ЛАБОРАТОРНАЯ РАБОТА №2 ПО ПРЕДМЕТУ «ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ И ПРОГРАММИРОВАНИЕ»

ТЕМА «РЕАЛИЗАЦИЯ ШАБЛОНОВ КЛАССОВ И ФУНКЦИЙ ПРИ ИСПОЛЬЗОВАНИИ SMART-УКАЗАТЕЛЕЙ И ТРАНЗАКЦИЙ»

ЦЕЛЬ РАБОТЫ: научиться использовать шаблоны функций и классов при работе со *smart*-указателями и транзакциями.

ШАБЛОНЫ ФУНКЦИЙ И КЛАССОВ

В языке *C++* ключевое слово *template* используется для обеспечения параметрического полиморфизма. Шаблонные классы и функции позволяют многократно использовать код, корректно по отношению к различным типам данных.

Шаблоны в *C++* являются средствами метапрограммирования и реализуют полиморфизм времени компиляции. Когда мы пишем код с полиморфным поведением, но само поведение определяется на этапе компиляции, в противовес полиморфизму виртуальных функций, полученный бинарный код уже будет иметь постоянное поведение.

Шаблоны позволяют достичь одну из самых трудных целей в программировании – создать многократно используемый код. В обобщенной функции или классе обрабатываемый ею (им) тип данных задается как параметр. Таким образом, одну функцию или класс можно использовать для различных типов данных, не предоставляя явным образом конкретные версии для каждого типа данных.

Механизм шаблонов в языке *C++* позволяет решать проблему унификации алгоритма для различных типов: нет необходимости писать различные функции для целочисленных, действительных или пользовательских типов. Достаточно создать обобщенный алгоритм, не зависящий от типа данных, основывающийся только на общих свойствах. Например, алгоритм сортировки может работать как с целыми числами, так и с объектами типа «автомобиль».

Обобщенная функция перегружает саму себя. Обобщенная функция определяет общий набор операций, которые предназначены для применения к данным различных типов. Тип данных, обрабатываемых функцией, передается ей как параметр. Используя обобщенную функцию к широкому диапазону данных, можно применить единую общую процедуру. По описанию шаблон функции похож на обычную функцию. Разница в том, что типы некоторых элементов в шаблонной функции не определены и являются параметризованными.

Обобщенная функция создается с помощью ключевого слова *template*. Значение слова "*template*" (шаблон, образец, лекало) точно отражает цель его применения в языке *C++*. Это ключевое слово используется для создания шаблона (или оболочки), который описывает действия, выполняемые функцией. Компилятору же остается "дополнить недостающие детали" в

соответствии с заданным значением параметра. Общий формат определения шаблонной функции имеет следующий вид.

```
template <class T>
тип имя_функции(список_параметров) {
     //тело функции
}
```

Определение обобщенной функции начинается с ключевого слова *template*. Здесь элемент *T* представляет собой "заполнитель" для типа данных, обрабатываемых функцией. Это имя может быть использовано в теле функции. Но оно означает всего лишь заполнитель, вместо которого компилятор автоматически подставит конкретный тип данных при создании конкретной версии функции. И хотя для задания обобщенного типа в *template*-объявлении применяется ключевое слово *class*, можно также использовать ключевое слово *typename*.

В следующем примере создается обобщенная функция, которая меняет местами значения двух переменных, используемых при ее вызове. Общий процесс обмена значениями переменных не зависит от их типа.

```
//Пример №1. Использование шаблонной функции
#include <iostream>
#include <windows.h>
using namespace std;
template <class T>
void swapArgs(T& ob1, T& ob2) {
    T temp;
    temp = ob1;
    ob1 = ob2;
    ob2 = temp;
}
void ex1() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    int i = 10, j = 20;
    double x = 10.1, y = 23.3;
    char a = 'x', b = 'z';
    cout << "Исходные значения i, j: " << i << ' ' << j << '\n';
    cout << "Исходные значения x, y: " << x << ' ' << y << '\n';
    cout << "Исходные значения a, b: " << a << ' ' << b << '\n';
    swapArgs(i, j); //перестановка целых чисел
    swapArgs(x, y); //перестановка значений с плавающей точкой
    swapArgs(a, b); //перестановка символов
    cout << "После перестановки i, j: " << i << ' ' << j << '\n';
    cout << "После перестановки x, y: " << x << ' ' << y << '\n';
    cout << "После перестановки a, b: " << a << ' ' << b << '\n';
}
```

Результаты выполнения этой программы.

```
Исходные значения i, j: 10 20
Исходные значения x, y: 10.1 23.3
Исходные значения a, b: x z
После перестановки i, j: 20 10
После перестановки x, y: 23.3 10.1
После перестановки a, b: z x
```

Строка

```
template <class T>
void swapArgs(T& ob1, T& ob2) {
```

сообщает компилятору, что создается шаблонная функция, и что здесь начинается её обобщенное определение. Обозначение *T* представляет собой обобщенный тип, который используется в качестве "заполнителя". За *template*-заголовком следует объявление функции *swapArgs(),* в котором символ *X* означает тип данных для значений, которые будут меняться местами. В функции *main()* демонстрируется вызов функции *swapArgs()* с использованием трех различных типов данных: *int, double* и *char*. Поскольку функция *swapArgs()* является обобщенной, компилятор автоматически создает три версии функции *swapArgs():* одну для обмена целых чисел, вторую для обмена значений с плавающей точкой и третью для обмена символов.

Обобщенная функция (т.е. функция, объявление которой предваряется *template*-инструкцией) также называется *шаблонной функцией*. Когда компилятор создает конкретную версию этой функции, то говорят, что создается ее специализация. Специализация функции также называется порожденной функцией (*generated function*). Другими словами, порождаемая функция является конкретным экземпляром шаблонной функции.

