуально установленное выражение передает информацию в обработчики. Часто обработчики не нуждаются в этой информации. Например, обработчик, который выводит сообщение и аварийно завершает работу, не нуждается ни в какой информации от окружения. Однако пользователь может захотеть выводить дополнительную информацию или использовать ее для принятия решения относительно действий обработчика. В таком случае допустимо формирование информации в виде объекта.

В C++ блоки `try` могут быть вложенными. Если в текущем блоке `try` нет соответствующего обработчика, выбирается обработчик из ближайшего внешнего блока `try`. Если он не обнаружен и там, тогда используется поведение по умолчанию. Блок `catch` выглядит подобно объявлению функции с одним параметром без возвращаемого типа:

```
catch (char *message) { cerr << message << endl; exit(1);
}
catch (...) {      // действие, которое нужно принять
                   // по умолчанию
cerr << " That's all folks" << andl; abort();
}
```

В списке аргументов допускается сигнатура, которая соответствует любому параметру. Кроме того, формальный параметр может быть абстрактным объявлением. Это значит, что он может иметь информацию о типе без имени переменной. Обработчик вызывается соответствующим выражением `throw`. При этом фактически происходит выход из блока `try`. Система вызывает функции освобождения, которые включают деструкторы для любых объектов, локальных для блока `try`.

Синтаксически спецификация исключения представляет собой части объявления функции и имеет форму:

```
заголовок_функции throw (список типов)
```

`Список типов` – это перечень типов, которые может иметь выражение `throw` внутри функции. Если этот список пуст, компилятор может предположить, что `throw` не будет выполняться функцией:

```
void foo() throw(int,over_flow);
void noex(int i) throw();
```

Если спецификация исключения оставлена, тогда возникает допущение, что такой функцией может быть установлено произвольное исключение. Хорошей практикой программирования будет показать с помощью спецификации, какие ожидаются исключения.

Пример кода, реализующего исключения при конструировании объекта:

```
vect::vect(int n) {
if(n <1) throw(n);
p=new int [n];
if(p==NULL) throw("NOT MEMORY");
}
void g(int m) {
try { vect a(m); }
catch (int n) {
cerr << "SIZE ERROR" << n << endl;
g(10);           // повторить g с допустимым размером
}
catch(const char *error) { cerr << error << endl; abort();
}
}
```

Обработчик заменил запрещенное значение на допустимое значение по умолчанию. Это может быть приемлемо на этапе отладки системы, когда большинство подпрограмм объединяются и проверяются. Система пытается продолжать обеспечивать дальнейшую диагностику. Это аналогично компилятору, пытающемуся продолжать анализировать неправильную программу после синтаксической ошибки. Часто компилятор предоставляет дополнительные сообщения об ошибках, которые оказываются полезными.

Вышеупомянутый конструктор проверяет только одну переменную на допустимое значение. Однако при такой форме записи разделение того, что является ошибкой и того, как она обрабатывается, очевидно. Это иллюстрирует ясную методологию разработки кода. Более обобщенно конструктор объекта может выглядеть так:

```
Object::Object(аргументы) {
if(недопустимый аргумент 1) throw выражение 1;
if(недопустимый аргумент 2) throw выражение 2;
. . . // попытка создания
}
```

Конструктор `Object` теперь обеспечивает набор выражений для установки запрещенного состояния. Теперь блок `try` может использовать информацию для восстановления или прерывания неправильного кода.

```
try { // оказоустойчивый код }
catch(объявление 1) { /* восстановление этого случая */ }
catch(объявление 2) { /* восстановление этого случая */ }
catch(объявление K) { /* восстановление этого случая */ }
// правильные или восстановленные переменные состояния
// теперь допустимы
```

Когда существует много определенных ошибочных условий, удобных для состояния данного объекта, может быть использована иерархия классов исключений. Эти иерархии позволяют соответственно упорядочному множеству `catch` обрабатывать исключения в логической последовательности. Тип базового класса в списке `catch` должен следовать после типа порожденного.

Философия восстановления после ошибок

Восстановление при возникновении ошибок в основном имеет отношение к правильности написания программы. Восстановление при возникновении ошибок основывается на передаче управления. Недисциплинированная передача управления ведет к хаосу. При восстановлении отказов предполагается, что исключение нарушило вычисления. Продолжать вычисления становится опасно. Обработка исключений влечет за собой упорядочное восстановление при появлении отказа. В большинстве случаев программирование, которое вызывает исключения, должно выводить диагностическое сообщение и элегантно завершать работу. При специальных формах обработки, типа работы в режиме реального времени и при отказоустойчивом вычислении существует необходимость в том, чтобы система не приостанавливалась. C++ использует модель завершения, которая вынуждает завершать текущий блок `try`. При этом режиме пользователь или повторяет код, игнорируя исключение, или подставляет результат по умолчанию и продолжает. При повторении кода более вероятно получить правильный результат. Опыт показывает, что обычно код слабо комментируется. Хорошо спланированный набор ошибочных условий, обнаруживаемых пользователями АТД, – важная часть проекта. Если при нормальном программировании слишком часто обнаруживаются ошибки и происходит прерывание – это признак того, что программа плохо продумана.

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 7.

2. Написать программу, в которой перехватываются исключения типа `int`, `string`. Сгенерировать исключительные ситуации.

3. Добавить к программе перехват любой исключительной ситуации `catch(...)`.

4. Добавить к программе перехват 2-3 исключительных ситуаций стандартных типов (`std::invalid_argument`, `std::length_error`, `std::out_of_range` или др.).

5. Создайть два собственных класса ошибки, наследуемых от стандартного. Добавить к программе перехват исключительных ситуаций созданных типов.

6. Программа должна демонстрировать обработку исключительных ситуаций на верхнем уровне (функция `main`), возникающих при вложенных вызовах методов объектов.

7. Программа должна демонстрировать локальную обработку исключительных ситуаций без передачи ее обработчику более высокого уровня.

8. Сделать выводы.

