САМАРСКИЙ УНИВЕРСИТЕТ
SAMARA UNIVERSITY

Институт информатики, математики и электроники
Кафедра геоинформатики и информационной безопасности

Технологии и методы программирования

Лабораторная работа № 1

Введение в объектно-ориентированное программирование

Последнее обновление: 14 сентября 2018

1 БАЗОВАЯ ЛИТЕРАТУРА

[B. Eckel] Философия C++. *Главы 1–5.*

[S. B. Lippman, J. Lajoie, B. E. Moo] C++. Базовый курс. *Глава 7.*

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1 Понятия объекта и класса

Центральными понятиями в объектно-ориентированном программировании (ООП) являются понятия объекта и класса. Для иллюстрации этих понятий приведём следующие примеры:

- класс – «фигура», примеры объектов: квадрат с вершинами в точках (0, 0) и (1, 1); прямоугольник с вершинами в точках (0, 0) и (4, 1).
- класс – «функция одного вещественного аргумента», примеры объектов: $\sin(x)$; $3x^2+2x+1$; e^x.

В программировании класс – это описание собственного типа данных; а объект – конкретный экземпляр класса. Описание класса состоит из описания данных (полей), которые смогут хранить экземпляры класса (т. е. объекты) и описания набора действий (методов), которые над этими данными могут производиться (другими словами, это операции, которые можно совершить над объектом). Например, тип `int` предназначен для хранения целых чисел и подразумевает набор арифметических операций над ними (сложение, вычитание, умножение, деление и т. д.).

Поля описывают то, какие данные смогут хранить экземпляры. Конкретные значения сохраняются уже внутри объектов. К полям внутри класса можно обращаться непосредственно по именам полей. В C++ имена полей рекомендуется начинать с символа нижнего подчёркивания (либо с префикса `m_`, либо выделять каким-либо другим образом), чтобы по имени переменной можно было сразу сказать, является ли эта переменная полем класса или обычной локальной переменной.

Методы – это функции, которые могут применяться к экземплярам класса. Они объявляются внутри класса и предназначены для работы с его объектами. Метод можно вызвать только у конкретного экземпляра класса (объекта).

Рассмотрим пример. Опишем класс комплексных чисел и снабдим его методом вычисления модуля комплексного числа. В теле программы создадим два объекта, соответствующих комплексным числам $1+i$ и $3+4i$ соответственно. Выведем на экран модули этих чисел.

```cpp
#include <cmath>

#include <cstdio>

// Описание класса комплексных чисел (объявление нового типа данных Complex)
class Complex {
public:
    // Поля класса (вещественная и мнимая части)
    double _re;
    double _im;

    // Метод класса (вычисление модуля комплексного числа)
    double abs() {
        return std::sqrt(_re * _re + _im * _im);
    }
};

int main() {
    // Создаём первый экземпляр класса: 1 + i
    Complex c1;
    c1._re = 1;
    c1._im = 1;
    // Создаём второй экземпляр класса: 3 + 4 * i
    Complex c2;
    c2._re = 3;
    c2._im = 4;
    // Выводим их модули
    printf("%.2f\n", c1.abs()); // Выведет 1.41
    printf("%.2f\n", c2.abs()); // Выведет 5.00
}
```

2.2 Инкапсуляция

Одним из ключевых принципов ООП является инкапсуляция. Инкапсуляция – это принцип, согласно которому любой класс должен рассматриваться как чёрный ящик – пользователь класса должен видеть и использовать только интерфейсную часть класса (т. е. часть, предназначенную для использования внешним кодом) и не вникать в его внутреннюю реализацию. Поэтому данные принято инкапсулировать в классе таким образом, чтобы доступ к ним по чтению или записи осуществлялся не напрямую, а с помощью методов. Принцип инкапсуляции позволяет минимизировать число связей между классами и, соответственно, упростить независимую реализацию и модификацию классов.

