Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационных технологий и управления

Кафедра экономической информатики

ОТЧЕТ

по лабораторной работе

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

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| Студент гр. 124404 |  | Н.А Мойсенович |
| Проверила |  | К.А. Борщевич |

Минск 2021

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

**Краткие теоретические сведения по лабораторной работе.**

В языке С++ 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 С++ 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 С++ 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.

В template-инструкции можно определить несколько обобщенных типов данных, используя список элементов, разделенных запятой. Создавая шаблонную функцию компилятору разрешается генерировать столько различных версий этой функции, сколько необходимо для обработки различных способов её вызова.

Помимо создания явным образом перегруженных версий обобщенной функции, можно также перегружать саму спецификацию шаблона функции. 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.

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.

Интеллектуальный указатель (умный указатель, smart pointer) действует, как обычный указатель, но с важным дополнением: smartpointer 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 С++ 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 . 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.

**Формулировка индивидуального задания.**

Разработать набор классов (минимум 5) по теме «Беспилотные летательные аппараты». 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.

**Листинг**

Файл source.cpp

#include <iostream>

#include "SmartPointer.h"

#include "Drone.h"

#include "Transaction.h"

using namespace std;

void CreateDrone(Transaction<Drone<int>>\* transaction)

{

Drone<int> drone;

drone.Create();

(\*transaction).Begin(drone);

}

void ShowStates(Transaction<Drone<int>>\* transaction)

{

(\*transaction).Show();

}

void CommitChanges(Transaction<Drone<int>>\* transaction)

{

(\*transaction).Commit();

}

void UndoChanges(Transaction<Drone<int>>\* transaction)

{

(\*transaction).Undo();

}

void SaveChanges(Transaction<Drone<int>>\* transaction)

{

(\*transaction).Write();

}

void CallCertainMethod(int answer, Transaction<Drone<int>>\* transaction)

{

switch (answer)

{

default:

cout << "Choose correct point" << endl;

return;

break;

case 1:

CreateDrone(transaction);

break;

case 2:

ShowStates(transaction);

break;

case 3:

CommitChanges(transaction);

break;

case 4:

UndoChanges(transaction);

break;

case 5:

SaveChanges(transaction);

break;

}

}

void ShowMenu(Transaction<Drone<int>>\* transaction)

{

int answer;

cout << "Here are a menu, for drone managment" << endl;

cout << "1.Create a new drone" << endl;

cout << "2.Show drone's states" << endl;

cout << "3.Confirm changes" << endl;

cout << "4.Undo changes" << endl;

cout << "5.Write states of drone's" << endl;

cout << "Or write any other button/phrase to exit" << endl;

cin >> answer;

if (cin)

CallCertainMethod(answer, transaction);

else

{

cout << "Incorrect input, exit programm" << endl;

return;

}

}

int main()

{

Transaction<Drone<int>> transaction;

while (true)

{

ShowMenu(&transaction);

}

return 0;

}

Файл “Transaction.h”

#pragma once

#include<iostream>

#include"SmartPointer.h"

using namespace std;

template<typename T>

class Transaction

{

SmartPointer<T> \_previousState;

SmartPointer<T> \_currentState;

public:

Transaction();

Transaction(Transaction<T>& other);

Transaction<T>& operator=(Transaction const& other);

T\* operator->();

void Write();

void Show();

bool Begin(T newState);

void Commit();

bool Undo();

void SetPreviousState(SmartPointer<T>\* data);

void SetCurrentState(SmartPointer<T>\* data);

};

template<typename T>

inline Transaction<T>::Transaction()

{

\_previousState = SmartPointer<T>();

\_currentState = SmartPointer<T>();

}

template<typename T>

Transaction<T>::Transaction(Transaction<T>& other)

{

\*(other.\_previousState) = \*\_previousState;

\*(other.\_currentState) = \*\_currentState;

}

template<typename T>

inline T\* Transaction<T>::operator->()

{

return \_currentState.GetPointer();

}

template<typename T>

Transaction<T>& Transaction<T>:: operator=(Transaction<T> const& other)

{

\*(other.\_previousState) = \*\_previousState;

\*(other.\_currentState) = \*\_currentState;

return this;

}

template<typename T>

void Transaction<T>::Write()

{

(\*\_previousState).WriteInfo("Previous State");

(\*\_currentState).WriteInfo("Current State");

}

template<typename T>

void Transaction<T>::Show()

{

if (\_previousState.GetPointer())

{

cout << "Previous state: " << endl;

(\*\_previousState).ShowInfo();

}

else

cout << "Have no previous state" << endl;

cout << endl;

cout << "Current state :" << endl;

(\*\_currentState).ShowInfo();

cout << endl;

}

template<typename T>

bool Transaction<T>::Begin(T newState)

{

\_previousState = \_currentState;

\*\_currentState.GetPointer() = newState;

if (\_currentState.GetPointer() == nullptr)

return false;

return true;

}

template<typename T>

void Transaction<T>::Commit()

{

\_previousState.SetPointer(nullptr);

}

template<typename T>

bool Transaction<T>::Undo()

{

if (\_previousState.GetPointer() == nullptr)

return false;

\_currentState = \_previousState;

\_previousState.SetPointer(nullptr);

}

template<typename T>

void Transaction<T>::SetPreviousState(SmartPointer<T>\* previousState)

{

\*\_previousState = \*previousState;