Пример упрощенного варианта реализации:

```
#include "pch.h"
#include <iostream>
#include <string>

using namespace std;

class MyException : public invalid_argument {    //собственный класс
                                                  // исключений, наследуемый
                                                // от исключения invalid_argument
public:
      MyException(string str) : invalid_argument(str) {}
};

template<class T>
class MyArray {
      T* arr;
      int count = 0;
      int size;
public:
      MyArray(int n) {
            arr = new T[n];
            size = n;
      }
      ~MyArray() {
            delete [] arr;
      }
```

```
	void addItem(T obj) {
		try {                    // обработка исключения внутри метода
			if (obj == NULL) {
				throw MyException("Argument is null");
			}
			if (count < size) {
				arr[count] = obj;
				count++;
			}
		}
		catch (MyException& my) {
			cout << "Error: " << my.what() << endl;
		}
	}
	T getItem(int i) {
		if (i >= size) {
			throw exception();
		}
		return arr[i];
	}
	int findItem(T obj) {
		int index = -1;
		for (int i = 0; i < count; i++) {
			if (arr[i] == obj) {
				index = i;
				break;
			}
		}
		return index;
	}
};


class Article {
	string name;
	string author;
public:
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
	}
	string getAuthor() {
		return author;
	}
	void setAuthor(string author) {
		this->author = author;
	}
	Article() {
		name = "No name";
```

```
            author = "No author";
        }
        Article(string _name, string _author) {
            name = _name;
            author = _author;
        }
        ~Article() {
        }
        virtual void print() = 0;
};

class Scientific : public Article {
        string science;
public:
        string getScience() {
            return science;
        }
        void setScience(string science) {
            this->science = science;
        }
        Scientific() {
            science = "No science";
        }
        Scientific(string _name, string _author, string science): Arti-
cle(_name, _author){
            this->science = science;
        }
        ~Scientific() {
        }
        void print() {                  // метод вывода
            cout << "Scientific -==-= Name: " << getName() << " Author: "
                << getAuthor() << " Science: " << getScience() << endl;
        }
};

class News : public Article {
        string area;
public:
        string getArea() {
            return area;
        }
        void setArea(string area) {
            this->area = area;
        }
        News() {
            area = "No area";
        }
        News(string _name, string _author, string area) {
            this->setName(_name);
            this->setAuthor(_author);
```

```
            this->area = area;
      }
      ~News() {
      }
      void print() {
            cout << "News -=-=- Name: " << getName() << " Author: " <<
            getAuthor() << " Area: " << getArea() << endl;
      }
};

int main()
{
      try {
            MyArray<int> intArr(3);
            intArr.addItem(12);
            intArr.addItem(22);
            cout << "Enter 0 for zero exception" << endl;
            char zero;
            cin >> zero;
            int izero = atoi(&zero);     // возможна ошибка конвертации
                                         // в число
            if (izero == 0) {
                  throw string("Error: Devide by zero or incorrect char");
                                    // генерация ошибки при делении на 0
            }
            else intArr.addItem(30 / izero);

            cout << "Enter count more then 3 for exception" << endl;
            int count;
            cin >> count;
            for (int i = 3; i < count; i++) {
                  if (count > 3) {
                        throw count; // генерация ошибки при попытке
                                     // выйти за границы массива
                  }
                  intArr.addItem(10);       // возможна ошибка
                                            // переполнения массива
            }
            cout << "Index of int: " << intArr.findItem(22) << endl;

            cout << "Enter index more then 3 for exception" << endl;
            int index;
            cin >> index;
            cout << "Item: " << intArr.getItem(index) << endl;
                        // возможна ошибка
                        //обращения к несуществующему элементу массива
      }
      catch (string str) {
            cout << str << endl;
```

```
}
catch (int i) {
	cout << "Error: Count is very large, count = " << i << endl;
}
catch (...) {
	cout << "Other ecxeption " << endl;
};

try {
	MyArray<char> charArr(3);
	charArr.addItem('a');
	charArr.addItem('b');

	cout << "Enter incorrect symbol for exception
	(correct is a-z)" << endl;
	char sym;
	cin >> sym;
	if (sym < 'a' || sym>'z') {
		throw invalid_argument("Incorrect symbol
		(correct is a - z)");
	}
	charArr.addItem(sym);

	cout << "Index of char: " << charArr.findItem('d') << endl;
}
catch (invalid_argument err) {
	cout << "Error: " << err.what() << endl;
}

try{
	Scientific* sc1 = new Scientific();
	Scientific* sc2 = new Scientific("Biology article",
	"Petrov", "ecology");
	News* n1 = NULL;
	News* n2 = new News("Top news", "Sidorov", "Politics");

	MyArray<Article*> myArr(4);
	myArr.addItem(sc1);
	myArr.addItem(sc2);
	myArr.addItem(n1);	// ошибка добавления null,
				// обрабатывается в методе addItem
	myArr.addItem(n2);
	cout << "Index of Article: " << myArr.findItem(n2) << endl;

	delete sc1;
	delete sc2;
	delete n2;
}
catch(...) {
```

```
            cout << "Error..." << endl;
        }
}
```

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Приведите листинг программы.
3. Разместите скриншоты результатов выполнения программы.
4. Сделайте краткие пояснения к алгоритму работы.
5. Сформулируйте выводы.

Дополнительные вопросы

1. Что такое исключительная ситуация?
2. Что произойдет, если поставить `catch(...)` первым в списке обработчиков?
3. Могут ли быть несколько обработчиков `catch` для одного блока `try`?
4. Как получить информацию о возникшей ошибке?
5. Какие классы исключений вы знаете?

ЛАБОРАТОРНАЯ РАБОТА № 9
Потоки ввода-вывода

Цель: научиться работать с потоками ввода-вывода информации, работать с файлами.

Теоретическая информация

Потоки для работы с текстовыми файлами представляют собой объекты, для которых не задан режим открытия `ios::binary`.

Запись в файл

Для записи в файл к объекту `ofstream` или `fstream` применяется оператор << (как и при выводе на консоль):

```
#include <iostream>
#include <fstream>

int main()
{
	std::ofstream out;          // поток для записи
	out.open("D:\\hello.txt");  // окрываем файл для записи
	if (out.is_open())
	{
		out << "Hello World!" << std::endl;
	}

	std::cout << "End of program" << std::endl;
	return 0;
}
```

Данный способ перезаписывает файл заново. Если надо дозаписать текст в конец файла, то для его открытия нужно использовать режим `ios::app`:

```
std::ofstream out("D:\\hello.txt", std::ios::app);
if (out.is_open())
{
	out << "Welcome to CPP" << std::endl;
}
out.close();
```

Чтение из файла

Если надо считать всю строку целиком или даже все строки из файла, то лучше использовать встроенную функцию `getline()`, которая принимает поток для чтения и переменную, в которую надо считать текст:

```
#include <iostream>
#include <fstream>
#include <string>

int main()
{
std::string line;

std::ifstream in("D:\\hello.txt"); // окрываем файл
                                   // для чтения
if (in.is_open())
{
     while (getline(in, line))
     {
          std::cout << line << std::endl;
     }
     }
     in.close();                   // закрываем файл

     std::cout << "End of program" << std::endl;
     return 0;
}
```

Также для чтения данных из файла для объектов `ifstream` и `fstream` может применяться оператор >> (также как и при чтении с консоли):

```
#include <iostream>
#include <fstream>
#include <vector>

struct Operation
{
     int sum;              // купленная сумма
     double rate;          // по какому курсу
     Operation(double s, double r) : sum(s), rate(r)
     {}
};
int main()
{
     std::vector<Operation> operations = {
          Operation(120, 57.7),
          Operation(1030, 57.4),
          Operation(980, 58.5),
          Operation(560, 57.2)
     };
```

```
    std::ofstream out("D:\\operations.txt");

    if (out.is_open())
    {
        for (int i = 0; i < operations.size(); i++)
        {
            out << operations[i].sum << " " << opera-
tions[i].rate << std::endl;
        }
    }
    out.close();

    std::vector<Operation> new_operations;
    double rate;
    int sum;
    std::ifstream in("D:\\operations.txt");   // окрываем
                                              // файл для чтения
    if (in.is_open())
    {
        while (in >> sum >> rate)
        {
            new_operations.push_back(Operation(sum, rate));
        }
    }
    in.close();

    for (int i = 0; i < new_operations.size(); i++)
    {
        std::cout << new_operations[i].sum << " - "
        << new_operations[i].rate << std::endl;
    }
    return 0;
}
```

Здесь вектор структур `Operation` записывается в файл:

```
for (int i = 0; i < operations.size(); i++)
{
    out << operations[i].sum << " " << operations[i].rate
    << std::endl;
}
```

При записи в данном случае будет создаваться файл в формате

```
120 57.7
1030 57.4
980 58.5
560 57.2
```

Используя оператор >>, можно последовательно считать данные в переменные `sum` и `rate` и ими инициализировать структуру:

```
while (in >> sum >> rate)
{
     new_operations.push_back(Operation(sum, rate));
}
```

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 8.
2. Определить глобальный оператор << для вывода данных базового класса на экран.
3. Определить глобальный оператор >> для вывода данных в объект базового класса на экран.
4. Организовать сохранение данных объекта основного класса в файл.
5. Организовать чтение данных из файла и занесение их в объект основного класса.
6. Сделать выводы.