По уровню доступа все члены класса делятся на открытые (публичные: `public`), закрытые (приватные: `private`) и защищённые (`protected`). Закрытые члены доступны только внутри класса. Открытые члены доступны как внутри, так вне класса. Защищённый уровень доступа будет рассмотрен в последующих лабораторных работах.

Перед объявлением членов внутри класса ставятся соответствующие ключевые слова. Если такое слово не поставлено, то считается, что член объявлен с уровнем `private`. В приведённом выше примере оба поля и метод являются публичными. Именно поэтому мы можем обращаться к ним в функции `main`. Заметим, что приведённый пример нарушает прин-

цип инкапсуляции (это было сделано для простоты примера), так как позволяет обращаться к данным класса напрямую, а не через методы чтения-записи. Пример концептуально более правильного кода будет приведён в подразделе 2.7.

2.3 Константные методы

Указание модификатора `const` при объявлении переменных гарантирует, что объект не будет изменён. Для объектов классов дело обстоит немного сложнее в связи с тем, что компилятору необходимо запрещать не только изменение данных объекта, но и вызовы тех методов, в которых эти данные могут быть изменены. Для того чтобы отличать те методы, которые не меняют данные объекта от остальных методов, используют всё тот же модификатор `const`. Этот модификатор указывается после списка параметров. Заметим, что в приведённом классе комплексных чисел метод вычисления модуля `abs()` не изменяет состояние объекта, и потому может быть сделан константным:

```
class Complex {
    ...

    // Константный метод вычисления модуля
    double abs() const {
        return std::sqrt(_re * _re + _im * _im);
    }

    ...
};
```

На реализацию константных методов класса накладывается ряд очевидных ограничений: такие методы не могут изменять значения полей объекта и вызывать неконстантные методы класса. При попытке вызова неконстантного метода для константного объекта должна быть сгенерирована ошибка компиляции.

2.4 Конструкторы

Конструкторы – это специальные методы класса, предназначенные для инициализации данных объекта. Конструкторы имеют то же имя, что и сам класс, не имеют никакого выходного значения. Для класса допустимо определить несколько конструкторов с различными параметрами. В этом случае говорят о перегрузке конструкторов. Пример перегрузки конструкторов для приведённого класса комплексных чисел:

```
class Complex {

    ...

    // Конструктор по умолчанию (без параметров)
    Complex() {
        _re = 0;
        _im = 0;
    }

    // Конструктор с параметрами
    Complex(double re, double im) {
        _re = re;
        _im = im;
    }

    ...
}
```

Если в классе не объявлен ни один конструктор, то компилятором будет сгенерирован конструктор по умолчанию (без параметров). Сгенерированный конструктор вызывает конструкторы по умолчанию для всех полей составного типа, но никак не инициализирует поля фундаментального типа (`int`, `float` и т. д.) и указатели. Непроинициализированные поля имеют произвольное значение, что приводит к ошибкам в процессе работы программы.

Конструктор, принимающий по ссылке константный объект текущего класса, называется конструктором копирования. Конструктор копирования генерируется компилятором всегда (независимо от наличия или отсутствия в классе других конструкторов), если только класс не предоставляет собственное явное определение конструктора копирования. Генерируемая компилятором версия по умолчанию копирует все поля объекта **по значению**:

```
class Complex {
    ...

    // Конструктор копирования. В данном случае его можно было вообще не определять,
    // т. к. сгенерированная компилятором версия по умолчанию делала бы то же самое.
    Complex(const Complex& rhs) {
        _re = rhs._re;
        _im = rhs._im;
    }

    ...
}
```

Конструкторы, принимающие только один параметр какого-либо типа T, могут использоваться для неявного преобразования из типа T в объект текущего класса:

```
class Complex {
    ...

    Complex(double re) {
        _re = re;
        _im = 0;
    }

    ...
}

int main() {
    // Происходит неявное преобразование типа int в комплексное число.
    // Эта строчка полностью эквивалентна выражению Complex c(10);
    Complex c = 10;
    ...
}
```

