A\*. Geometry

|  |  |
| --- | --- |
| Ограничение времени | 1 секунда |
| Ограничение памяти | 64.0 Мб |
| Ввод | стандартный ввод |
| Вывод | стандартный вывод |

<https://gitlab.com/ibr11/cppmipt_spring2022/-/tree/master/geometry>

В этой задаче необходимо реализовать набор классов для решения геометрических задач на плоскости. Все координаты предполагаются целочисленными.

Vector

Реализовать класс Vector для вектора на плоскости с необходимыми арифметическими операциями (унарные и бинарные +/-, умножение/деление на скаляр, присваивающие версии операций, сравнение на равенство).

Shape

Создать набор классов-фигур, которые наследуются от абстрактного базового класса IShape для работы с двумерными геометрическими примитивами:

* Point (точка);
* Segment (отрезок);
* Line (линия);
* Ray (луч);
* Polygon (простой многоугольник - часть плоскости, ограниченная замкнутой ломаной без самопересечений);
* Circle (окружность).

В базовом классе IShape предусмотреть чисто виртуальные методы:

* Move(const Vector&) - сдвига на заданный вектор, метод должен изменять состояние объекта и возвращать ссылку на сам объект;
* ContainsPoint(const Point&) - проверка (true/false) содержит ли фигура (внутренность фигуры) точку;
* CrossesSegment(const Segment&) проверка (true/false) пересекается ли фигура (граница фигуры) с отрезком;
* Clone() - копирование объекта (необходимо вернуть умный или обычный указатель на копию фигуры);
* ToString() - строковое представление фигуры (формат см. в примерах).

В производных классах - реализовать эти методы. Разность двух точек должна возвращать вектор перемещения одной точки в другую.

Детали

Все классы должны располагаться в пространстве имен geometry.

С публичным интерфейсом классов, который используется в задаче, подробнее можно ознакомиться в файле geometry\_main.cpp. В частности, Point должен уметь конструироваться от двух целых чисел, Segment, Line и Ray - от двух Point, многоугольник - от std::vector<Point>, окружность - от центра (Point) и радиуса (int).

Объявления классов расположите в соответствующих .h файлах, а необходимые файлы реализации положите в папку src. После этого вы можете запустить geometry\_public\_test (по обычной инструкции - через терминал, либо с помощью IDE) и протестировать на различных входных данных. Вы можете прочитать, что принимает программа на вход из файла geometry\_main.cpp и найти примеры корректной работы на странице задачи в контесте.

**Где ошибка?** Тесты:

* 1-6 совпадают с примером
* 7-15 *Point*
* 16-46 *Segment*
* 47-55 *Ray*
* 56-60 *Line*
* 61-71 *Polygon*
* 72-80 *Circle*

Формат ввода

В первой строчке задается тип геометрического примитива: <<point>>, <<segment>>, <<ray>>, <<line>>, <<circle>> или <<polygon>>. Далее вводится сам примитив.

После чего вводится две точки A*A* и B*B*, которые используются в CheckFunctions*CheckFunctions*. Все числа целочисленные и не превосходят 10000 по модулю.

Для Point выводятся ее координаты; Segment - два конца (точки); Ray - начало (точка) и направляющий вектор; Line - коэффициенты уравнения прямой (ax + by + c) с точностью до некоторого множителя (по идее система должна принимать и Line(1, −1, 1), и Line(−2, 2, −2)); Polygon - последовательность вершин; Circle - центр и радиус.

Пример 1

| **Ввод**  Скопировать ввод | **Вывод**  Скопировать вывод |
| --- | --- |
| point  0 0  -1 -1 1 1 | Given shape does not contain point A  Given shape crosses segment AB  Point(2, 2) |

Пример 2

| **Ввод**  Скопировать ввод | **Вывод**  Скопировать вывод |
| --- | --- |
| segment  0 -1 0 1  0 0 0 1 | Given shape contains point A  Given shape crosses segment AB  Segment(Point(0, 0), Point(0, 2)) |

Пример 3

| **Ввод**  Скопировать ввод | **Вывод**  Скопировать вывод |
| --- | --- |
| ray  0 0 1 1  2 2 3 2 | Given shape contains point A  Given shape crosses segment AB  Ray(Point(1, 0), Vector(1, 1)) |

Пример 4

| **Ввод**  Скопировать ввод | **Вывод**  Скопировать вывод |
| --- | --- |
| line  0 0 0 1  1 1 2 1 | Given shape does not contain point A  Given shape does not cross segment AB  Line(1, 0, -1) |