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Приведите листинг программы.
3. Разместите скриншоты результатов выполнения программы.
4. Сделайте краткие пояснения к алгоритму работы.
5. Сформулируйте выводы.

Дополнительные вопросы

1. Какие классы потоков вы знаете?
2. Что требуется для организации чтения и записи в файл?
3. Как можно отформатировать вывод данных в поток?
4. Как можно объект вывести в поток или сохранить в файл?
5. Как можно связать поток с конкретным файлом?

ЛАБОРАТОРНАЯ РАБОТА № 10
Контейнеры

Цель: научиться применять контейнеры различных видов, методы для работы с содержимым контейнеров и контейнеры для хранения объектов пользовательских классов.

Теоретическая информация

Стандартная библиотека шаблонов (Standard Template Library, STL) входит в стандартную библиотеку языка C++. В неё включены реализации наиболее часто используемых контейнеров и алгоритмов, что избавляет программистов от рутинного их переписывания. Разработчики STL поставили перед собой и реализовали сверхзадачу: сделать библиотеку одновременно эффективной и универсальной. STL строится на основе шаблонов классов, поэтому входящие в неё алгоритмы и структуры применимы почти ко всем типам данных.

Состав STL

Ядро библиотеки образуют три элемента: *контейнеры*, *алгоритмы* и *итераторы*. *Контейнеры* (`containers`) – это объекты, предназначенные для хранения других элементов, например: `vector`, `list`, `set`. *Ассоциативные контейнеры* (`associative containers`) позволяют с помощью ключей получить быстрый доступ к хранящимся в них значениям. В каждом классе-контейнере определен набор функций для работы с ними. Например, список содержит функции для вставки, удаления и слияния элементов. *Алгоритмы* (`algorithms`) выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнеров. Многие алгоритмы предназначены для работы с последовательностью (`sequence`), которая представляет собой линейный список элементов внутри контейнера. *Итераторы* (`iterators`) – это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера. С итераторами можно работать так же, как с указателями. К ним можно применить операции *, инкремента, декремента. Типом итератора объявляется тип `iterator`, который определен в различных контейнерах. В STL также поддерживаются *обратные итераторы* (`reverse iterators`). *Обратными итераторами*

могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, но проходящие последовательность в обратном направлении.

Вдобавок к контейнерам, алгоритмам и итераторам в STL поддерживается ещё несколько стандартных компонентов. Главными среди них являются *распределители памяти*, *предикаты* и *функции сравнения*. У каждого контейнера имеется определенный для него распределитель памяти (`allocator`), который управляет процессом выделения памяти для контейнера. По умолчанию распределителем памяти является объект класса `allocator`. Можно определить собственный распределитель. В некоторых алгоритмах и контейнерах используется функция особого типа, называемая *предикатом*. Предикат может быть *унарным* и *бинарным*. Возвращаемое значение – истина либо ложь. Точные условия получения того или иного значения определяются программистом. Тип унарных предикатов – `UnPred`, бинарных – `BinPred`. Тип аргументов соответствует типу хранящихся в контейнере объектов. Определен специальный тип бинарного предиката для сравнения двух элементов. Он называется *функцией сравнения* (`comparison function`). Функция возвращает истину, если первый элемент меньше второго. Тип функции – `Comp`.

Особую роль в STL играют объекты-функции. *Объекты-функции* – это экземпляры класса, в котором определена операция «круглые скобки» (). В ряде случаев удобно заменить функцию на объект-функцию. Когда объект-функция используется в качестве функции, то для ее вызова используется `operator ()`.

Пример:

```
class less{
public:
    bool operator()(int x, int y){
       return x<y;
    }
};
```

Классы-контейнеры

В STL определены два типа контейнеров: последовательности и ассоциативные. Ключевая идея для стандартных контейнеров заключается в том, что когда это представляется разумным, они должны быть логически взаимозаменяемыми. Пользователь может выбирать между ними, основываясь

на соображениях эффективности и потребности в специализированных операциях. Например, если часто требуется поиск по ключу, можно воспользоваться `map` (ассоциативным массивом). Однако если преобладают операции, характерные для списков, можно воспользоваться контейнером `list`. Если добавление и удаление элементов часто производится в концы контейнера, следует подумать об использовании очереди `queue`, очереди с двумя концами `deque`, стека `stack`. По умолчанию пользователь должен использовать `vector`; он реализован, чтобы хорошо работать для самого широкого диапазона задач.

Идея обращения с различными видами контейнеров и в общем случае со всеми видами источников информации унифицированным способом ведет к понятию обобщенного программирования. Для поддержки этой идеи STL содержит множество обобщенных алгоритмов. Такие алгоритмы избавляют программиста от необходимости знать подробности отдельных контейнеров. В STL определены следующие классы-контейнеры (в угловых скобках указаны заголовочные файлы, где определены эти классы):

`bitset`	множество битов <bitset.h>
`vector`	динамический массив <vector.h>
`list`	линейный список <list.h>
`deque`	двусторонняя очередь <deque.h>
`stack`	стек <stack.h>
`queue`	очередь <queue.h>
`priority_queue`	очередь с приоритетом <queue.h>
`map`	ассоциативный список для хранения пар ключ/значение <map.h>
`multimap`	с каждым ключом связано два или более значений <map.h>
`set`	множество <set.h>
`multiset`	множество, в котором элемент не обязательно уникален <set.h>

Итераторы: `begin(),end(),rbegin(),rend()`.

Доступ к элементам: `front(),back(),operator[](i),at(i)`.

Включение элементов: `insert(p,x), insert(p,n,x), insert(p,first,last),push_back(x),push_front(x)`.

Удаление элементов: `pop_back(), pop_front(), erase(p), erase(first,last), clear()`.

Другие операции: `size(), empty(), capacity(), reserve(n), resize(n), swap(x), ==, !=, <`.

Операции присваивания: operator=(x), assign(n,x), assign(first,last).

Ассоциативные операции: operator[](k), find(k), lower_bound(k), upper_bound(k), equal_range(k).

Контейнер vector-вектор

Вектор vector в STL определен как динамический массив с доступом к его элементам по индексу:

```
template<class T,class Allocator=allocator<T>>class
std::vector{…};
```

где T – тип предназначенных для хранения данных;

Allocator задает распределитель памяти, который по умолчанию является стандартным.

Пример:

```
vector<int> a; vector<double> x(5); vector<char> c(5,'*');
vector<int> b(a);
```

Для любого объекта, который будет храниться в векторе, должен быть определен конструктор по умолчанию. Кроме того, для объекта должны быть определены операторы < и ==.

Для класса vector определены следующие операторы сравнения: ==, <, <=, !=, >, >=. Кроме этого, для класса vector определяется оператор индекса []. Новые элементы могут включаться с помощью функций insert(), push_back(), resize(), assign(). Существующие элементы могут удаляться с помощью функций erase(), pop_back(), resize(),clear(). Доступ к элементам осуществляется с помощью итераторов begin(), end(), rbegin(), rend(). Манипулирование контейнером, сортировка, поиск в нем и тому подобное возможно с помощью глобальных функций <algorithm.h>.