Обычно неявное преобразование типов приносит больше проблем, чем пользы. Для того, чтобы конструктор, принимающий только один параметр, не мог использоваться для неявного преобразования типов, можно указать ключевое слово `explicit` («явный»):

```
class Complex {
    ...

    explicit Complex(double re) {
        _re = re;
        _im = 0;
    }

    ...
}

int main() {
    // Теперь эта строчка не скомпилируется:
    Complex c = 10;
    // А эта по-прежнему корректна:
    Complex c(10);
    ...
}
```

2.5 Списки инициализации в конструкторах

При создании объектов, до того, как управление будет передано конструктору, поля составных типов инициализируются конструкторами по умолчанию. Пусть есть некий класс `Storage`, который внутри себя хранит объект составного типа `Complex`:

```
class Storage {
    Complex _complex;

public:
    Storage(const Complex& complex) {
        _complex = complex;
    }

    ...
};
```

До того, как выполнится строчка `_complex = complex;` поле `_complex` уже будет проинициализировано конструктором по умолчанию класса `Complex`. Такое поведение (ненужная инициализация конструктором по умолчанию) в общем случае приводит к ненужным дополнительным затратам времени и памяти. Решением является использование списков инициализации – специальной конструкции вида « `: _field1(value1), field2(value2), ...`», указываемой после списка параметров конструктора:

```
class Storage {
    Complex _complex;

public:
    Storage(const Complex& complex) : _complex(complex) {
        // Пустое тело конструктора.
        // В общем случае здесь может остаться код какой-либо сложной инициализации,
        // которую нетривиально сделать через список инициализации.
    }
};
```

Такой подход является практичным (экономит память и время при инициализации сложных типов) и рекомендуемым. В таком списке инициализации объект сложного типа сразу создаётся и инициализируется. Это позволяет инициализировать сложные типы без вызова их конструкторов по умолчанию, что невозможно при инициализации внутри тела конструктора, когда объекты составных типов уже созданы этими самыми конструкторами по умолчанию.

Важным моментом является то, что **при использовании списков инициализации инициализация полей выполняется в том порядке, в котором поля перечислены в самом классе, а не в том порядке, в котором они указаны в списке инициализации**. Приведём пример неправильного использования списка инициализации:

```
struct FailExample {
    int _a, _b;

    // НЕКОРРЕКТНОЕ ИСПОЛЬЗОВАНИЕ СПИСКА ИНИЦИАЛИЗАЦИИ
    // ВАЖНО: Инициализация полей будет идти в том порядке,
    //        в котором они объявлены в самом классе: сначала _a, потом _b.
    // Таким образом, в приведённом ниже примере:
    // 1. Сначала будет проинициализировано поле _a значением поля _b.
    //    Так как поле _b ещё не проинициализировано,
    //    то в _a запишется произвольное значение.
    // 2. Поле _b проинициализируется значением value.
    // В итоге выполнения такого конструктора поле _a останется непроинициализированным.
    FailExample(int value) : _b(value), _a(_b) { }
};
```

Для избежания таких ошибок строго рекомендуется перечислять поля в списке инициализации в том же порядке, в котором они перечислены в самом классе.

2.6 Темы для самостоятельного изучения

- статические поля и методы;
- ключевое слово `mutable` и область его применения.