Поскольку язык *C++* не распознает символ конца строки в качестве признака конца инструкции, *template*-часть определения обобщенной функции может не находиться в одной строке с именем этой функции. В следующем примере показан еще один распространенный способ написания шаблонной функции.

```
template <class X>
void swapArgs(X& ob1, X& ob2) {
    X temp;
    temp = ob1;
    ob1 = ob2;
    ob2 = temp;
}
```

При использовании этой формы объявления функции необходимо понимать, что между *template*-инструкцией и началом определения обобщенной функции никакие другие инструкции находиться не могут. Например, следующий фрагмент кода не скомпилируется.

```
template <class X>
int i;//incomplete type is not allowed
void swapArgs(X& ob1, X& ob2) {
    X temp;
    temp = ob1;
```

```
    ob1 = ob2;
    ob2 = temp;
     }
```

Template-спецификация должна стоять непосредственно перед определением функции. Между ними не может находиться какая-либо инструкция.

ФУНКЦИЯ С ДВУМЯ ОБОБЩЕННЫМИ ТИПАМИ

В *template*-инструкции можно определить несколько обобщенных типов данных, используя список элементов, разделенных запятой. Например, в следующей программе создается шаблонная функция с двумя обобщенными типами.

```
//Пример №2. Шаблонная функция с двумя обобщенными типами
#include <iostream>
using namespace std;
template <class T1, class T2>
void display(T1 x, T2 y) {
    cout << x << ' ' << y << endl;
}
void ex2() {
    system("chcp 1251");
    system("cls");
    display(10, "Информация для вывода на консоль");
    display(0.23, 10L);
}
```

Результаты работы программы:

```
10 Информация для вывода на консоль
0.23 10
```

В этом примере при выполнении функции *main(),* когда компилятор генерирует конкретные экземпляры функции *ex2(),* заполнители типов *T1* и *T2* заменяются сначала парой типов данных *int* и *char**, а затем парой *double* и *long* соответственно.

Создавая шаблонную функцию компилятору разрешается генерировать столько различных версий этой функции, сколько необходимо для обработки различных способов её вызова.

ПЕРЕГРУЗКА ШАБЛОНА ФУНКЦИИ

Помимо создания явным образом перегруженных версий обобщенной функции, можно также перегружать саму спецификацию шаблона функции. Для этого достаточно создать еще одну версию шаблона, которая будет отличаться от остальных списком параметров.

```
//Пример №3. Создание перегруженного шаблона функции
#include <iostream>
using namespace std;
//Первая версия шаблонной функции
template <class T>
void display(T ob) {
    cout << "Выполняется функция concat(T ob)\n";
    cout << ob << endl;
}
```

```
//Вторая версия шаблонной функции
template <class T1, class T2>
void display(T1 ob1, T2 ob2) {
    cout << "Выполняется функция concat(T1 ob1, T2 ob2)\n";
    cout << ob1 << " " << ob2;
}
void ex3() {
    system("chcp 1251");
    system("cls");
    display("Первая строка"); //вызов display(T ob)
    display("Первая строка", "Вторая строка"); //вызов display(T1
ob1, T2 ob)
}
```

Результаты работы программы:

```
Выполняется функция concat(T ob)
Первая строка
Выполняется функция concat(T1 ob1, T2 ob2)
Первая строка Вторая строка
```

Здесь шаблон для функции *display()* перегружается, чтобы обеспечить возможность приема как одного, так и двух параметров.

Обобщенные функции подобны перегруженным функциям, но имеют больше ограничений по применению. При перегрузке функций в теле каждой из них обычно задаются различные действия. Но обобщенная функция должна выполнять одно и то же действие для всех версий — отличие между версиями состоит только в типе данных.

ОБОБЩЕННЫЕ КЛАССЫ

Шаблоны классов – обобщенное описание пользовательского типа, в котором могут быть параметризованы как поля, так и методы. Шаблоны классов представляют собой конструкции, по которым могут быть сгенерированы классы путём подстановки вместо параметров типов конкретных типов данных.

Для создания обобщенного класса необходимо создать класс, в котором определяются все используемые им алгоритмы; при этом реальный тип обрабатываемых в нем данных будет задан как параметр при создании объектов этого класса.

Общий формат объявления обобщенного класса имеет следующий вид:

```
template <class T>
class ClassName {
    //class content
     };
```

Здесь элемент *T* представляет собой "заполнитель" для типа данных, который будет задан при инстанциировании класса. При необходимости можно определить несколько обобщенных типов данных, используя список элементов, разделенных запятыми. Идентификатору типа предшествует ключевое слово *class* или *typename*.

Создав обобщенный класс, можно создать его конкретный экземпляр, используя следующий общий формат.

```
ClassName <тип данных> objectName;
```

Здесь элемент *тип данных* означает имя типа данных, который будет использоваться экземпляром обобщенного класса. Метода обобщенного класса автоматически являются обобщенными. Поэтому не нужно использовать ключевое слово *template* для явного определения методов класса шаблонными.

В следующей программе класс *Queue* может быть использован для организации очереди объектов любого типа. В данном примере создаются две очереди: для целых чисел и для дробных значений.

```
//Пример №4. Демонстрация использования обобщенного класса очереди
#include <iostream>
using namespace std;
const int SIZE = 100;
//Создание обобщенного класса Queue
template <class T>
class Queue {
    T queue[SIZE];
    int indexAdd, indexGet;
public:
    Queue() { indexAdd = indexGet = 0; }
    void put(T element);
    T get();
};
//добавление элемента в очередь
template <class T>
void Queue<T>::put(T element) {
    if (indexAdd == SIZE) {
        cout << "Очередь заполнена.\n";
        return;
    }
    indexAdd++;
    queue[indexAdd] = element;
}
//извлечение элемента из очереди
template <class T>
T Queue<T>::get() {
    if (indexGet == indexAdd) {
        cout << "Очередь пуста.\n";
        return 0;
    }
    indexGet++;
    return queue[indexGet];
}
void ex4() {
    Queue<int> queue1, queue2; //Создаем две очереди для целых
чисел
    queue1.put(10);
    queue1.put(20);
    queue2.put(19);
    queue2.put(1);
    cout << queue1.get() << " ";
    cout << queue1.get() << " ";
```

```
    cout << queue2.get() << " ";
    cout << queue2.get() << "\n";
    Queue<double> queue3, queue4;//Создаем две очереди для дробных
чисел
    queue3.put(10.12);
    queue3.put(-20.0);
    queue4.put(19.99);
    queue4.put(0.986);
    cout << queue3.get() << " ";
    cout << queue3.get() << " ";
    cout << queue4.get() << " ";
    cout << queue4.get() << "\n";
}
```