Пример:

```
#include<iostream.h>
#include<vector.h>
using namespace std;
void main(){
    vector<int> v; int i;
```

```
    for(i=0;i<10;i++) v.push_back(i);
    for(i=0;i<10;i++)v[i]=v[i]+v[i];
    for(i=0;i<v.size();i++) cout<<v[i]<<" "; cout<<endl; }
```

Пример доступа к вектору через итератор:

```
#include<iostream.h>
#include<vector.h>
using namespace std;
void main(){
    vector<int> v; int i;
    for(i=0;i<10;i++)v.push_back(i);
cout<<"size="<<v.size()<<"\n";
    vector<int>::iterator p=v.begin();
    while(p!=v.end()) {cout<<*p<<" ";p++;}
}
```

Пример. Вектор содержит объекты пользовательского класса:

```
#include<iostream.h>
#include<vector.h>
#include"student.h"
using namespace std;
void main() {
    vector<STUDENT> v(3); int i;
    v[0]=STUDENT("Иванов",45.9);
    v[1]=STUDENT("Петров",30.4);
    v[0]=STUDENT("Сидоров",55.6);
    for(i=0;i<3;i++) cout<<v[i]<<" "; cout<<endl; // вывод
}
```

Ассоциативные контейнеры (массивы)

Ассоциативный массив содержит пары значений. Зная одно значение, называемое *ключом* (`key`), мы можем получить доступ к другому, называемому *отображенным значением* (`mapped value`). Ассоциативный массив можно представить как массив, для которого индекс не обязательно должен иметь целочисленный тип: `V& operator[](const K&)` возвращает ссылку на `V`, соответствующий `K`.

Ассоциативные контейнеры – это обобщение понятия ассоциативного массива. Ассоциативный контейнер `map` – это последовательность пар (ключ, значение), которая обеспечивает быстрое получение значения по ключу. Контейнер `map` предоставляет двунаправленные итераторы. Ассоциативный контейнер `map` требует, чтобы для типов ключа существовала операция «<».

Он хранит свои элементы отсортированными по ключу так, что перебор происходит по порядку.

Спецификация шаблона для класса `map`:

```
template<class Key,classT,class Comp=less<Key>,
class Allocator=allocator<pair> > class std::map
```

Определена операция присваивания:

```
map& operator=(const map&);
```

Определены следующие операции: ==, <, <=, !=, >, >=.

В `map` хранятся пары ключ/значение в виде объектов типа `pair`.

Создавать пары ключ/значение можно не только с помощью конструкторов класса `pair`, но и с помощью функции `make_pair`, которая создает объекты типа `pair`, используя типы данных в качестве параметров.

Типичная операция для ассоциативного контейнера – это ассоциативный поиск при помощи операции индексации (`[]`):

```
mapped_type& operator[](const key_type& K);
```

Множества `set` можно рассматривать как ассоциативные массивы, в которых значения не играют роли, так что мы отслеживаем только ключи:

```
template<classT,class Cmp=less<T>,
class Allocator=allocator<T>>class std::set{…};
```

Множество, как и ассоциативный массив, требует, чтобы для типа `T` существовала операция «меньше» (<). Оно хранит свои элементы отсортированными, так что перебор происходит по порядку.

Контейнерные классы

Контейнерные классы используются тогда, когда необходимо накапливать и хранить большое количество однородных индивидуальных элементов. Если вы не знаете, сколько объектов вам будет необходимо для решения проблемы или как долго они будут существовать, то вы также не знаете, как хранить эти объекты. Вы не можете знать, сколько пространства необходимо для этих объектов, так как эта информация не известна, пока не настанет время выполнения. Решение большинства проблем в объектно-ориентированном дизайне выглядит следующим образом: вы создаете новый тип объектов, который решает эту обычную проблему хранения ссылок на другие

объекты. Конечно, вы можете сделать то же самое с помощью массива, который поддерживают многие языки. Но это гораздо большее. Этот новый объект, обычно называемый контейнером, будет расширять себя при необходимости, чтобы аккомодировать все, что вы поместите внутрь него. Так что вы не должны знать, сколько объектов вы положили храниться в контейнер. Просто создайте объект контейнера и дайте ему возможность позаботится о деталях. К счастью, хорошие языки объектно-ориентированного программирования пришли с набором готовых контейнеров.

Контейнеры – это эффективная альтернатива традиционным массивам и указателям языка C++, обеспечивающая полностью автоматическое распределение оперативной памяти. Преимущество автоматического распределения памяти по сравнению с ручным заключается в упрощении программирования и сокращении числа ошибок, а также в предсказуемости накладных расходов: не более 50% по времени выполнения программы и не более 100% по объему оперативной памяти, запрашиваемой у операционной системы. При ручном распределении памяти накладные расходы зависят от квалификации программиста, а также наличия времени для подбора оптимального метода распределения памяти. Обычно для контейнеров реализовано корректное их копирование, создание, удаление и другие операции.

Итераторы

Итераторы – удобная обертка для указателя. Кстати говоря, обычный указатель тоже можно считать итератором, правда, очень примитивным. Итераторы обладают массой достоинств, например таких, как автоматическое отслеживание размера типа, на который указывает итератор, или автоматизированные операции инкремента и декремента для перехода от элемента к элементу. Существует пять типов итераторов:

1. *Итераторы ввода* (`input_iterator`) поддерживают операции равенства, разыменования и инкремента: ==, !=, *i, ++i, i++, *i++.
2. *Итераторы вывода* (`output_iterator`) поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента: ++i, i++, *i=t, *i++=t.
3. *Однонаправленные итераторы* (`forward_iterator`) поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без ограничения применять присваивание: ==, !=, =, *i, ++i, i++, *i++.

4. *Двунаправленные итераторы* (`biderectional_iterator`) обладают всеми свойствами `forward`-итераторов, а также имеют дополнительную операцию декремента (--i, i--, *i--), что позволяет им проходить контейнер в обоих направлениях.

5. *Итераторы произвольного доступа* (`random_access_iterator`) обладают всеми свойствами `biderectional`-итераторов, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ по индексу: i+=n, i+n, i-=n, i-n, i1-i2, i[n], i1<i2, i1<=i2, i1>i2, i1>=i2.

Как это работает

Хотя внутреннее устройство контейнеров очень сильно различается, каждый контейнер обязан предоставить строго определённый интерфейс, через который с ним будут взаимодействовать алгоритмы. Этот интерфейс обеспечивают итераторы. Важно подчеркнуть, что никакие дополнительные функции-члены для взаимодействия алгоритмов и контейнеров не используются. Это сделано потому, что стандартные алгоритмы должны работать в том числе со встроенными контейнерами языка C++, у которых есть только итераторы (указатели). Таким образом, при написании собственного контейнера реализация итератора – необходимый минимум.

Можно выделить набор функций, которые реализует практически каждый контейнер. Они не требуются для взаимодействия с алгоритмами, но их реализация улучшает взаимозаменяемость контейнеров в прграмме:

`begin, end`	Возвращают итераторы начала и конца прямой последовательности
`rbegin, rend`	Возвращают итераторы начала и конца обратной последовательности
`front, back`	Возвращают ссылки на первый и последний элемент, хранящийся в контейнере
`push_back, pop_back`	Позволяют добавить или удалить последний элемент в последовательности
`push_front, pop_front`	Позволяют добавить или удалить первый элемент в последовательности
`size`	Возвращает количество элементов в контейнере
`empty`	Проверяет, есть ли в контейнере элементы
`clear`	Удаляет из контейнера все элементы
`insert, erase`	Позволяют вставить или удалить элемент(ы) в середине последовательности

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 3.
2. Создать объект-контейнер в соответствии с вариантом задания и заполнить его данными, тип которых определяется вариантом задания.
3. Просмотреть контейнер.
4. Изменить контейнер, удалив из него одни элементы и изменив другие.
5. Просмотреть контейнер, используя для доступа к его элементам итераторы.
6. То же самое повторить для данных пользовательского типа.
7. Сделать выводы.