2.7 Пример

Приведём пример класса комплексных чисел из подраздела 2.1, переписанный с учётом принципа инкапсуляции и использованием константных методов, конструкторов и списков инициализации. Также дополним класс методами вычисления суммы и разности двух комплексных чисел:

```
#include <cmath>
#include <cstdio>

class Complex {
    // Закрытые (приватные) поля
    double _re;
    double _im;

public:
    // Конструктор по умолчанию
    Complex() : _re(0), _im(0) { }

    // Конструктор с параметрами
    Complex(double re, double im) : _re(re), _im(im) { }

    // Константный метод получения вещественной части
    double getRe() const {
        return _re;
    }

    // Константный метод получения мнимой части
    double getIm() const {
        return _im;
    }
```

```
    // Константный метод вычисления модуля
    double abs() const {
        return std::sqrt(_re * _re + _im * _im);
    }

    // Константный метод вычисления суммы комплексных чисел
    Complex sum(const Complex& rhs) const {
    // Используем конструктор с параметрами для создания нового объекта
        return Complex(_re + rhs._re, _im + rhs._im);
    }

    // Константный метод вычисления разности комплексных чисел
    Complex dif(const Complex& rhs) const {
        // Используем конструктор с параметрами для создания нового объекта
        return Complex(_re - rhs._re, _im - rhs._im);
    }

    // Метод установки нового значения вещественной части
    void setRe(double re) {
        _re = re;
    }

    // Метод установки нового значения мнимой части
    void setIm(double im) {
        _im = im;
    }
};

int main() {
    // Создаём первый экземпляр класса (0), используя конструктор по умолчанию.
    Complex c1;

    // Присваиваем первому экземпляру новые значения (1 + i).
    // Мы можем вызвать неконстантные методы setRe и setIm, так как объект c1 неконстантный.
    c1.setRe(1);
    c1.setIm(1);

    // Создаём второй константный экземпляр класса (3 + 4 * i),
    // используя конструктор с параметрами.
    // Для константного объекта мы не сможем вызвать неконстантные методы setRe и setIm.
    const Complex c2(3, 4);

    // Создаём третий константный экземпляр класса, как сумму c1 и c2.
    // При этом используется неявно сгенерированный конструктор копирования.
    const Complex c3(c1.sum(c2));

    // Создаём четвёртый константный экземпляр класса, как разность c1 и c2.
    // При этом используется неявно сгенерированный конструктор копирования.
    const Complex c4(c1.dif(c2));

    // Выводим модули созданных комплексных чисел.
    // Константный метод abs можно вызвать как для неконстантного объекта,
    // так и для константного.
    printf("%.2f\n", c1.abs()); // Выведет 1.41
    printf("%.2f\n", c2.abs()); // Выведет 5.00
    printf("%.2f\n", c3.abs()); // Выведет 6.40
    printf("%.2f\n", c4.abs()); // Выведет 3.61
}
```

3 ТРЕБОВАНИЯ К ПРОЕКТИРОВАНИЮ И СТИЛЮ КОДА

Код лабораторной работы должен удовлетворять всем перечисленным ниже рекомендациям. Отклонения от приведённых рекомендаций допускаются только в случае, если вы сможете обосновать использованный подход.

Будут приведены только номера и названия рекомендаций из соответствующих книг. Подробные объяснения с примерами и исключениями из правил см. в самих книгах.

3.1 [H. Sutter, A. Alexandrescu] Стандарты программирования на C++

- 1. Компилируйте без замечаний при максимальном уровне предупреждений. *Следует серьёзно относиться к предупреждениям компилятора и использовать максимальный уровень вывода предупреждений вашим компилятором. Компиляция должна выполняться без каких-либо предупреждений. Вы должны понимать все выдаваемые предупреждения и устранять их путём изменения кода, а не снижения уровня вывода предупреждений.*
- 5. Один объект – одна задача. *Концентрируйтесь одновременно только на одной проблеме. Каждый объект (переменная, класс, функция, пространство имён, модуль, библиотека) должны решать одну точно поставленную задачу. С ростом объектов, естественно, увеличивается область их ответственности, но они не должны отклоняться от своего предназначения.*
- 6. Главное – корректность, простота и ясность. *Корректность лучше быстроты. Простота лучше сложности. Ясность лучше хитроумия. Безопасность лучше ненадёжности.*
- 10. Минимизируйте глобальные и совместно используемые данные. *Совместное использование вызывает споры и раздоры. Избегайте совместного использования данных, в особенности глобальных данных. Совместно используемые данные усиливают связность, что приводит к снижению сопровождаемости, а зачастую и производительности.*
- 11. Сокрытие информации. *Не выпускайте внутреннюю информацию за пределы объекта, обеспечивающего абстракцию.*
- 15. Активно используйте `const`. `const` – *ваш друг: неизменяемые значения проще понимать и отслеживать. Там, где это целесообразно, лучше использовать константы вместо переменных. Сделайте const описанием по умолчанию при определении значения – это безопасно, проверяемо во время компиляции и интегрируемо с системой типов C++.*