При выполнении этой программы получаем следующие результаты.

```
10 20 19 1
10.12 -20 19.99 0.986
```

В этой программе объявление обобщенного класса подобно объявлению обобщенной функции. Тип данных, хранимых в очереди, обобщен в объявлении класса. Он неизвестен до тех пор, пока не будет объявлен объект класса *Queue*, который и определит реальный тип используемых данных. После объявления конкретного экземпляра класса *Queue* компилятор автоматически сгенерирует все функции и переменные, необходимые для обработки реальных типов данных. В данном примере объявляются четыре различные по типу хранимых элементов очереди: две очереди для хранения целых чисел и две очереди для значений типа *double*. Обратите особое внимание на эти объявления:

```
Queue<int> queue1, queue2;
Queue<double> queue3, queue4;
```

Нужный тип данных объектов при объявлении заключается в угловые скобки. Изменяя тип данных при создании объектов класса *Queue*, можно изменить тип данных, хранимых в очереди. Например, используя следующее объявление, можно создать еще одну очередь, которая будет содержать указатели на символы:

```
Queue<char*> queueChar;
```

Обобщенные функции и классы представляют собой средства, которые помогут увеличить эффективность работы программиста, поскольку они позволяют определить общий формат объекта, который можно затем использовать с любым типом данных. Обобщенные функции и классы избавляют от создания отдельных реализаций для каждого типа данных, подлежащих обработке единым алгоритмом. Эту работу сделает компилятор: он автоматически создаст конкретные версии шаблонного класса.

Шаблонный класс может иметь несколько обобщенных типов данных. Для этого достаточно объявить все нужные типы данных в *template*-спецификации в виде элементов списка, разделенных запятыми. Например, в следующей программе создается класс, который использует два обобщенных типа данных.

Далее будет приведен пример определения шаблона класса массива (вектора). Какой бы тип ни имели элементы массива (целый, вещественный, с двойной точностью и т. д.), в этом классе должны быть определены одни и те же базовые операции, например, доступ к элементу по индексу, помещение элемента в массив и т. д. Если тип элементов вектора задавать как параметр шаблона класса, то система будет формировать массив нужного типа (и соответствующий класс) при каждом определении конкретного объекта.

В программе шаблон семейства классов с общим именем *Vector* используется для формирования двух классов с массивами целого и символьного типов. В соответствии с требованием синтаксиса имя параметризованного класса, определенное в шаблоне (*Vector*), используется в программе только с последующим конкретным фактическим параметром (аргументом), заключенным в угловые скобки. Параметром может быть имя стандартного или определенного пользователем типа. В данном примере использованы стандартные типы *int* и *char*. Использовать тип *Vector* без указания фактического параметра шаблона нельзя, т. к. никакое значение по умолчанию не предусмотрено. В списке параметров шаблона могут присутствовать формальные параметры, не определяющие тип, точнее – это параметры, для которых тип фиксирован.

Рассмотрим пример использования шаблонного класса, реализующего вектор. Реализованы некоторые методы для работы с вектором, а также перегружен оператор [].

```
//Пример №5. Использование шаблонного класса, реализующего вектор
#include <iostream>
#include <string.h>
#include <windows.h>
#include <iomanip>
using namespace std;
template <class T>
class Vector {
    T* pointer; //указатель на начало вектора
    int size, index; //размер вектора и индекс элемента
public:
    Vector() : size(0), index(0), pointer(NULL) {}
    Vector(int size);
    ~Vector() { delete[] pointer; }
    void set(const T& elem); //добавление элемента в вектор
    T* getBegin(); //возвращается указатель на начало вектора
    int getSize(); //возвращается размер вектора
    T& operator[](int index); //перегрузка оператора []
};
template <class T>
Vector<T>::Vector(int size) : size(size), index(0) {
    pointer = new T[size];
    const type_info& type = typeid(T); //получение ссылки типа
параметра T
    const char* typeName = type.name();
    for (int i = 0; i < size; i++) //в зависимости от типа T
        if (!strcmp(typeName, "char")) *(pointer + i) = ' ';
//записываем пустой символ
        else *(pointer + i) = 0; //записываем 0
```

```
}
template <class T>
void Vector<T>::set(const T& elem) { //добавление элемента в массив
    T* tmp = NULL;// временный массив
    if (++index >= size) {
        tmp = pointer;
        pointer = new T[size + 1];
    }
    //memcpy() copies bytes between buffers
    if (tmp) memcpy(pointer, tmp, sizeof(T) * size); //перезапись
tmp в pointer
    pointer[size++] = elem; //добавление нового элемента  в массив
    if (tmp) delete[] tmp; //удаление временного массива
}
template <class T>
T* Vector<T>::getBegin() { return pointer; }
template <class T>
int Vector<T>::getSize() { return size; }
template <class T>
T& Vector<T>::operator[](int n) { //перегрузка оператора []
    if (n < 0 || (n >= size)) n = 0; //контроль выхода за границы
массива
    return pointer[n];
}
void ex5() {
    Vector <int> intVect;
    Vector <char> charVect;
    intVect.set(3);
    intVect.set(26);
    intVect.set(12);
    charVect.set('A');
    charVect.set('B');
    cout << intVect[0] << endl;
    cout << charVect[0] << endl;
    intVect[0] = 111;
    charVect[0] = 'C';
    int* intPointer = intVect.getBegin();
    //вывод целочисленного вектора
    for (int i = 0; i < intVect.getSize(); i++)
        cout << setw(3) << *(intPointer + i);
    cout << endl;
    char* charPointer = charVect.getBegin();
    //вывод символьного вектора
    for (int i = 0; i < charVect.getSize(); i++)
        cout << setw(3) << *(charPointer + i);
    cout << endl;
}
```

Результаты работы программы:

ДИНАМИЧЕСКАЯ ПАМЯТЬ И ИНТЕЛЛЕКТУАЛЬНЫЕ УКАЗАТЕЛИ

Для управления динамической памятью в языке *C++* используются два оператора:

– **оператор *new***, который резервирует, а при необходимости и инициализирует, объект в динамической памяти и возвращает указатель на него;

– **оператор *delete***, который получает указатель на динамический объект и удаляет его, освобождая зарезервированную память.