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Приведите листинг программы.
3. Разместите скриншоты результатов выполнения программы.
4. Сделайте краткие пояснения к алгоритму работы.
5. Сформулируйте выводы.

Дополнительные вопросы

1. Какие виды контейнеров существуют?
2. В чем особенность ассоциативных контейнеров?
3. Что такое итератор?
4. Как организовать цикл для перебора всех элементов контейнера?
5. Как добавить или удалить элемент контейнера?

ЛАБОРАТОРНАЯ РАБОТА № 11
Алгоритмы

Цель: научиться применять алгоритмы различных видов для работы над контейнерами.

Теоретическая информация

Каждый алгоритм выражается шаблоном функции или набором шаблонов функций. Таким образом, алгоритм может работать с очень разными контейнерами, содержащими значения разнообразных типов. Алгоритмы, которые возвращают итератор, как правило, для сообщения о неудаче используют конец входной последовательности. Алгоритмы не выполняют проверки диапазона на их входе и выходе. Когда алгоритм возвращает итератор, это будет итератор того же типа, что и был на входе. Алгоритмы в STL реализуют большинство распространенных универсальных операций с контейнерами, таких как просмотр, сортировка, поиск, вставка и удаление элементов.

Алгоритмы определены в `<algorithm.h>`. Ниже приведены имена некоторых наиболее часто используемых функций-алгоритмов STL.

1. Немодифицирующие операции:

`for_earch()`	выполняет операции для каждого элемента последовательности
`find()`	находит первое вхождение значения в последовательность
`find_if()`	находит первое соответствие предикату в последовательности
`count()`	подсчитывает количество вхождений значения в последовательность
`count_if()`	подсчитывает количество выполнений предиката в последовательности
`search()`	находит первое вхождение последовательности как подпоследовательности
`search_n()`	находит `n`-е вхождение значения в последовательность

2. Модифицирующие операции:

`copy()`	копирует последовательность, начиная с первого элемента
`swap()`	меняет местами два элемента
`replace()`	заменяет элементы с указанным значением
`replace_if()`	заменяет элементы при выполнении предиката
`replace_copy()`	копирует последовательность, заменяя элементы с указанным значением
`replace_copy_if()`	копирует последовательность, заменяя элементы при выполнении предиката
`fill()`	заменяет все элементы данным значением

`remove()`	удаляет элементы с данным значением
`remove_if()`	удаляет элементы при выполнении предиката
`remove_copy()`	копирует последовательность, удаляя элементы с указанным значением
`remove_copy_if()`	копирует последовательность, удаляя элементы при выполнении предиката
`reverse()`	меняет порядок следования элементов на обратный
`random_shuffle()`	перемещает элементы согласно случайному равномерному распределению («тасует» последовательность)
`transform()`	выполняет заданную операцию над каждым элементом последовательности
`unique()`	удаляет равные соседние элементы
`unique_copy()`	копирует последовательность, удаляя равные соседние элементы

3. Сортировка:

`sort()`	сортирует последовательность с хорошей средней эффективностью
`partial_sort()`	сортирует часть последовательности
`stable_sort()`	сортирует последовательность, сохраняя порядок следования равных элементов
`lower_bound()`	находит первое вхождение значения в отсортированной последовательности
`upper_bound()`	находит первый элемент, больший чем заданное значение
`binary_search()`	определяет, есть ли данный элемент в отсортированной последовательности
`merge()`	сливает две отсортированные последовательности

4. Работа с множествами:

`includes()`	проверка на вхождение
`set_union()`	объединение множеств
`set_intersection()`	пересечение множеств
`set_difference()`	разность множеств

5. Минимумы и максимумы:

`min()`	меньшее из двух
`max()`	большее из двух
`min_element()`	наименьшее значение в последовательности
`max_element()`	наибольшее значение в последовательности

6. Перестановки:

`next_permutation()`	следующая перестановка в лексикографическом порядке
`pred_permutation()`	предыдущая перестановка в лексикографическом порядке

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 3.
2. Создать контейнер, содержащий объекты пользовательского типа. Тип контейнера выбирается в соответствии с вариантом задания.
3. Отсортировать его по убыванию элементов.
4. Просмотреть контейнер.
5. Используя подходящий алгоритм, найти в контейнере элемент, удовлетворяющий заданному условию.
6. Переместить элементы, удовлетворяющие заданному условию в другой (предварительно пустой) контейнер. Тип второго контейнера определяется вариантом задания.
7. Просмотреть второй контейнер.
8. Отсортировать первый и второй контейнеры по возрастанию элементов.
9. Просмотреть их.
10. Сделать выводы.

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Приведите листинг программы.
3. Разместите скриншоты результатов выполнения программы.
4. Сделайте краткие пояснения к алгоритму работы.
5. Сформулируйте выводы.

Дополнительные вопросы

1. Какие алгоритмы для работы над контейнерами существуют?
2. Как организовать сортировку контейнера, начиная не с первого элемента?
3. Как организовать сортировку контейнера по убыванию?

4. Как организовать сортировку контейнера, содержащего объекты пользовательского класса?

5. Как организовать поиск элемента, удовлетворяющего какому-либо требованию?

ЛАБОРАТОРНАЯ РАБОТА № 12
Лямбда-функции

Цель: научиться создавать лямбда-функции в классе, применять лямбда-функции для решения практических задач.

Теоретическая информация

Лямбда-функции – это то же, что функциональные_объекты_классов, но имеющие свой собственный синтаксис сущности.

К лямбда-функции можно относиться как к обычной функции, но из-за странного, на первый взгляд, синтаксиса, исходные коды с ними могут быть трудночитаемыми. Состоит лямбда-функция из двух основных частей: квадратных скобок и фигурных скобок:

```
C++
          // Листинг #1   Лямбда-функция C++11
     mingw
          int main(){
               []{};
          }
```

Такое написание – аналог объявления пустой обычной функции, только в нашем случае функция получается анонимной, потому что имя задавать лямбда-функции не нужно. Они обычно используются, как правило, везде для объявления анонимных функций.

Поскольку лямбда-функции могут заменять функции, то давайте произведём вызов.

```
C++
// Листинг #2  Использование лямбда-функции        C++11
     mingw
#include <iostream>

using namespace std;

int main() {
     []{ cout << "lambda-func\n"; }; //Это объявление
```

```
    []{ cout << "lambda-func\n"; }(); //А вызов делается
с помощью круглых скобок в конце
}
```

В круглые скобки вызова можно отдавать значения аргументов, однако где-то должны быть описаны типы принимаемых лямбда-функцией значений, и эти типы описываются в круглых скобках слева от фигурных:

```
C++
// Листинг #3  Лямбда-функции        Параметры и аргументы
     C++11     mingw
#include <iostream>

using namespace std;

int main() {
     [](const int x){ cout << "x == " << x; }(10); // Отдаём
                                                   // аргументы
     cout << "\n\n";
     [](const int x, const double d){ cout << "x == " << x
<< "\nd == " << d; }(10, 20.3); // Принимаем в параметры
                      // лямбда-функции переданные значения
}
```

Квадратные скобки представляют собой своеобразную ловушку для захвата внешних по отношению к лямбда-функции переменных. Можно отлавливать всё, что находится в той же области видимости, где расположилась сама лямбда-функция. Есть несколько вариантов отлова: по значению, по ссылке, захват указателя `this` по значению, можно захватывать или отдельные переменные, или всё, что есть в зоне видимости.