- 17. Избегайте магических чисел. *Избегайте использования в коде литеральных констант наподобие* `42` *или* `3.1415926`. *Такие константы не самоочевидны и усложняют сопровождение кода, поскольку вносят в него трудноопределимый вид дублирования. Используйте вместо них символьные имена и выражения наподобие* `width * aspectRatio`.
- 18. Объявляйте переменные как можно локальнее. *Избегайте «раздувания» областей видимости. Переменных должно быть как можно меньше, а время их жизни – как можно короче. Эта рекомендация по сути является частным случаем рекомендации 10.*
- 19. Всегда инициализируйте переменные. *Неинициализированные переменные – распространённый источник ошибок в программах на C и C++. Избегайте их, выработав привычку очищать память перед её использованием; инициализируйте переменные при их определении.*
- 20. Избегайте длинных функций и глубокой вложенности. *Краткость – сестра таланта. Чересчур длинные функции и чрезмерно вложенные блоки кода зачастую препятствуют реализации принципа «одна функция – одна задача» (см. рекомендацию 5), и обычно эта проблема решается лучшим разделением задачи на отдельные части.*
- 25. Передача параметров по значению, (интеллектуальному) указателю или ссылке. *Вы должны чётко уяснить разницу между входными, выходными параметрами и параметрами, предназначенными и для ввода, и для вывода информации, а также между передачей параметров по значению и по ссылке, и корректно их использовать.*
- 40. Избегайте возможностей неявного преобразования типов. *Не все изменения прогрессивны: неявные преобразования зачастую приносят больше вреда, чем пользы. Дважды подумайте перед тем, как предоставить возможность неявного преобразования к типу и из типа, который вы определяете, и предпочитайте полагаться на явные преобразования (используйте конструкторы, объявленные как* `explicit`*, и именованные функции преобразования типов).*
- 41. Делайте данные-члены закрытыми (кроме случая агрегатов в стиле структур C). *Данные-члены должны быть закрыты. Только в случае простейших типов в стиле структур языка C, объединяющих в единое целое набор значений, не претендующих на инкапсуляцию и не обеспечивающих поведение, делайте все данные-члены открытыми. Избегайте смешивания открытых и закрытых данных, что практически всегда говорит о бестолковом дизайне.*

- 47. Определяйте и инициализируйте переменные-члены в одном порядке. *Переменные члены всегда инициализируются в том порядке, в котором они объявлены при определении класса; порядок из упоминания в списке инициализации конструктора игнорируется. Убедитесь, что в коде конструктора указан тот же порядок, что и в определении класса.*
- 48. В конструкторах предпочитайте инициализацию присваиванию. *В конструкторах использование инициализации вместо присваивания для установки значений переменных-членов предохраняет от ненужной работы времени выполнения при том же объёме вводимого исходного текста.*

