Семинар #2: Классы и перегрузка операторов. Домашнее задание.

Задача 1. Класс `Circle`

Допустим, что мы хотим создать программу, которая будет работать с окружностями (это может быть игра или, например, графический редактор). Для того, чтобы сделать код более понятным и удобным в использовании, мы решили создать класс окружности. Кроме того, мы решили использовать уже ранее написанный класс точки в 2D пространстве (файл `point.hpp`). Создайте класс окружности, который будет включать следующие поля:

- Поле `center` типа `Point` – центр окружности.
- Поле `radius` типа `float` – радиус окружности.

И следующие методы:

- Конструктор `Circle(const Point& center, float radius)`, который будет задавать поля `center` и `radius` соответствующими значениями.
- Конструктор по умолчанию `Circle()` – задаются значения, соответствующие единичной окружности с центром в начале координат.
- Конструктор копирования `Circle(const Circle& circle)`
- Сеттеры и геттеры, для полей `center` и `radius`. Поле `radius` нельзя задать отрицательным числом. При попытке задания его отрицательным числом оно должно устанавливаться в значение `0`.
- Метод `float area() const`, который будет возвращать площадь поверхности круга.
- Метод `float distance(const Point& p) const`, который будет возвращать расстояние от точки `p`, до ближайшей точки окружности.
- Метод `bool isColliding(const Circle& c) const`, который будет возвращать `true`, если круг пересекается с кругом `c`.
- Метод `void move(const Point& p)`, который будет перемещать кружок на вектор `p`.

Весь начальный код содержится в папке `seminar2_encapsulation/homework/code/circle`.

Задача 2. Математический вектор

В папке `seminar2_encapsulation/homework/code/complex` лежит реализация комплексного числа с перегруженными операторами. Используйте её в качестве примера для решения этой задачи.

- Создайте класс `Vector2f` – вектор в двумерном пространстве с полями `x` и `y` типа `float` в качестве координат. Перегрузите следующие операторы для работы с вектором. Для передачи вектора в функции используте ссылки и, там где возможно, модификатор `const`.
  - Сложение векторов (`+`)
  - Вычитание (`-`)
  - Умножение вектора на число типа `float` (число `*` вектор и вектор `*` число)
  - Скалярное произведение (`*`)
  - Унарный `-`
  - Унарный `+`
  - Проверка на равенство `==` (должна возвращать тип `bool`)
  - Проверка на неравенство `!=` (должна возвращать тип `bool`)
  - Операторы `+=` и `-=` (вектор `+=` вектор)
  - Операторы `*=` (вектор `*=` число)

- Оператор вывода `ostream >>` вектор. Выводите вектор в виде `(x, y)`.
- Оператор ввода `istream <<` вектор

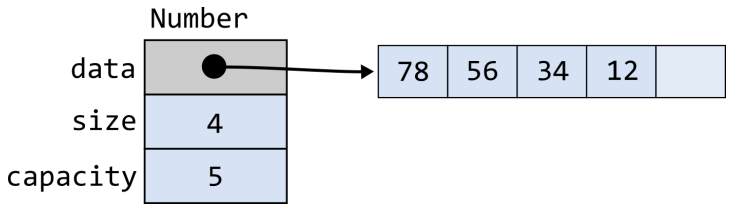
- Протестируйте ваши функции:

```
#include <iostream>
#include "vector2f.hpp"
using std::cout, std::endl;

int main()
{
    Vector2f a = {1.0, 2.0};
    Vector2f b = {4.0, -1.0};
    cout << "a = " << a << endl << "b = " << b << endl;
    cout << "a + b = " << a + b << endl;
    cout << "-a = " << -a << endl;
    cout << "Scalar product of a and b = " << a * b << endl;
    a /= 5;
    cout << "a after a /= 5;"  << a << endl
    a += b;
    cout << "a after a+= b;" << a << endl;
}
```

Задача 3. Большие числа

Стандартные целочисленные типы данных, такие как `int` имеют фиксированный небольшой размер. Соответственно значения, которые можно хранить в переменных этих типов ограничены. Тип `int` обычно ограничен $2^{31} - 1 = 2147483647$ и даже тип `unsigned long long` имеет ограничение обычно равное $2^{64} - 1 \approx 1.8 * 10^{19}$. Хранить действительно большие числа в переменных этих типов невозможно. В этом задании нужно создать класс, с помощью которого можно будет удобно складывать и умножать большие целые положительные числа. Начальный код этого класса содержится в папке `seminar2_encapsulation/homework/code/number`.