Интеллектуальный указатель (умный указатель, *smart pointer*) действует, как обычный указатель, но с важным дополнением: *smart-pointer* автоматически удаляет объект, на который он указывает. Умный указатель обычно является шаблонным классом. Чаще всего умный указатель инкапсулирует семантику **владения ресурсом**. В таком случае он называется **владеющим указателем**. Владеющие указатели применяются для борьбы с утечками памяти и висячими ссылками.

Утечкой памяти (*memory leak*) называется ситуация, когда в программе нет ни одного указателя, хранящего адрес объекта, созданного в динамической памяти.

Висячим указателем (*dangling pointer, wild pointer*) называется указатель, ссылающийся на уже удалённый объект или не ссылающийся на допустимый объект соответствующего типа.

Семантика владения для динамически созданных объектов означает, что удаление или присваивание нового значения указателю должно быть согласовано с временем жизни этого объекта.

Библиотека *C++* определяет три типа интеллектуальных указателей, отличающихся способом управления:

– указатель типа ***std::shared_ptr*** позволяет **нескольким** указателям указывать на один и тот же объект;

– указатель типа ***std::unique_ptr*** позволяет **только одному** указателю указывать на объект. В отличие от указателя *std::shared_ptr,* только один указатель типа *std::unique_ptr* может одновременно указывать на конкретный объект;

– указатель типа ***std::weak_ptr*** является **второстепенной ссылкой на объект**, управляемый указателем *std::shared_ptr*. Указатель типа *std::weak_ptr* моделирует временное владение: когда объект должен быть доступен только если он существует и может быть удален в любой момент кем-то другим.

Интеллектуальные указатели определены в пространстве имен *std* в заголовочном файле <*memory*>.

Интеллектуальные указатели важны для реализации идиомы программирования *RAII* или *Resource Acquisition Is Initialization* — получение ресурса является инициализацией. То есть, **при получении какого-либо ресурса, его инициализируют в конструкторе, а, поработав с ним, корректно освобождают в деструкторе.** Ресурсом может быть что угодно, к примеру файл, сетевое соединение или блок памяти.

Главная задача идиомы *RAII* — обеспечить, чтобы одновременно с получением ресурса производилась инициализация объекта, чтобы все ресурсы для объекта создавались и подготавливались в одном блоке кода. На практике основным принципом *RAII* является предоставление владения любым ресурсом в куче (например, динамически выделенной памятью или

дескрипторами системных объектов) объекту, выделенному стеком. Деструктор используемого ресурса должен содержать код удаления или освобождения всех задействованных объектов, а также весь необходимый код очистки.

При работе с динамическими объектами, используются указатели на них, а в случае, если объект становится не нужен, то объект необходимо уничтожить. В то же время на один объект могут ссылаться множество указателей приложения и, следовательно, нельзя однозначно сказать, нужен ли на данный момент этот объект или он уже может быть уничтожен без каких-либо последствий для других частей приложения.

Рассмотрим пример реализации класса, для которого происходит уничтожение объектов, в случае если уничтожается последняя ссылка на него. Это достигается, например, тем, что создается дополнительная структура, в которой наряду с указателем на динамический объект хранится так же счетчик числа указателей, «прикрепленных» к этому объекту. Динамический созданный объект может быть уничтожен только в том случае, если счетчик ссылок на этот объект станет равным нулю.

```
//Пример №6. Создание и использование умного указателя
#include <iostream>
#include <windows.h>
using namespace std;
class Employee {//класс, количество объектов которого будет
отслеживаться
	int age;
	string department;
	string position;
public:
	void setAge(int age) { this->age = age; }
	void setDepartment(string department) { this->department =
department; }
	void setPosition(string position) { this->position = position;
}
	void showDepartment() { cout << "Сотрудник работает в
подразделении: " << department << endl; }
	void showPosition() { cout << "Сотрудник работает на
должности: " << position << endl; }
	void showAge() { cout << "Возраст сотрудника: " << age <<
endl; }
};
template <class T>
struct Status {
	T* ptr;//указатель на объект
	int counter;//счетчик количества ссылок на объект
};
//Класс умного указателя. Умный указатель должен вести подсчет
количества созданный ссылок на объект
template <class T>
class SmartPointer {
	Status<T>* smartPtr;// указатель на объект
public:
	SmartPointer(T* ptr);//конструктор с одним параметром типа T
```

```
	SmartPointer(const SmartPointer& obj);//конструктор
копирования
	~SmartPointer();//деструктор
	SmartPointer& operator=(const SmartPointer& obj); //перегрузка
оператора присваивания
	T* operator->() const;//оператор получения доступа к объекту
через указатель
	void showCounter() { cout << "Значение счетчика для объекта "
<< smartPtr << " равно: " << smartPtr->counter << endl; }
};
template <class T>
SmartPointer<T>::SmartPointer(T* ptr) {
	if (!ptr)
		smartPtr = NULL;//указатель на объект пустой
	else {
		smartPtr = new Status<T>();
		smartPtr->ptr = ptr;//инициализирует объект указателем
		smartPtr->counter = 1;//счетчик «прикрепленных» объектов
инициализируем единицей
	}
}
template <class T>
SmartPointer<T>::SmartPointer(const SmartPointer& obj)
:smartPtr(obj.smartPtr) {
	if (smartPtr) smartPtr->counter++;//увеличение количества
ссылок на объект
}
template <class T>
SmartPointer<T>::~SmartPointer() {
	if (smartPtr) {
		smartPtr->counter--;//уменьшение количества ссылок на
объект
		if (smartPtr->counter == 0) {//если число ссылок на
объект равно нулю, то должен уничтожится объект
			delete smartPtr->ptr;//освобождение памяти,
выделенной для объекта
			delete smartPtr;//освобождение памяти, выделенной
для указателя
		}
	}
}
template <class T>
T* SmartPointer<T>::operator->() const {
	if (smartPtr) return smartPtr->ptr;
	else return NULL;
}
template <class T>
SmartPointer<T>& SmartPointer<T>::operator=(const SmartPointer&
obj) {
	if (smartPtr) {
		smartPtr->counter--;//уменьшаем счетчик «прикрепленных»
объектов для текущего указателя
		if (smartPtr->counter == 0) {//если объектов больше нет,
то необходимо освобождить выделенную память
			delete smartPtr->ptr;
			delete smartPtr;
```

```
            }
        }
        smartPtr = obj.smartPtr;//присваивание нового адреса указателю
        if (smartPtr) smartPtr->counter++;//увеличить счетчик
«прикрепленных» объектов
        return *this;
}
void ex6() {
        SetConsoleCP(1251);
        SetConsoleOutputCP(1251);
        SmartPointer<Employee> employeeIT = new Employee();
        employeeIT->setDepartment("Отдел аналитики");
        employeeIT->showDepartment();
        employeeIT.showCounter();
        SmartPointer<Employee> employeeAdmin(new Employee);
        SmartPointer<Employee> employeeService = employeeIT;
        employeeIT.showCounter();
        employeeAdmin.showCounter();
        employeeService.showCounter();
        employeeAdmin = employeeIT;
        employeeAdmin.showCounter();
        //когда область видимости закончится, объект будет удален
}
```