3.2 [S. Meyers] Эффективное использование C++

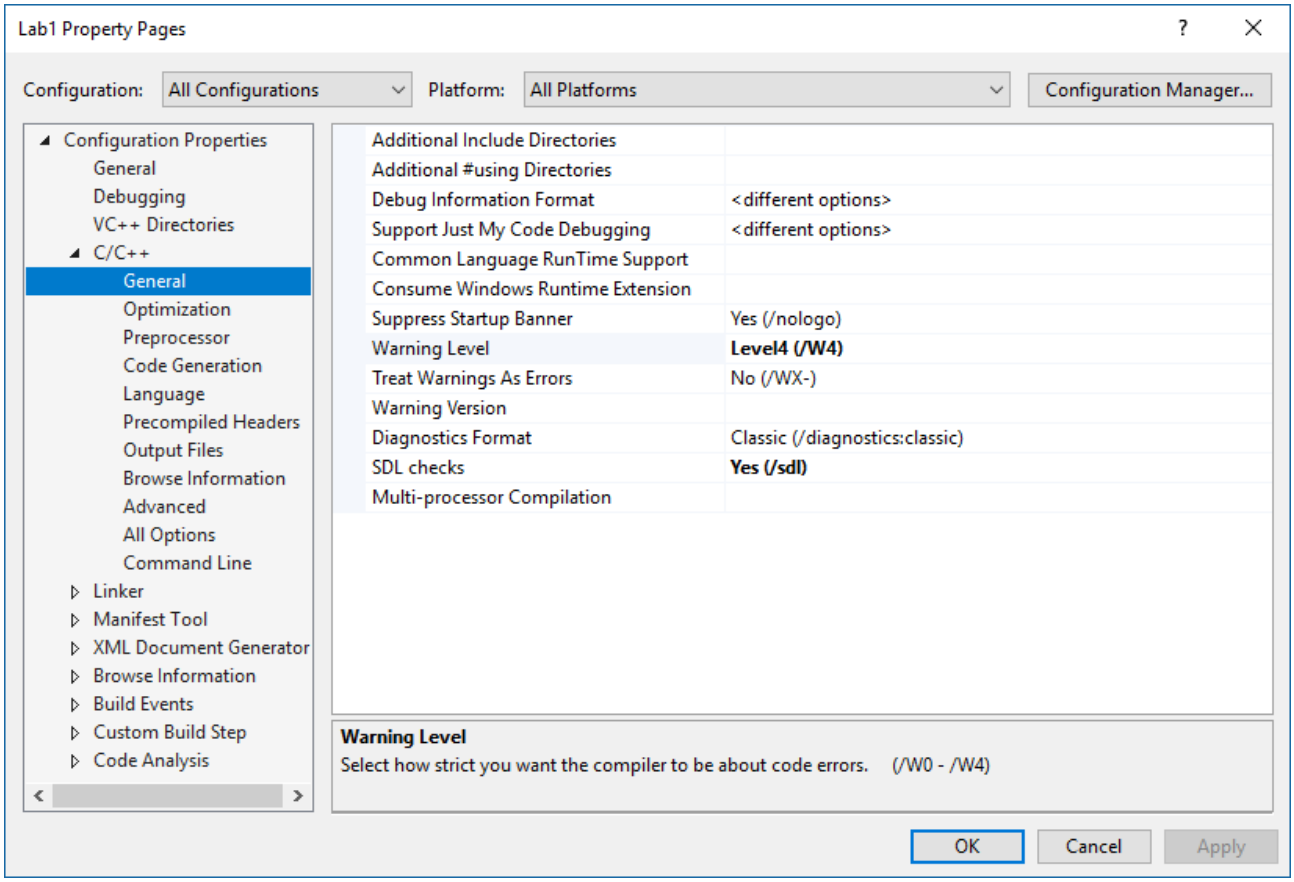
- 2. Предпочитайте `const`, `enum` и `inline` использованию `#define`.
- 3. Везде, где только можно, используйте `const`.
- 4. Прежде чем использовать объекты, убедитесь, что они инициализированы.
- 20. Предпочитайте передачу по ссылке на `const` передаче по значению.
- 22. Объявляйте данные-члены закрытыми.
- 26. Откладывайте определение переменных насколько возможно.
- 53. Обращайте внимание на предупреждения компилятора.

3.3 [S. Meyers] Эффективный и современный C++

- 3.1. Различие между `{}` и `()` при создании объектов.