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 11.
2. Для алгоритмов, используемых в лабораторной работе № 11 использовать не глобальные функции, а лямбда-функции.
3. Сделать выводы.

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Приведите листинг программы.
3. Разместите скриншоты результатов выполнения программы.
4. Сделайте краткие пояснения к алгоритму работы.
5. Сформулируйте выводы.

Дополнительные вопросы

1. Что такое лямбда-функция?
2. Где может применяться лямбда-функция?
3. Как связана лямбда-функция и оператор `()`?
4. Является ли на самом деле лямбда-функция функцией?
5. Должна ли лямбда-функция иметь имя?

ЛАБОРАТОРНАЯ РАБОТА № 13
Поведенческие паттерны

Цель: научиться применять паттерн проектирования *Стратегия* для реализации различного поведения объектов, реализовывать динамическую смену поведения.

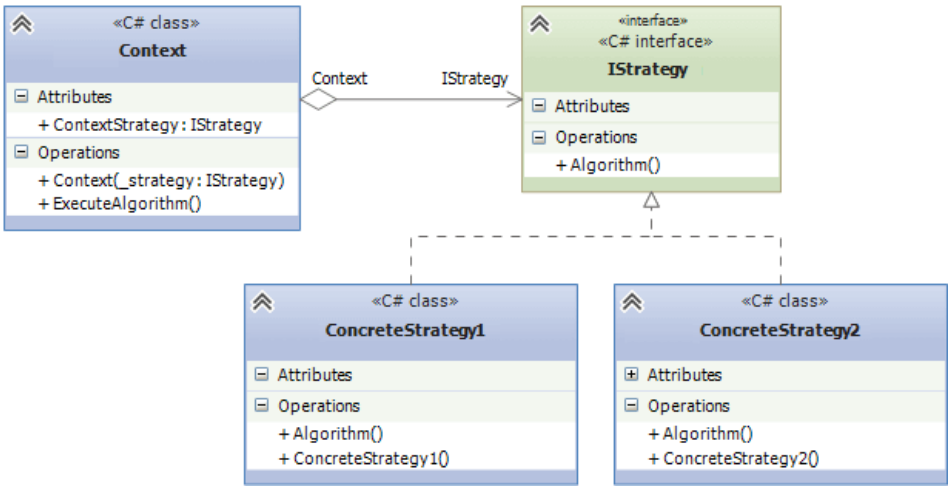
Теоретическая информация

Стратегия (Strategy). Паттерн *Стратегия* (*Strategy*) представляет шаблон проектирования, который определяет набор алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. В зависимости от ситуации мы можем легко заменить один используемый алгоритм другим. При этом замена алгоритма происходит независимо от объекта, который использует данный алгоритм.

Когда использовать стратегию?

1. Когда есть несколько родственных классов, которые отличаются поведением. Можно задать один основной класс, а разные варианты поведения вынести в отдельные классы и при необходимости их применять.
2. Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий.
3. Когда необходимо менять поведение объектов на стадии выполнения программы.
4. Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации.

Формально паттерн *Стратегия* можно выразить следующей схемой UML:



Формальное определение паттерна на языке C# может выглядеть следующим образом:

```
public interface IStrategy
{
     void Algorithm();
}

public class ConcreteStrategy1 : IStrategy
{
     public void Algorithm()
     {}
}

public class ConcreteStrategy2 : IStrategy
{
     public void Algorithm()
     {}
}

public class Context
{
     public IStrategy ContextStrategy { get; set; }

     public Context(IStrategy _strategy)
     {
          ContextStrategy = _strategy;
     }

     public void ExecuteAlgorithm()
     {
          ContextStrategy.Algorithm();
     }
}
```

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 3.

2. Применить паттерн *Стратегия* для обеспечения динамической смены поведения объектов приложения. Все записи массива основного класса должны представляться в различных форматах в зависимости от запроса пользователя (`html`: каждая запись обрамляется в теги `<p> </p>`; формат `text`: все записи выводятся с новой строки).

3. Сделать выводы.

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Составьте диаграмму классов.
3. Приведите листинг программы.
4. Разместите скриншоты результатов выполнения программы.
5. Сделайте краткие пояснения к алгоритму работы.
6. Сформулируйте выводы.

Дополнительные вопросы

1. Для чего служит паттерн *Стратегия*?
2. Как обеспечивается разнообразие поведения контекста?
3. Как возможно динамически менять поведение?
4. Чем заменяется понятие интерфейса в C++?
5. Как хранится объект стратегии в контексте?

ЛАБОРАТОРНАЯ РАБОТА № 14
Порождающие паттерны

Цель: научиться применять паттерн проектирования *Абстрактная фабрика* для создания объектов разных видов.

Теоретическая информация

Паттерн проектирования *Абстрактная фабрика* предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Применяется в следующих случаях:

– когда программа должна быть независимой от процесса и типов создаваемых новых объектов;

– когда необходимо создать семейства или группы взаимосвязанных объектов исключая возможность одновременного использования объектов из разных этих наборов в одном контексте.

Плюсы:

– изолирует конкретные классы;

– упрощает замену семейств продуктов;

– гарантирует сочетаемость продуктов.

Минусы:

– сложно добавить поддержку нового вида продуктов.

```
using System;
namespace DoFactory.GangOfFour.Abstract.Structural
{
  class MainApp
  {
    /// <summary>
    /// Точка входа в приложение
    /// </summary>
    public static void Main()
    {
      // Вызов абстрактной фабрики №1
      AbstractFactory factory1 = new ConcreteFactory1();
      Client client1 = new Client(factory1);
      client1.Run();

      // Вызов абстрактной фабрики №2
      AbstractFactory factory2 = new ConcreteFactory2();
```

```
    Client client2 = new Client(factory2);
    client2.Run();
    // Ожидание ввода
    Console.ReadKey();
  }
}

/// <summary>
/// Класс абстрактной фабрики
/// </summary>
abstract class AbstractFactory
{
  public abstract AbstractProductA CreateProductA();
  public abstract AbstractProductB CreateProductB();
}

/// <summary>
/// Класс фабрики №1
/// </summary>
class ConcreteFactory1 : AbstractFactory
{
  public override AbstractProductA CreateProductA()
  {
    return new ProductA1();
  }
  public override AbstractProductB CreateProductB()
  {
    return new ProductB1();
  }
}

/// <summary>
/// Класс фабрики №2
/// </summary>
class ConcreteFactory2 : AbstractFactory
{
  public override AbstractProductA CreateProductA()
  {
    return new ProductA2();
  }
  public override AbstractProductB CreateProductB()
  {
    return new ProductB2();
  }
}

/// <summary>
```

```
/// Абстрактный класс продукта A
/// </summary>
abstract class AbstractProductA
{
}
/// <summary>
/// Абстрактный класс продукта B
/// </summary>
abstract class AbstractProductB
{
  public abstract void Interact(AbstractProductA a);
}

/// <summary>
/// Первый класс продукта типа A
/// </summary>
class ProductA1 : AbstractProductA
{
}

/// <summary>
/// Первый класс продукта типа B
/// </summary>
class ProductB1 : AbstractProductB
{
  public override void Interact(AbstractProductA a)
  {
    Console.WriteLine(this.GetType().Name +
      " interacts with " + a.GetType().Name);
  }
}

/// <summary>
/// Второй класс продукта типа A
/// </summary>
class ProductA2 : AbstractProductA
{
}

/// <summary>
/// Второй класс продукта типа B
/// </summary>
class ProductB2 : AbstractProductB
{
  public override void Interact(AbstractProductA a)
  {
    Console.WriteLine(this.GetType().Name +
```

```
            " interacts with " + a.GetType().Name);
        }
    }

    /// <summary>
    /// Класс клиента, в котором происходит взаимодействие
    /// между объектами
    /// </summary>
    class Client
    {
        private AbstractProductA _abstractProductA;
        private AbstractProductB _abstractProductB;

        // Конструктор
        public Client(AbstractFactory factory)
        {
            _abstractProductB = factory.CreateProductB();
            _abstractProductA = factory.CreateProductA();
        }

        public void Run()
        {
            _abstractProductB.Interact(_abstractProductA);
        }
    }
}
```

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 12.
2. Применить один из порождающих паттернов для создания объектов классов-наследников.
3. Сделать выводы.

