# **8. Лабораторная работа № 8. Изучение «умных» указателей**

**Часть 1**

Реализовать шаблон для задания «умного» указателя по аналогии шаблона **std::unique\_ptr.** В шаблоне предусмотреть конструктор, который получает «сырой» указатель, деструктор, метод get, возвращающий «сырой указатель», выполнить перегрузку операций \* и ->, запретить создание копий объектов, реализовать перемещение. Возможные заголовки методов шаблона кроме конструкторов и операторов присваивания копирования и перемещения приведены ниже:

template<class T>

class **MyUnique**

{

T \* p=nullptr;

public:

**MyUnique**(T \*p);

~**MyUnique**();

T \* **get**() const;

T & operator\*();

T \* operator->();

};

Для создания объекта типа **MyUnique** разработать глобальную шаблонную функцию Make\_MyUnique с переменным числом параметров, которая получает параметры как у конструктора объекта, на который указывает указатель. Теория по таким функциям приведена ниже в подразделе **Шаблоны с переменным числом параметров (*variadic template*).**

В функции main продемонстрировать все заданные возможности, в том числе, создать указатель на объект своего класса, имеющего конструктор с параметрами (например, класса MyPoint – точка на плоскости) c помощью функции Make\_MyUnique, получающей параметры как конструктора своего класса (MyPoint).

**Часть 2**

Аналогично разработать шаблон для задания «умного» указателя по аналогии шаблона **std::shared\_ptr,** который отличается от шаблона части 1 тем, что разрешает копирование. Назвать его можно MyShared. Разработать также глобальную шаблонную функцию Make\_MyShared, которая получает параметры как у конструктора объекта, на который указывает указатель. Продемонстрировать все возможности в функции main по аналогии с частью 1.

**Шаблоны с переменным числом параметров (*variadic template*)**

Пример рекурсивного раскрытия шаблона с переменным числом параметров:

#include <iostream>

using namespace std;

template<typename T>

T Sum(const T& x)

{

return x;

}

template<typename T, typename...Args>

T Sum(const T& x, const Args&... args)

{

return x + Sum(args...);

}

int main()

{

cout << Sum(1, 2, 3, 5, 6) << endl;

cout << Sum(1.4, 2.4, 3.7, 5.7) << endl;

return 0;

}

Другой способ:

#include <iostream>

using namespace std;

template<typename...Args>

auto Sum(const Args&... args)

{

return (args+...); // Требуется стандарт С++ 17, используется операция +

}

int main()

{

cout << Sum(1, 2, 3, 5, 6) << endl;

cout << Sum(1.4, 2.4, 3.7, 5.7) << endl;

return 0;

}

Напишем функцию очистки нескольких разных контейнеров, каждый из которых имеет метод clear(), это могут быть vector, string и другие. И функцию печати размеров контейнеров, каждый их которых имеет метод size().

#include <iostream>

#include <vector>

#include <string>

using namespace std;

// Очистка нескольких разных контейнеров

template<typename...Args>

void Clear(Args&... args)

{

(args.clear(),...); // Операция , (запятая)

}

// Печать размеров контейнеров

template<typename...Args>

void PrintSizes(const Args&... args)

{

((cout<<args.size()<<endl), ...); // Операция , (запятая)

}

int main()

{

vector<int> vec = { 1, 2, 3, 4 };

string str = "Hello World";

PrintSizes(vec, str); // Печать размеров до очистки

Clear(vec, str); // Очистка

PrintSizes(vec, str); // Печать размеров после очистки

return 0;

}

Применим подобный подход для шаблонной функции с переменным числом параметров, которая принимает параметры конструктора объекта, создает объект динамически и возвращает указатель на этот объект. Для простого класса MyPoint данный подход работает:

#include <iostream>

#include <string>

using namespace std;

// Простой класс - Точка на плоскости

class MyPoint

{

int x, y;

public:

MyPoint(int x, int y) : x(x), y(y)

{

cout << "MyPoint(int x, int y): x="<<x<<" y="<<y << endl;

}

};

// Функция которая создает объект, принимая параметры его конструктора

template<typename T, typename...Args>

T\* CreateObject(const Args& ... args)

{

return new T(args...); // Создаем объект, вызываем конструктор

}

int main()

{

auto p1 = CreateObject<MyPoint>(5, 10);

return 0;

}

Но если применить это напрямую к более сложному объекту, например, для имитатора файла (класс File), то при создании объекта в функции получим ошибку. Упрощённо, проблема заключается в том, что переменные аргументы шаблонной функции передаются по константной ссылке (const Args& ... args), а у класса есть конструкторы с разными видами ссылок: константная ссылка (const string& filename), просто ссылка (ostream& log) и ссылка на временный объект (File&& other). Возникает проблема приведения этих типов ссылок. Пример ниже:

#include <iostream>

#include <string>

using namespace std;

// Упрощенная обертка над файлом - имитатор

class File

{

ostream& log;

public:

File(const string& filename, ostream& log) : log(cout)

{

log << "Opening file: " << filename << endl;

}

File(const File&) = delete;

File(File&& other) noexcept: log(other.log)

{

log << "File(File&& other)" << endl;

}

};

// Функция которая создает объект, принимая параметры его конструктора

template<typename T, typename...Args>

T\* CreateObject(const Args& ... args)

{

return new T(args...); // Создаем объект, Здесь будут ошибки!!!!!

}

int main()

{

string s = "1.txt";

auto p1 = CreateObject<File>(s, cout);

File file2("2.txt", cout);

auto p2 = CreateObject<File>(move(file2));

return 0;

}

Для решения проблемы с преобразование разных типов ссылок необходимо вызвать специальную шаблонную функцию, std::forward, определенную в заголовочном файле utility, которая и осуществляет преобразования разных типов ссылок. Аргумент в функцию должен передаваться Args&& ... args, при вызове конструктора forward<Args>(args)... Пример ниже:

#include <iostream>

#include <string>

using namespace std;

// Упрощенная обертка над файлом - имитатор

class File

{

ostream& log;

public:

File(const string& filename, ostream& log) : log(cout)

{

log << "Opening file: " << filename << endl;

}

File(const File&) = delete;

File(File&& other) noexcept: log(other.log)

{

log << "File(File&& other)" << endl;

}

};

// Функция которая создает объект, принимая параметры его конструктора

template<typename T, typename...Args>

T\* CreateObject(Args&& ... args)

{

return new T(forward<Args>(args)...); // Создаем объект, вызываем конструктор

}

int main()

{

string s = "1.txt";

auto p1 = CreateObject<File>(s, cout);

File file2("2.txt", cout);

auto p2 = CreateObject<File>(move(file2));

return 0;

}

***Варианты заданий.***

В качестве класса, для которого необходимо реализовать умные указатели, необходимо взять класс в соответствии со своим вариантом из первой лабораторной.

***Вопросы для самоподготовки***

1. Умный указатель std::unique\_ptr, внутреннее устройство, пример использования.
2. Умный указатель std::shared\_ptr, внутреннее устройство, пример использования.
3. Умный указатель std::weak\_ptr, назначение, пример использования.