Пример 5

| **Ввод**  Скопировать ввод | **Вывод**  Скопировать вывод |
| --- | --- |
| polygon  4  0 0 3 0 3 3 0 3  1 1 2 2 | Given shape contains point A  Given shape does not cross segment AB  Polygon(Point(1, 1), Point(4, 1), Point(4, 4), Point(1, 4)) |

Пример 6

| **Ввод**  Скопировать ввод | **Вывод**  Скопировать вывод |
| --- | --- |
| circle  0 0 5  0 0 5 5 | Given shape contains point A  Given shape crosses segment AB  Circle(Point(5, 5), 5) |

# B\*. UniquePtr

|  |  |
| --- | --- |
| Ограничение времени | 1 секунда |
| Ограничение памяти | 64.0 Мб |
| Ввод | стандартный ввод или input.txt |
| Вывод | стандартный вывод или output.txt |

<https://gitlab.com/ibr11/cppmipt_spring2022/-/tree/main/unique_ptr>

В качестве решения ожидается zip архив с решением (даже если решение состоит из одного файла). Присылайте файлы только с ВАШИМ кодом, то есть файлы с тестами и CMakeLists.txt присылать не нужно.

### Условие

С++ предоставляет шаблонный класс std::unique\_ptr ([*https://en.cppreference.com/w/cpp/memory/unique\_ptr*](https://en.cppreference.com/w/cpp/memory/unique_ptr)) для безопасной работы с динамически выделенной памятью (и другими ресурсами). В std::unique\_ptr перегружены все необходимые операции для комфортной работы с объектами так, как если бы это были обычные указатели (которые еще память за собой очищают). Объекты этого класса считаются единственными владельцами ресурса, на который они указывают, поэтому эти объекты нельзя копировать, но можно перемещать - передавать владение другому объекту. Использование std::unique\_ptr почти не накладывает дополнительных вычислительных расходов, поэтому пользоваться им так же эффективно, как и обычными указателями, и при этом гораздо безопаснее.

Пример:

#include <memory>

// ...

std::unique\_ptr<int> ptr(new int(10));

std::cout << \*ptr << '\n'; // 10

// std::unique\_ptr<int> copy = ptr; копирование запрещено

std::unique\_ptr<int> moved = std::move(ptr); // теперь ptr пуст

std::cout << \*moved << '\n'; // 10

auto rational = std::make\_unique<Rational>(1, 2);

// эквивалентно std::unique\_ptr<Rational> rational(new Rational(1, 2));

std::cout << rational->Numerator() << ' ' << rational->Denominator() << '\n'; // 1 2

// delete вызовутся автоматически в деструкторах!

### Задание

Реализуйте шаблон UniquePtr - упрощенный аналог класса умного указателя с уникальным владением, std::unique\_ptr (C++11).

### Детали реализации

Шаблонный класс должен поддерживать:

Конструктор по умолчанию (создает нулевой указатель).

Конструктор от указателя (сохраняет указатель на объект).

Конструктор копирования и копирующее присваивание должны отсутствовать.

Перемещающий конструктор и перемещающее присваивание должны передавать владение объектом.

Метод Release(), который отлучает класс от владения текущим ресурсом и возвращает указатель на него.

Метод Reset(T\* ptr = nullptr), меняет указатель, которым владеет объект (старый ресурс удаляется).

Метод Swap(UniquePtr<T>&).

Метод Get(), возвращающий указатель на объект.

Оператор разыменовывания operator\*.

Оператор "стрелочка" operator->.

Явный оператор приведения к bool (operator bool).

### Замечания.

<https://en.cppreference.com/w/cpp/memory/unique_ptr>

Решение должно состоять из одного файла unique\_ptr.h

### Дополнительное задание

Дополнительно можете реализовать внешнюю шаблонную функцию MakeUnique, принимающую произвольное число параметров и возвращающую UniquePtr на объект созданный с помощью данных параметров конструктора. В этом случае добавьте в файл с решением директиву

#define MAKE\_UNIQUE\_IMPLEMENTED.

# Выбрать C. SharedPtr

|  |  |
| --- | --- |
| Ограничение времени | 1 секунда |
| Ограничение памяти | 64.0 Мб |
| Ввод | стандартный ввод или input.txt |
| Вывод | стандартный вывод или output.txt |

<https://gitlab.com/ibr11/cppmipt_spring2022/-/tree/main/shared_ptr>

Умные указатели реализуют важную идиому C++ - RAII (Resource Acquisition Is Initialization) и позволяют не заботиться о ручном управлении памятью, так как инкапсулируют управление ресурсами посредством конструкторов и деструкторов.