Варианты

На основании предыдущих лабораторных работ.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Составьте диаграмму классов.

3. Приведите листинг программы.
4. Разместите скриншоты результатов выполнения программы.
5. Сделайте краткие пояснения к алгоритму работы.
6. Сформулируйте выводы.

Дополнительные вопросы

1. Какие порождающие паттерны существуют?
2. Какова сфера применения паттерна *Абстрактная фабрика*?
3. Из какого набора классов состоит паттерн *Абстрактная фабрика*?
4. Какие недостатки имеет паттерн *Абстрактная фабрика*?
5. Можно ли с помощью *Абстрактной фабрики* создавать объекты с разным набором продуктов?

ЛАБОРАТОРНАЯ РАБОТА № 15
Структурные паттерны

Цель: научиться применять паттерн проектирования *Декоратор* для реализации объекта с дополнительной функциональностью.

Теоретическая информация

Задача. Объект, который предполагается использовать, выполняет основные функции. Однако может потребоваться добавить к нему некоторую дополнительную функциональность, которая будет выполняться до, после или даже вместо основной функциональности объекта.

Способ решения. *Декоратор* предусматривает расширение функциональности объекта без определения подклассов.

Участники. Класс `ConcreteComponent` – класс, в который с помощью шаблона *Декоратор* добавляется новая функциональность. В некоторых случаях базовая функциональность предоставляется классами, производными от класса `ConcreteComponent`. В подобных случаях класс `ConcreteComponent` является уже не конкретным, а абстрактным. Абстрактный класс `Component` определяет интерфейс для использования всех этих классов.

Следствия:

– добавляемая функциональность реализуется в небольших объектах. Преимущество состоит в возможности динамически добавлять эту функциональность до или после основной функциональности объекта `ConcreteComponent`;

– позволяет избегать перегрузки функциональными классами на верхних уровнях иерархии;

– декоратор и его компоненты не являются идентичными.

Реализация. Создаётся абстрактный класс, представляющий как исходный класс, так и новые, добавляемые в класс функции. В классах-декораторах новые функции вызываются в требуемой последовательности – до или после вызова последующего объекта.

При желании остаётся возможность использовать исходный класс (без расширения функциональности), если на его объект сохранилась ссылка.

Замечания и комментарии. Хотя объект-декоратор может добавлять свою функциональность до или после функциональности основного объекта,

цепочка создаваемых объектов всегда должна заканчиваться объектом класса `ConcreteComponent`.

Базовые классы языка Java широко используют шаблон *Декоратор* для организации обработки операций ввода-вывода.

И декоратор, и адаптер являются обёртками вокруг объекта – хранят в себе ссылку на оборачиваемый объект и часто передают в него вызовы методов. Отличие декоратора от адаптера в том, что адаптер имеет внешний интерфейс, отличный от интерфейса оборачиваемого объекта, и используется именно для стыковки разных интерфейсов, в то время как декоратор имеет точно такой же интерфейс и используется для добавления функциональности.

Для расширения функциональности класса возможно использовать как декораторы, так и стратегии. Декораторы оборачивают объект снаружи, стратегии же вставляются в него внутрь по неким интерфейсам.

Недостаток стратегии: класс должен быть спроектирован с возможностью добавления стратегий, декоратор же не требует такой поддержки.

Недостаток декоратора: он оборачивает ровно тот же интерфейс, что предназначен для внешнего мира, что вызывает смешение публичного интерфейса и интерфейса кастомизации, которое не всегда желательно.

Применение шаблона. Драйверы-фильтры в ядре Windows (архитектура WDM [Windows Driver Model]) представляют собой декораторы. Несмотря на то, что WDM реализована на необъектном языке Си, в ней чётко прослеживаются паттерны проектирования – декоратор, цепочка обязанностей и команда (объект IRP).

Архитектура COM (Component Object Model) не поддерживает наследование реализаций, вместо него предлагается использовать декораторы (в данной архитектуре это называется «агрегация»). При этом архитектура решает (с помощью механизма `pUnkOuter`) проблему `object identity`, возникающую при использовании декораторов: `identity` агрегата есть `identity` его самого внешнего декоратора.

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианту лабораторной работы № 3.
2. С помощью паттерна *Декоратор* декорировать объекты классов-наследников дополнительными свойствами.
3. Сделать выводы.

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Составьте диаграмму классов.
3. Приведите листинг программы.
4. Разместите скриншоты результатов выполнения программы.
5. Сделайте краткие пояснения к алгоритму работы.
6. Сформулируйте выводы.

Дополнительные вопросы

1. Из какого набора классов состоит паттерн *Декоратор*?
2. Как согласно паттерну реализована возможность «оборачивания» (как декоратор связан с компонентом)?
3. Что требуется выполнить для добавления нового декорирующего класса?
4. Чем отличаются компонент и декоратор?
5. Чем отличаются реализации операций в конкретных компонентах и конкретных декораторах?

ЛАБОРАТОРНАЯ РАБОТА № 16
Паттерн MVC

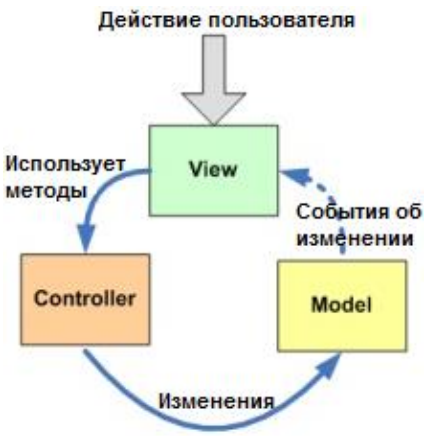
Цель: научиться применять составной паттерн MVC для разделения отображения и логики работы с данными.

Теоретическая информация

Шаблон *Model-View-Controller* – это методология разделения структуры приложения на специализированные компоненты. Схема MVC предполагает разделение всей системы на 3 взаимосвязанных компонента (подсистемы): так называемую *модель* (*Model*), *представление* (*View*) и *контроллер* (*Controller*). У каждого компонента своя цель, а главная особенность в том, что любой из них можно с легкостью заменить на другой или модифицировать, практически не затронув другие подсистемы. Преимущества такого подхода: модульность, расширяемость, простота поддержки и тестирования.

На диаграмме показана структура шаблона MVC.

Контроллер перехватывает событие извне и в соответствии с заложенной в него логикой реагирует на это событие, изменяя *Модель*, посредством вызова соответствующего метода. После изменения *Модель* использует событие о том, что она *изменилась*, и все подписанные на это события *Представления*, получив его, обращаются к *Модели* за обновленными данными, после чего их и отображают.



Model-View-Controller

Модель

Создадим класс модели задача которого – инкапсулировать всю логику работы с данными (`StudentModel`).

```
class StudentModel {
private:
	string group;
	string name;
public:
	string getGroup() {
		return group;
	}
```

```
        void setGroup(string group) {
            this->group = group;
        }
        string getName() {
            return name;
        }
        void setName(string name) {
            this->name = name;
        }
};
```

Теперь наша задача – адаптировать этот класс под шаблон `Observer`.