}

template<typename T>

void Transaction<T>::SetCurrentState(SmartPointer<T>\* currentState)

{

\*\_currentState = \*currentState;

}

Файл “SmartPointer.h”

#pragma once

#include<iostream>

template<class T>

class SmartPointer

{

T\* \_pointer;

public:

SmartPointer();

SmartPointer(T pointerData);

SmartPointer(SmartPointer const& other);

~SmartPointer();

T\* GetPointer();

void SetPointer(T\* pointer);

SmartPointer<T>& operator=(SmartPointer<T> const& other);

T& operator\*();

};

template<class T>

SmartPointer<T>::SmartPointer()

{

\_pointer = new T();

}

template<class T>

inline SmartPointer<T>::SmartPointer(T pointerData)

{

\*\_pointer = pointerData;

}

template<class T>

SmartPointer<T>::SmartPointer(SmartPointer const& other)

{

\*(other->\_pointer) = \*\_pointer;

}

template<class T>

SmartPointer<T>::~SmartPointer()

{

delete \_pointer;

}

template<class T>

T\* SmartPointer<T>::GetPointer()

{

return \_pointer;

}

template<class T>

void SmartPointer<T>::SetPointer(T\* pointer)

{

\_pointer = pointer;

}

template<class T>

SmartPointer<T>& SmartPointer<T>:: operator=(SmartPointer<T> const& other)

{

if (\_pointer == nullptr)

\_pointer = new T();

\*\_pointer = \*(other.\_pointer);

return \*this;

}

template<typename T>

T& SmartPointer<T>::operator\*()

{

return \*\_pointer;

}

Файл “Drone.h”

#pragma once

#include <iostream>

#include <string>

#include <fstream>

using std::cin;

using std::cout;

using std::string;

using std::ofstream;

using std::endl;

enum Color

{

Nothing,

Red,

Yellow,

Magenta

};

class Framework

{

protected:

Color \_color;

};

class Engine

{

protected:

int \_enginePower;

};

class SpeedLimiter

{

protected:

int \_maxSpeed;

};

class Accumulator

{

protected:

int \_capacity;

};

template<typename T>

class Drone: Framework, Engine, SpeedLimiter, Accumulator

{

int \_version;

T \_customData;

string GetColor();

public:

Drone();

Drone(Color color, int enginePower, int maxSpeed, int capacity, int version, T customData);

void ShowInfo();

void WriteInfo(string stateName);

void Create();

};

template<typename T>

string Drone<T>::GetColor()

{

switch (\_color)

{

case Red:

return "Red";

break;

case Yellow:

return "Yellow";

break;

case Magenta:

return "Magenta";

break;

case Nothing:

return "Nothing";

break;

}

}

template<typename T>

Drone<T>::Drone()

{

\_version = int();

\_customData = T();

\_capacity = int();

\_maxSpeed = int();

\_enginePower = int();

\_color = Color(0);

}

template<typename T>

Drone<T>::Drone(Color color, int enginePower, int maxSpeed, int capacity, int version, T customData)

{

\_color = color;

\_enginePower = enginePower;

\_maxSpeed = maxSpeed;

\_capacity = capacity;

\_version = version;

\_customData = customData;

}

template<typename T>

void Drone<T>::ShowInfo()

{

cout << "Version of the drone: " << \_version << endl;

cout << "Custom data in drone: " << \_customData << endl;

cout << "Capacity of the accumulator: " << \_capacity << endl;

cout << "Max speed of the drone: " << \_maxSpeed << endl;

cout << "Drone engine power: " << \_enginePower << endl;

cout << "Drone color: " << GetColor() << endl;

}

template<typename T>

void Drone<T>::WriteInfo(string stateName)

{

ofstream fout;

fout.open("dataBase.txt", ofstream::app);

fout << stateName << '\n';

if (this == nullptr)

{

fout << "Have no previous state\n";

return;

}

fout << "Color: " << GetColor() << "\nVersion: " << \_version << "\nCustom data: " << \_customData << "\nAccumulator capacity: " << \_capacity << "\nDrone max speed: " << \_maxSpeed << "\nEngine power: " << \_enginePower << "\n\n";

}

template<typename T>

void Drone<T>::Create()

{

int colorID;

int version;

T customData;

int capacity;

int maxSpeed;

int enginePower;

cout << "Choose a color: " << endl;

cout << "1.Red" << endl;

cout << "2.Yellow" << endl;

cout << "3.Magenta" << endl;

cin >> colorID;

cout << "Enter a version of drone" << endl;

cin >> version;

cout << "Enter a custom data" << endl;

cin >> customData;

cout << "Enter an accumulator capacity " << endl;

cin >> capacity;

cout << "Enter drone max speed" << endl;

cin >> maxSpeed;

cout << "Enter engine power" << endl;

cin >> enginePower;

if (!(colorID > 0 && colorID <= 3) || !cin)

{

cout << "\nInput data incorrect, please try to input digit numbers or correct color" << endl;

cin.clear();

cin.ignore(INT\_MAX, '\n');

Create();

}

else

{

\_color = Color(colorID);

\_version = version;

\_customData = customData;

\_capacity = capacity;

\_maxSpeed = maxSpeed;

\_enginePower = enginePower;

}

}

**Вывод.** Научился использовать шаблоны функций и классов при работе со smart-указателями и транзакциями.