Представление числа 12345678 в памяти с помощью нашего класса Number

Подзадачи:

1. **Конструктор по умолчанию:** Напишите конструктор по умолчанию `Number()`, который будет создавать число равное нулю.

2. **Конструктор копирования:** Напишите конструктор копирования `Number(const Number& n)`.

3. **Конструктор из строки:** Напишите конструктор `Number(const std::string& str)`, который будет создавать большое число на основе строки. Предполагаем, что на вход конструктору всегда идёт корректная строка. Например, число из примера можно будет создать так:

   ```
   Number a("12345678");
   ```

   или так:

   ```
   Number a = "12345678";
   ```

4. **Присваивание:** Напишите оператор присваивания `Number& operator=(const Number& n)`.

5. **Сложение:** Напишите и протестируйте операторы сложения `operator+` и оператор присваивания сложения `operator+=`. Реализовывать оба этих оператора с нуля необязательно. Ведь, если написан один из этих операторов, то очень просто написать другой.

6. **Числа Фибоначчи:** Числа Фибоначчи задаются следующим образом:

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

   Используйте класс `Number`, чтобы вычислить F_{1000}. Правильный ответ:

```
F(1000) = 43466557686937456435688527675040625802564660517371780402481729089536555417949051890
40387984007925516929592259308032263477520968962323987332247116164299644090653318793829896964992
85160037044761377951668492288 75
```

7. **Четность:** Напишите метод `bool isEven() const`, который будет проверять является ли наше число чётным и, если это верно, возвращает `true`, в ином случае возвращает `false`.

8. **Произведение:** Напишите метод `Number operator*(const Number& n) const` - оператор умножения одного числа `Number` на другое. Протестируйте вашу функцию на различных примерах (умножение большого числа на большое, умножение большого числа на небольшое, умножение двух небольших чисел и т. д.).

9. **Факториал:** Используйте написанный оператор для вычисления факториала от 1000.
   Правильный ответ:

```
1000! = 40238726007709377354370243392300398571937486421071463254379991042993851239862902059
20442084869694048004799886101971960586316668729948085589013238296699445909974245040870737 59
91882362772718873251977950595099527612087497546249704360141827809464649629105639388743788 64
87337119181045825783647849977012476632889835955735432513185323958463075557409114262417474 34
93475534286465766116677973966688202912073791438537195882498081268678383745597317461360853 79
53452422158659320192809087829730843139284440328123155861103697680135730421616874760967587 13
48312025478589320767169132448426236131412508780208000261683151027341827977704784636586170 16
43650241536913982812648102130927612448963599287051149649754199093422215668325720808213331 86
11681155361583654698404670897560290095053761647584772842188967964624494516076535340819890 13
85442487984959953319101723355556602139450399736280750137837615307127761926849034352625200 01
58885351473316117021039681759215109077880193931781141945452572238655414610628921879602238 38
97147608850627686296714667469756291123408243920816015378088989396451826324367161676217916 89
09779911903754031274622289988005195444414282012187361745992642956581746628302955570299024 32
41531816172104658320367869061172601587835207515162842255402651704833042261439742869330616 90
89796848259012545832716822645806652676995865268227280707578139185817888965220816434834482 59
93266043367660176999612831860788386150279465955131156552036093988180612138558600301435694 52
72242063446317974605946825731037900840244324384656572450144028218852524709351906209290231 36
49327349756551395872055965422874977401141334696271542284586237738753823048386568897646192 73
83814900140767310446640259899490222221765904339901886018566526485061799702356193897017860 04
08118897299183110211712298459016419210688843871218556461249607987229085192968193723886426 14
83965738229112312502418664935314397013742853192664987533721894069428143411852015801412334 48
28015051399694290153483077644569099073152433278288269864602789864321139083506217095002597 38
98635542771967428222487575867657523442202075736305694988250879689281627538488633969099598 26
28095612145099487170124451646126037902930912088908694202851064018215439945715680594187274 89
98094254742173582401063677404595741785160829230135358081840096996372524230560855903700624 27
12434169090041536901059339838357779394109700277534720000000000000000000000000000000000000 00
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 00
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000 00
00000000000000000000000000
```

10. **Большие числа с использованием вектора:** Напишите новый класс `Integer`, который будет делать всё то же самое, что и `Number`, но не будет самостоятельно выделять память в куче. Вместо этого для хранения разрядов числа класс `Integer` будет использовать внутри себя класс `std::vector`. Этот класс должен находиться в новых файлах `integer.hpp` и `integer.cpp`.