Шаблон «Наблюдатель» (`Observer`) определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

Это нужно для того, чтобы остальные классы приложения «знали» о любых изменениях в модели. Для этого создадим еще два класса, которые будут базовыми для остальных – это `Observable` (определяет методы для добавления, удаления и оповещения «наблюдателей») и `Observer` (класс, с помощью которого наблюдаемый объект оповещает наблюдателей).

```
class Observer{
public:
    virtual void update() = 0;
};
```

Класс `Observevable` содержит список всех «наблюдателей». Нового «наблюдателя» можно будет добавить с помощью метода `addObserver()`. При вызове метода `notifyUpdate()` класс `Observable` пройдется по списку «наблюдателей» и вызовет их методы `update()`, а они, в свою очередь, смогут каким-то образом на это отреагировать.

```
class Observable{
protected:
    vector<Observer*> _observers;
public:
    void addObserver(Observer *observer){
        _observers.push_back(observer);
    }
    void notifyUpdate(){
        int size = _observers.size();
        for (int i = 0; i < size; i++){
            _observers[i]->update();
        }
    }
};
```

Теперь нужно сделать класс StudentModel «оповещателем», чтобы у него в последствии могли быть «слушатели», следящие за его изменениями. Т.е. сделать класс TemperatureModel наследником класса Observable.

```
class StudentModel: public Observable {
private:
	string group;
	string name;
public:
	string getGroup() {
		return group;
	}
	void setGroup(string group) {
		this->group = group;
		notifyUpdate();
	}
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
		notifyUpdate();
	}
};
```

Обратите внимание, что добавились вызовы `notifyUpdate()` в методах модели. Таким образом мы достигаем нашей цели: «слушатели» будут оповещены в случае любых изменений в модели.

Представление

Теперь нам нужно создать представление – класс, выводящий изменения модели на консоль. Назовем его `ConsoleView`.

Класс `StudentView` является наследником класса `Observer`, потому он сможет получать сообщения от модели. `StudentView` хранит в себе указатель на модель, который передается в конструкторе. Обратите внимание, что там же `StudentView` «подписывает» себя на изменения модели вызовом метода `addObserver()`. Метод `update()` выводит текущие данные, вызывая метод `print()`.

```
class StudentView: public Observer{
private:
	StudentModel* model;
public:
	StudentView(StudentModel *model) {
		this->model = model;
```

```
            this->model->addObserver(this);
        }
        void update() {
            cout << "-=-=(view1 update)=-=-" << endl;
            print();
        }
        void print() {
            cout << "Student" << endl;
            cout << "Name: " << model->getName();
            cout << " Group: " << model->getGroup() << endl << endl;
        }
};
```

Для большей наглядности создадим второе представление `StudentView2`, которое отличается от первого форматом выводимых данных:

```
class StudentView2 : public Observer {
private:
    StudentModel* model;
public:
    StudentView2(StudentModel *model)
    {
        this->model = model;
        this->model->addObserver(this);
    }
    void update() {
        cout << "-=-=(view2 update)=-=-" << endl;
        print();
    }
    void print() {
        cout << "Student" << endl;
        cout << "--- NAME:  " << model->getName() << endl;
        cout << "--- GROUP: " << model->getGroup() << endl << endl;
    }
};
```

Контроллер

Класс контроллера, так же как и представление, получает ссылку на модель и представление в конструкторе. Также контроллер содержит методы для изменения модели.

Итоговая реализация:

```
#include "pch.h"
#include <iostream>
#include <string>
#include<vector>
using namespace std;
```

```
class Observer{
public:
	virtual void update() = 0;
};

class Observable{
protected:
	vector<Observer*> _observers;
public:
	void addObserver(Observer *observer){
		_observers.push_back(observer);
	}
	void notifyUpdate(){
		int size = _observers.size();
		for (int i = 0; i < size; i++){
			_observers[i]->update();
		}
	}
};

class StudentModel: public Observable {
private:
	string group;
	string name;
public:
	string getGroup() {
		return group;
	}
	void setGroup(string group) {
		this->group = group;
		notifyUpdate();
	}
	string getName() {
		return name;
	}
	void setName(string name) {
		this->name = name;
		notifyUpdate();
	}
};

class StudentView: public Observer{
private:
	StudentModel* model;
public:
	StudentView(StudentModel *model) {
		this->model = model;
		this->model->addObserver(this);
	}
	void update() {
		cout << "-=-=(view1 update)=-=-" << endl;
```

```
            print();
        }
        void print() {
            cout << "Student" << endl;
            cout << "Name: " << model->getName();
            cout << " Group: " << model->getGroup() << endl << endl;
        }
};

class StudentView2 : public Observer {
private:
        StudentModel* model;
public:
        StudentView2(StudentModel *model)
        {
            this->model = model;
            this->model->addObserver(this);
        }
        void update() {
            cout << "-=-=(view2 update)=-=-" << endl;
            print();
        }
        void print() {
            cout << "Student" << endl;
            cout << "--- NAME:  " << model->getName() << endl;
            cout << "--- GROUP: " << model->getGroup() << endl << endl;
        }
};

class StudentController {
private:
        StudentModel* model = new StudentModel();
        StudentView* view = new StudentView(model);

public:
        StudentController(StudentModel* model, StudentView* view) {
            this->model = model;
            this->view = view;
        }
        void setStudentName(string name) {
            model->setName(name);
        }
        void setStudentGroup(string group) {
            model->setGroup(group);
        }
};

int main()
{
        StudentModel* student = new StudentModel();
        student->setName("Robert");
        student->setGroup("10");
```

```
    StudentView* view = new StudentView(student);
    StudentView2* view2 = new StudentView2(student);

    view->print();
    view2->print();

    StudentController controller(student, view);
    controller.setStudentGroup("20");
// приведет к автоматическому обновлению модели
// и изменению представлений
    controller.setStudentName("John");
// приведет к автоматическому обновлению модели
// и изменению представлений
}
```

Результат:

```
Student
--- NAME:  Robert
--- GROUP: 10

-=-=(view1 update)=-=-
Student
Name: Robert Group: 20

-=-=(view2 update)=-=-
Student
--- NAME:  Robert
--- GROUP: 20

-=-=(view1 update)=-=-
Student
Name: John Group: 20

-=-=(view2 update)=-=-
Student
--- NAME:  John
--- GROUP: 20
```

Задание

1. Дополнить и при необходимости модифицировать приложение, разработанное согласно варианта предыдущих лабораторных работ.
2. Разработать приложение (применение Windows-форм повышает оценку) с применением шаблона MVC для разделения сложной модели (основной класс) и её представления.
3. Сделать выводы.

Варианты

Варианты распределяются аналогично предыдущим лабораторным работам.

Итоговый отчет

1. Укажите в отчете тему лабораторной работы, свою фамилию и имя, группу, вариант задания.
2. Составьте диаграмму классов;
3. Приведите листинг программы.
4. Разместите скриншоты результатов выполнения программы.
5. Сделайте краткие пояснения к алгоритму работы.
6. Сформулируйте выводы.

Дополнительные вопросы

1. Для чего служит шаблон MVC?
2. Какой паттерн проектирования лежит в основе шаблона MVC?
3. Из каких компонентов состоит шаблон MVC?
4. Как реализовать автоматическое обновление представления при смене модели?
5. Как реализовать смену модели при изменении данных в представлении?