Результаты работы программы:

```
Сотрудник работает в подразделении: Отдел аналитики
Значение счетчика для объекта 000001E5C73E2C40 равно: 1
Значение счетчика для объекта 000001E5C73E2C40 равно: 2
Значение счетчика для объекта 000001E5C73E2BA0 равно: 1
Значение счетчика для объекта 000001E5C73E2C40 равно: 2
Значение счетчика для объекта 000001E5C73E2C40 равно: 3
```

ИСПОЛЬЗОВАНИЕ ТРАНЗАКЦИЙ

Концепция *smart*-указателей позволяет решать задачу поддержки транзакций. **Транзакция** в информатике – группа последовательных операций, которая представляет собой логически неделимую единицу работы с данными.

Транзакция может быть выполнена либо целиком и успешно, соблюдая целостность данных и независимо от параллельно идущих других транзакций, либо не выполнена вообще и тогда она не должна произвести никакого эффекта над данными (ни одна из частей транзакции не должна быть выполнена). При этом должно соблюдаться следующее:

– если клиент начал и не завершил транзакцию, то другие клиенты не видят его изменений;

– две транзакции не могут одновременно менять одни данные.

Для поддержки механизма транзакции объект должен содержать **два указателя** (на текущий объект и на объект в его предыдущем состоянии) и функции (начать транзакцию, зафиксировать транзакцию, отменить транзакцию, повторить транзакцию). Если требуется закрепить или отменить выполненные транзакции, то необходимо хранить состояние объекта на заданный момент – начало транзакции, и в момент принятия решения фиксации транзакции или уничтожать предыдущее состояние (закрепление транзакции), или возвращаться к предыдущему состоянию (отмена

транзакции). Базовые элемента механизма транзакций можно рассмотреть на примере следующей программы.

```
//Пример №7. Использование механизма транзакции
#include <windows.h>
#include <iostream>
using namespace std;
enum State { begin, middle };
// класс, над объектами которого выполняются транзакции
template <class T>
class Account {
	T number;
public:
	Account() : number(0) {}
	void setNumber(int number) { this->number = number; }
	T getNumber() { return number; }
};
// класс транзакции
template <class T>
class Transaction {
	T* currentState;//текущее значение объекта класса
	T* prevState;//предыдущее состояние объекта
public:
	Transaction() :prevState(NULL), currentState(new T)
{}//конструктор без параметров
	Transaction(const Transaction& obj) : currentState(new
T(*(obj.currentState))), prevState(NULL) {}// конструктор
копирования
	~Transaction() { delete currentState; delete prevState;
}//деструктор
	Transaction& operator=(const Transaction& obj);//перегрузка
оператора присваивания
	void showState(State state);//отображение состояний
(предыдущего и текущего) объекта
	int beginTransactions(int accountNumber);//метод начала
транзакции
	void commit();//метод подтверждения (коммита) транзакции
	void deleteTransactions(); //метод отката транзакции
	T* operator->();//перегрузка оператора доступа к содержимому
объекта через указатель
};
template <class T>
Transaction<T>& Transaction<T>::operator=(const Transaction<T>&
obj) {
	if (this != &obj) {//проверка на случай obj=obj
		delete currentState;//удаление текущего значения объекта
		currentState = new T(*(obj.currentState));//создание и
копирование объекта, используя присваеваемую транзакцию
	}
	return *this;
}
template <class T>
T* Transaction<T>::operator->() {
	return currentState;
}
```

```
template <class T>
void Transaction<T>::showState(State state) {//метод отображение
состояния объекта
	cout << "Состояние объекта ";
	if (!state) cout << "до начала транзакции " << endl;
	else cout << "после выполнения транзакции " << endl;
	if (prevState) cout << "prevState = " << prevState-
>getNumber() << endl;//предыдущее состояние
	else cout << "prevState = NULL" << endl;
	cout << "currentState = " << currentState->getNumber() <<
endl;//текущее состояние
}
template <class T>
int Transaction<T>::beginTransactions(int accountNumber) {// метод
начала транзакции
	delete prevState;//удаление предыдущего значения
	prevState = currentState;//сохранение предыдущего состояния
как текущего
	currentState = new T(*prevState);//текущее состояние создается
!!!!!!!!!!!!!!
	if (!currentState) return 0;//ошибка (необходимо отменить
транзакцию)
	currentState->setNumber(accountNumber);//изменение состояния
объекта
	return 1;//успешное окончание транзакции
}
template <class T>
void Transaction<T>::commit() {
	delete prevState;//удаление предыдущего значения
	prevState = NULL;//предыдущего состояния нет
}
template <class T>
void Transaction<T>::deleteTransactions() {
	if (prevState != NULL) {
		delete currentState;//удаление текущего значения
		currentState = prevState;//предыдущее становится текущим
		prevState = NULL;//предыдущего состояния нет
	}
}
void ex7() {
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	Transaction<Account<int>> firstAccount;
	firstAccount->setNumber(5);//начальная инициализация объекта
	firstAccount.showState(State::begin);
	cout << "НАЧАЛЬНАЯ ИНИЦИАЛИЗАЦИЯ ПРОШЛА УСПЕШНО" << endl;
	cout << "-----------------------------------------------"
<< endl;
	cout << "ПЕРВАЯ ТРАНЗАКЦИЯ НАЧАТА СО ЗНАЧЕНИЕМ 10" << endl;
	if (firstAccount.beginTransactions(10)) {//начало транзакции
(изменение объекта)
		firstAccount.showState(State::middle);
	}
	cout << "ПЕРВАЯ ТРАНЗАКЦИЯ ПРОШЛА УСПЕШНО" << endl;
	cout << endl;
```

```
	cout << "-----------------------------------------------"
<< endl;
	firstAccount.deleteTransactions();//отмена транзакции при
ошибке
	cout << "ПЕРВАЯ ТРАНЗАКЦИЯ БЫЛА ОТМЕНЕНА" << endl;
	firstAccount.commit();//закрепление транзакции
	firstAccount.showState(State::middle);
	cout << "-----------------------------------------------"
<< endl;
	cout << "ВТОРАЯ ТРАНЗАКЦИЯ НАЧАТА СО ЗНАЧЕНИЕМ 2" << endl;
	if (firstAccount.beginTransactions(2)) {//начало транзакции
(изменение объекта)
		firstAccount.showState(State::begin);
		firstAccount.commit();//закрепление транзакции
	}
	cout << "ВТОРАЯ ТРАНЗАКЦИЯ ПРОШЛА УСПЕШНО" << endl;
	firstAccount.showState(State::middle);
}
```