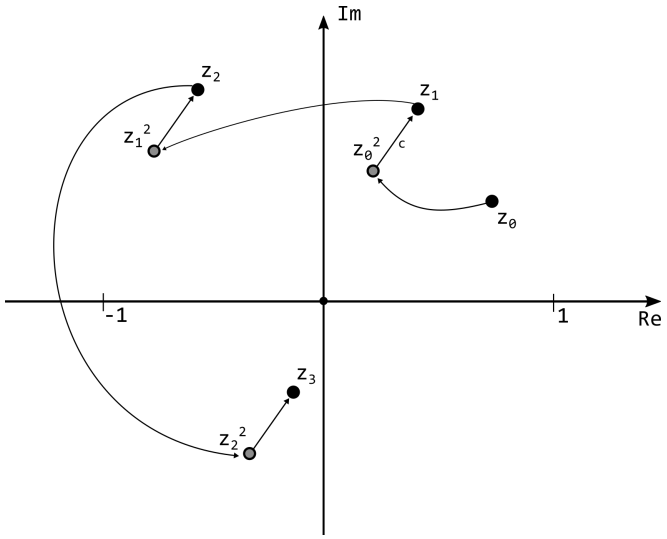
Необязательные задачи (никак не учитываются)

Задача 1. Убегающая точка

Предположим, что у нас есть комплексная функция $f(z) = z^2$. Выберем некоторое комплексное число z_0 и будем проводить следующие итерации:

$$z_1 = f(z_0) \quad z_2 = f(z_1) \quad ... \quad z_{k+1} = f(z_k) \quad ...$$

В зависимости от выбора точки z_0 эта последовательность либо разойдётся, либо останется в некоторой ограниченной области. Будем называть точку z_0 убегающей, если $z_k \to \infty$ при $k \to \infty$. Область неубегания для функции z^2, т.е. множество всех начальных значений z_0, при которых последовательность (1) остаётся ограниченной, тривиальна – это единичный круг. Но всё становится сложнее для функции вида $f(z) = z^2 + c$, где c – некоторое комплексное число. Для такой функции найти область неубегания можно только численно.



1. **Множество Жюлиа**
   Численно найдите область неубегания для функций вида $f(z) = z^2 + c$, где c – некоторое комплексное число. Для этого создайте изображение размера 800x800, покрывающую область `[-2:2]x[-2:2]` на комплексной плоскости. Для каждой точки этой плоскости проведите $N \approx 20$ итераций и, в зависимости от результата, окрасьте пиксель в соответствующий цвет (цвет можно подобрать самим, он должен быть пропорционален значению z_N - меняться от яркого если z_N мало и до черного если z_N большое). Нарисуте изображения для $c = -0.4 + 0.6i$; $c = -0.70 - 0.38i$; $c = -0.80 + 0.16i$ и $c = 0.280 + 0.011i$.

   Программа должна создавать изображения в формате `.ppm`. Для работы с изображениями в формате `.ppm` используйте класс `Image` из файла `code/fractal/image.hpp`. Для просмотра изображений в формате `.ppm` можно использовать программу IrfanView. Для комплексных чисел используйте класс `Complex` из файла `code/fractal/complex.hpp`. Начальный код для задачи лежит в файле `code/fractal/complex_image.cpp`.

2. Добавьте параметры командной строки: 2 вещественных числа, соответствующие комплексному числу c, и целое число итераций N.

3. **Множество Мандельброта**
   Зафиксируем теперь $z_0 = 0$ и будем менять c. Численно найдите все параметры c, для которых точка z_0 не является убегающей. Для этого создайте изображение размера 800x800, покрывающую область `[-2:2]x[-2:2]` возможных значений c на комплексной плоскости. Программа должна создавать файл `mandelbrot.ppm`.

4. **Анимация**
   Программа `complex_movie.cpp` создаёт множество изображений и сохраняет их в папку `animation` (если у вас нет такой папки – создайте её). Эти изображения представляют собой отдельные кадры будущей анимации. Чтобы их объединить в одно видео можно использовать программу ffmpeg. (ffmpeg можно установить с помощью MSYS2. Или можно скачать тут: www.ffmpeg.org и изменить переменную среды `PATH` в настройках системы.) После этого можно будет объединить все изображения в одно видео такой командой:

```
$ ffmpeg -r 60 -i animation/complex_%03d.ppm complex_movie.mp4
```

Создайте анимацию из изображений множеств Julia при c линейно меняющемся от $(-1.5 - 0.5i)$ до i.