#### std::shared\_ptr (C++11)

Шаблонный класс std::shared\_ptr предоставляет интерфейс указателя на данные в динамической области с автоматическим контролем своевременного выделения и удаления ресурсов. В отличие от std::unique\_ptr, std::shared\_ptr реализует семантику разделяемого владения ресурсом, что означает, что у одного ресурса может быть несколько равноправных владельцев. При этом гарантируется, что ресурс не будет освобожден пока существует хотя бы один его владелец (хотя бы один объект std::shared\_ptr указывает на ресурс). Для решения проблемы циклических ссылок в язык также введены "слабые указатели" std::weak\_ptr, которые не считаются полноценными владельцами ресурса, но из них при необходимости можно получить "сильный указатель" std::shared\_ptr на объект, если тот еще не удален.

Несмотря на то, что std::shared\_ptr удобнее в использовании чем std::unique\_ptr, на практике последний работает гораздо эффективней - std::shared\_ptr реализует дополнительную логику с подсчетом ссылок на объект. Поэтому std::shared\_ptr стоит использовать только тогда, когда вам действительно нужно разделяемое владение.

### Задание

Реализуйте шаблон SharedPtr - упрощенный аналог класса умного указателя с разделяемым владением.

Идея реализации: помимо самого указателя на выделенный ресурс необходимо знать число "сильных" и "слабых" ссылок на объект. Для этого в куче выделим специальную счетчик strong\_counter, который будет хранить эту информацию (при создании нового указателя соответствующий счетчик увеличивается, при удалении - уменьшается; если число сильных ссылок стало равно 0, то объект удаляется).

### Детали реализации

Детали обсуждались на лекции и семинарах. Шаблонный класс SharedPtr должен поддерживать:

Конструктор по умолчанию (создает нулевой указатель).

Конструктор от указателя (сохраняет указатель на владеемый объект).

Конструктор копирования и копирующее присваивание (создают новую ссылку на тот же объект).

Перемещающий конструктор и перемещающее присваивание должны передавать владение объектом.

Метод Reset(T\* ptr = nullptr), меняет указатель, которым владеет объект.

Метод Swap(SharedPtr<T>&).

Метод Get(), возвращающий указатель на владеемый объект.

Метод UseCount(), возвращающий число "сильных" ссылок на объект.

Оператор разыменовывания operator\*.

Оператор "стрелочка" operator->.

Явный оператор приведения к bool (operator bool).

### Замечания.

<https://en.cppreference.com/w/cpp/memory/shared_ptr>

Решение должно состоять из одного файла shared\_ptr.h

### Дополнительное задание

Реализуйте шаблон WeakPtr - аналог std::weak\_ptr, а также функцию MakeShared - аналог std::make\_shared. Для поддержки "слабых" ссылок (см. выше) предлагается вынести счетчик обычных (сильных) ссылок strong\_count в отдельную структуру Counter, в которую дополнительно добавить поле weak\_count для подсчета количества слабых указателей, указывающих на данный объект. Как и ранее, сам счетчик (объект Counter) должен находиться в динамической памяти (вспомните почему), а указатель на него храниться в соответствующих объектах SharedPtr и WeakPtr. Логика освобождения ресурса теперь следующая:

Если число сильных и слабых ссылок стало равно нулю, то удаляется и объект, и выделенный для него счетчик Counter.

Если число сильных стало равно нулю при ненулевом числе слабых, то удаляется только объект, счетчик остается жить (то есть все сильные указатели уже "умерли", а некоторые слабые все еще ссылаются на данный счетчик).

Шаблон WeakPtr состоит из:

Аналогичных конструкторов (по умолчанию, копирования, перемещения) и операторов присваивания.

Конструктора от SharedPtr (увеличивает число "слабых" ссылок)

Метода Swap(WeakPtr<T>&)

Метода Reset(), отвязывающего указатель от объекта

Метода UseCount(), возвращающего число "сильных" ссылок на объект

Метод Expired(), возвращающий true, если сильных ссылок на объект уже нет (объект удален)

Метод Lock(), возвращающий SharedPtr на объект (если Expired() == true, то возвращается пустой указатель)

В SharedPtr необходимо добавить конструктор от WeakPtr, который работает аналогично методу Lock(), но в случае Expired() == true должно бросаться исключение BadWeakPtr.

Дополнительно необходимо реализовать внешнюю шаблонную функцию MakeShared, принимающую произвольное число параметров и возвращающую SharedPtr на объект созданный с помощью данных параметров конструктора. (Важно: в задании подойдет любая наивная реализация. На практике std::make\_shared позволяет выделить счетчики и объект единым блоком в памяти, что часто повышает эффективность программы).

### Замечания.

<https://en.cppreference.com/w/cpp/memory/weak_ptr>

Для проверки этой части добавьте в файл с решением (shared\_ptr.h) директиву

#define WEAK\_PTR\_IMPLEMENTED