Результаты работы программы:

```
Состояние объекта до начала транзакции
prevState = NULL
currentState = 5
НАЧАЛЬНАЯ ИНИЦИАЛИЗАЦИЯ ПРОШЛА УСПЕШНО
-----------------------------------------------
ПЕРВАЯ ТРАНЗАКЦИЯ НАЧАТА СО ЗНАЧЕНИЕМ 10
Состояние объекта после выполнения транзакции
prevState = 5
currentState = 10
ПЕРВАЯ ТРАНЗАКЦИЯ ПРОШЛА УСПЕШНО

-----------------------------------------------
ПЕРВАЯ ТРАНЗАКЦИЯ БЫЛА ОТМЕНЕНА
Состояние объекта после выполнения транзакции
prevState = NULL
currentState = 5
-----------------------------------------------
ВТОРАЯ ТРАНЗАКЦИЯ НАЧАТА СО ЗНАЧЕНИЕМ 2
Состояние объекта до начала транзакции
prevState = 5
currentState = 2
ВТОРАЯ ТРАНЗАКЦИЯ ПРОШЛА УСПЕШНО
Состояние объекта после выполнения транзакции
prevState = NULL
currentState = 2
```

КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНОЙ РАБОТЕ:

1. Каковы особенности использования динамической памяти в *C++*?

2. С какой целью используются *smart*-указатели? Каковы особенности их использования? Что представляет собой *smart*-указатель?

3. Какие виды интеллектуальных указателей предоставляет библиотека *C++*?

4. Какие методы должны быть реализованы в собственном классе «умного указателя»? Приведите пример реализации двух таких методов.

5. Каковы особенности использования указателя *std::shared_ptr*? Приведите пример его создания и инициализации.

6. Каковы особенности использования указателя *std::unique_ptr*? Приведите пример его создания и инициализации.

7. Каковы особенности использования указателя *std::weak_ptr*? Приведите пример его создания и инициализации.

8. Что такое утечка памяти в программировании? Приведи пример кода.

9. Что такое висячий указатель в программировании? Приведи пример кода.

10. Что собой представляет идиома программирования *RAII*?

11. Что такое транзакция в программировании? В чем состоит механизм транзакций? Когда используются транзакции?

12. Какие основные операции используются при реализации механизма транзакций? Приведите пример реализации одной из операций.

ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ:

1. Изучить теоретические сведения, полученные на лекции и лабораторной работе, ознакомиться с соответствующими материалами литературных источников.
2. Ответить на контрольные вопросы лабораторной работы.
3. Разработать алгоритм программы по индивидуальному заданию.
4. Написать, отладить и проверить корректность работы созданной программы.
5. Написать электронный отчет по выполненной лабораторной работе.

Отчет должен быть оформлен по стандарту БГУИР (Стандарт предприятия СТП 01-2017 "Дипломные проекты (работы). Общие требования"**) и иметь следующую структуру:**

1. титульный лист
2. цель выполнения лабораторной работы
3. теоретические сведения по лабораторной работе
4. формулировка индивидуального задания
5. весь код решения индивидуального задания, разбитый на необходимые типы файлов
6. скриншоты выполнения индивидуального задания
7. выводы по лабораторной работе

В РАМКАХ ВСЕГО КУРСА ООПиП ВСЕ ЛАБОРАТОРНЫЕ РАБОТЫ ДОЛЖНЫ ХРАНИТЬСЯ В ОДНОМ РЕШЕНИИ (SOLUTION), В КОТОРОМ ДОЛЖНЫ БЫТЬ СОЗДАНЫ ОТДЕЛЬНЫЕ ПРОЕКТЫ (PROJECTS) ДЛЯ КАЖДОЙ ЛАБОРАТОРНОЙ РАБОТЫ. ВО ВСЕХ ПРОЕКТАХ ПОЛЬЗОВАТЕЛЬ ДОЛЖЕН САМ РЕШАТЬ ВЫЙТИ ИЗ ПРОГРАММЫ ИЛИ ПРОДОЛЖИТЬ ВВОД ДАННЫХ. В КАЖДОМ ЗАДАНИИ НЕОБХОДИМО СОЗДАТЬ ДИАГРАММУ КЛАССОВ НА ЯЗЫКЕ UML ДЛЯ ОТОБРАЖЕНИЯ ВСЕХ СОЗДАННЫХ КЛАССОВ И СВЯЗЕЙ МЕЖДУ НИМИ.

ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ К ЛАБОРАТОРНОЙ РАБОТЕ №2 (в каждом индивидуальном задании должны быть использованы smart-указатели и транзакции):

1. Разработать набор классов (минимум 5) по теме «издательство печатной продукции». Корректно реализовать связи между классами. Использовать *smart*-указатели для созданий программы учета деятельности издательства. Имеется несколько объектов печатной продукции и некоторое количество сотрудников издательства, которые работают с издаваемой продукцией. Печатная продукция считается готовой к изданию, когда с ней не работает ни один сотрудник издательства. В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Реализовать механизм транзакций, который позволит сотрудникам откатывать изменения, внесенные в печатную продукцию. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

2. Разработать набор классов (минимум 5) по теме «делопроизводство в учреждении образования». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета делопроизводства в учреждении образования. Имеется несколько объектов документации и некоторое количество сотрудников учреждения образования, которые работают с документацией. Документ считается сформированным, когда с ним не работает ни один сотрудник. В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Реализовать механизм транзакций, который позволит сотрудникам откатывать изменения, внесенные в документы. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

3. Разработать набор классов (минимум 5) по теме «учет сотрудников ИТ-компании». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета сведений о сотрудниках и расчета для них заработной платы. В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Реализовать механизм транзакций, который позволит откатывать изменения, внесенные в сведения о сотрудниках ИТ-организации. Предусмотреть возможность автоматического «отката» к предыдущему состоянию данных о сотруднике, если текущее состояние является неудовлетворительным. Смоделировать данную ситуацию. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

4. Разработать набор классов (минимум 5) по теме «использование текстового редактора пользователями с автоматическим сохранением данных в файл». Корректно реализовать связи между классами. Редактор должен поддерживать возможность отмены внесенных изменений (отменить ввод, *Undo*) и восстановления отмененной информации (вернуть ввод, *Redo*). Для хранения историй модификации файла использовать список (его рациональную организацию определить самостоятельно). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Использовать *smart*-указатели для создания программы, которая определяет возможность самостоятельного закрытия файла (т.е. данные файла сохранены и с ним никто сейчас не работает). Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

5. Разработать набор классов (минимум 5) по теме «банковские операции». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета проводимых банковский операций и подсчета количества различных видов (переводы денежных средств, оформление договора поручительства, выдача справки и т.д.) банковских операций и сумм по ним. Реализовать механизм транзакций, который позволит откатывать изменения, если операция выполнена неверно (например, произведена попытка списать со счета больше, чем на нем имеется, неверно указаны данные в договоре). Программа должна обеспечивать вывод информации о проведении банковских операций только по запросу клиента. В разработанном наборе классов должен быть хотя бы один шаблонный класс.

Все классы должны иметь методы получения и установки значений полей. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

6. Разработать набор классов (минимум 5) по теме «Розничная продажа товаров и услуг». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета продаваемых и закупаемых товаров. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о товаре введены неверно (например, цена имеет некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод информации об осуществленных покупках и итоговой сумме только по запросу клиента. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

7. Разработать набор классов (минимум 5) по теме «Тестирование знаний студентов», включающий обязательно следующие классы: «Тест» (название теста, тема теста, перечень вопросов, перечень полученных ответов), «Пользователь» (ФИО, факультет, номер группы), «Ответ» (дата выполнения теста, ФИО выполнившего тест). **Корректно реализовать связи между классами.** Использовать *smart*-указатели для создания программы учета полученных ответов на тесты. Реализовать механизм транзакций, который позволит откатывать изменения, внесенные в ответ. В разработанном наборе классов должен быть хотя бы один шаблонный класс. Программа должна обеспечивать вывод итоговой информации о выполнении тестов в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

8. Разработать набор классов (минимум 5) по теме «Разработка мобильных приложений». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета создаваемых мобильных приложений. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о продукте введены неверно (например, операционная система или язык разработки имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод и вывод информации обо всех разработанных мобильных приложениях, всех приложениях для конкретной платформы в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

9. Разработать набор классов (минимум 5) по теме «Деятельность библиотеки». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета книжной продукции в библиотеке. Имеются абонементы читателей и сотрудники библиотеки, которые работают с абонементами. Абонемент считается готовым к редактированию, когда с ним не работает ни один сотрудник. В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Реализовать механизм транзакций, который позволит сотрудникам откатывать изменения,

внесенные в документы. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

10. Разработать набор классов (минимум 5) по теме «Строительная компания». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета строительных объектов компании. Имеются строящиеся объекты и сотрудники компании, которые работают с данными строящихся объектов. Данные строительного объекта считаются готовыми к редактированию, когда с ним не работает ни один сотрудник. В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Реализовать механизм транзакций, который позволит сотрудникам откатывать изменения, внесенные в документы. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

11. Разработать набор классов (минимум 5) по теме «Розничный бизнес». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета продаваемых и закупаемых товаров. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о товаре введены неверно (например, наименование или количество имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод информации о покупках и итоговой сумме для клиента, обо всех продажах для менеджера в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

12. Разработать набор классов (минимум 5) по теме «Каталог мобильных телефонов». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета продаваемых и закупаемых мобильных телефонов. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о товаре введены неверно (например, цена или количество имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод информации о покупках и итоговой сумме для клиента, обо всех продажах для менеджера в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

13. Разработать набор классов (минимум 5) по теме «Деятельность кинотеатра». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета продаваемых и закупаемых билетов в кинотеатре. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о билетах введены неверно (например, цена или количество имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод информации о покупках и итоговой сумме для клиента, обо всех продажах для менеджера в табличном виде на экран и в файл.

Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

14. Разработать набор классов (минимум 5) по теме «Конвертор файлов». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета конверторов файлов. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о выбранном конверторе файлов введены неверно (например, формат или качество итогового файла имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод информации обо всех выполненных успешно и неуспешно конвертациях для отдельного пользователя и формата файлов. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

15. Разработать набор классов (минимум 5) по теме «Системы управления базами данных». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета используемых систем управления базами данных. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о выбранной системе управления базами данных указаны неверно (например, драйвер или порт имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод информации обо всех выполненных успешно и неуспешно соединениях с системами управления базами данных. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

16. Разработать набор классов (минимум 5) по теме «Графический редактор». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета созданных файлов в графическом редакторе. Реализовать механизм транзакций, который позволит откатывать изменения в редактируемом файле, если данные при его редактировании или сохранении указаны неверно (например, формат сохранения или директория имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод информации обо всех выполненных успешно и неуспешно конвертациях для отдельного пользователя и формата файлов. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

17. Разработать набор классов (минимум 5) по теме «Логистическая компания». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета перевозимых грузов. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о перевозимом грузе введены неверно (например, цена или маршрут имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна

обеспечивать вывод и ввод информации о реализованных перевозках и итоговой сумме транспортировки в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

18. Разработать набор классов (минимум 5) по теме «Беспилотные летательный аппараты». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета беспилотных летательных аппаратов. Реализовать механизм транзакций, который позволит откатывать изменения, если данные об аппарате введены неверно (например, позиция или маршрут имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод информации о всех контролируемых летательных аппаратах в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

19. Разработать набор классов (минимум 5) по теме «Машиностроительное предприятие». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета продаваемых и закупаемых деталей на машиностроительном предприятии. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о продукции введены неверно (например, цена или количество имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод детальной информации о созданной продукции, итоговой сумме в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

20. Разработать набор классов (минимум 5) по теме «Словарь английского языка». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета имеющихся переводов для слов и фраз. Реализовать механизм транзакций, который позволит откатывать изменения, внесенные в перевод. В разработанном наборе классов должен быть хотя бы один шаблонный класс. Программа должна обеспечивать вывод итоговой информации обо всех имеющихся языковых конструкциях в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

21. Разработать набор классов (минимум 5) по теме «Спортивные соревнования». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета проведенных и планируемых спортивных соревнований. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о соревновании введены неверно (например, дата или количество участников имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод детальной

информации о проводимых соревнованиях, итоговых результатах соревнований в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

22. Разработать набор классов (минимум 5) по теме «Высшее учебное заведение». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета обучения студентов в ВУЗе. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о студенте введены неверно (например, дата или оценка студента имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод детальной информации о студенте, итогах сессии для сотрудников деканата в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

23. Разработать набор классов (минимум 5) по теме «HR-менеджмент». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета, подбора, переподготовки сотрудников компании. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о сотруднике введены неверно (например, ФИО или требования к вакансии имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод подробной информации о сотрудниках в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

24. Разработать набор классов (минимум 5) по теме «Маркетинговые услуги». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета предлагаемых маркетинговых услуг. Реализовать механизм транзакций, который позволит откатывать изменения, если данные об услуге введены неверно (например, срок проведения или цена имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод информации обо всех проведенных маркетинговых услугах для одного заказчика, обо всех проведенных услугах для менеджера в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

25. Разработать набор классов (минимум 5) по теме «Система конвертации валют». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета проводимых операций конвертации валют. Реализовать механизм транзакций, который позволит откатывать изменения, если данные об операции введены неверно (например, сумма или валюта конвертации имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс.

Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод информации обо всех проведенных операциях конвертации для одного клиента, обо всех проведенных услугах для менеджера в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

26. Разработать набор классов (минимум 5) по теме «Очередь обслуживания клиентов банка». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета проводимых операций обслуживания клиентов банка. Реализовать механизм транзакций, который позволит откатывать изменения, если данные об операции введены неверно (например, сумма или валюта операции имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод информации обо всех проведенных банковских операциях для сотрудника банка, обо всех проведенных банковских операциях для менеджера в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

27. Разработать набор классов (минимум 5) по теме «Список сотрудников ИТ компании». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета принятых на работу и уволенных сотрудников ИТ-компании. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о сотруднике введены неверно (например, ФИО или дата приема имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод подробной информации о сотрудниках в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

28. Разработать набор классов (минимум 5) по теме «Разработка программного обеспечения». Корректно реализовать связи между классами. Использовать *smart*-указатели для учета созданного и сопровождаемого программного обеспечения. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о программном обеспечении введены неверно (например, язык программирования или операционная система имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод информации обо всех созданных и поддерживаемых программных продуктах в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

29. Разработать набор классов (минимум 5) по теме «Разработка компьютерных игр». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета созданных компьютерных игр. Реализовать механизм транзакций, который позволит

откатывать изменения, если данные об игре введены неверно (например, сеттинг или музыка имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод детальной информации о созданных компьютерных играх в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.

30. Разработать набор классов (минимум 5) по теме «Разработка 3D графики». Корректно реализовать связи между классами. Использовать *smart*-указатели для создания программы учета созданных и удаленных изображений *3D* графики. Реализовать механизм транзакций, который позволит откатывать изменения, если данные о графике введены неверно (например, материал или источник света имеют некорректное значение). В разработанном наборе классов должен быть хотя бы один шаблонный класс. Все классы должны иметь методы получения и установки значений полей. Программа должна обеспечивать вывод детальной информации о созданных *3D* изображениях в табличном виде на экран и в файл. Использовать конструктор с параметрами, конструктор без параметров, конструктор копирования, деструктор.