3.4 Включение опций контроля качества кода в Microsoft Visual Studio

Откройте свойства проекта (пункт меню «Project» ⟶ «Properties»). В появившемся окне слева выберите пункт «C/C++» ⟶ «General». Выставите свойству «Warning Level» значение «Level4 (/W4)».



Более подробную информацию по диагностике кода в Microsoft Visual Studio можно посмотреть по следующему адресу:

https://docs.microsoft.com/en-us/visualstudio/code-quality/code-analysis-for-c-cpp-overview

4 ВАРИАНТЫ ЗАДАНИЙ

Общие требования

В начале программы вывести задание; в процессе работы выводить подсказки пользователю (что ему нужно ввести, чтобы продолжить выполнение программы). Взаимодействие с пользователем организовать в виде простого меню, обеспечивающего возможность переопределения начальных данных и выполнения различных операций (предусмотренных вариантом задания) и завершение работы программы. Предусмотреть контроль вводимых пользователем данных.

Код должен удовлетворять всем требованиям, приведённым в разделе 3.

ВАРИАНТ 1

Класс рациональных чисел.

Любой объект этого класса должен представлять собой несократимую дробь. Для приведения дроби к несократимому виду использовать алгоритм Евклида нахождения наибольшего общего делителя (НОД):

1) большее число делим на меньшее;
2) если делится без остатка, то меньшее число и есть НОД (следует выйти из цикла);
3) если есть остаток, то большее число заменяем на остаток от деления;
4) переходим к пункту 1.

Реализовать следующие методы:

- сложение двух рациональных чисел;
- вычитание двух рациональных чисел;
- умножение двух рациональных чисел;
- деление двух рациональных чисел;
- конвертирование рационального числа в приближённое вещественное число типа `double`.

ВАРИАНТ 2

Класс интервала с учётом включения/исключения концов.

Реализовать следующие методы:

- вычисление пересечения двух интервалов;
- вычисление объединения двух интервалов

Для простоты считать, что интервалы, не имеющие общих точек, пересекаться/объединяться не могут (результат в таком случае должен равняться пустому интервалу $(0;0)$ с исключенными концами).

ВАРИАНТ 3

Класс матриц $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ **размерности** 2×2**.**

Реализовать следующие методы:

- сложение двух матриц;
- вычитание двух матриц;
- умножение двух матриц;
- умножение матрицы на вещественное число;
- вычисление определителя матрицы.

ВАРИАНТ 4

Класс квадратичных функций $f(x)=ax^2+bx+c$**.**

Реализовать следующие методы:

- сложение двух функций;
- вычитание двух функций;
- вычисление производной функции;
- вычисление значения функции в указанной точке.

ВАРИАНТ 5

Класс прямых $Ax+By+C=0$ **на плоскости.**

Реализовать следующие методы:

- вычисление угла между прямой и осью абсцисс (угол отсчитывать против часовой стрелки от положительного направления оси абсцисс);
- вычисление угла между двумя прямыми;
- вычисление значения логического выражения «две указанные точки лежат по разные стороны от прямой»;
- определение точки пересечения двух прямых (если прямые не пересекаются, возвращать точку (NaN, NaN)).

Необходимо описать и использовать вспомогательный класс точек (x, y) на плоскости для реализации последних методов.

ВАРИАНТ 6

Класс треугольников на плоскости, заданных своими вершинами (x_1, y_1)**,** (x_2, y_2) **и** (x_3, y_3)**.**

Необходимо описать и использовать вспомогательный класс точек (x, y) на плоскости.

Реализовать следующие методы:

- вычисление периметра треугольника;
- вычисление площади треугольника по формуле Герона;
- вычисление значения логического выражения «указанная точка находится внутри треугольника».

ВАРИАНТ 7

Класс сфер $(x-x_C)^2+(y-y_C)^2+(z-z_C)^2=r^2$ **в пространстве.**

Необходимо описать и использовать вспомогательный класс точек (x, y, z) в пространстве.

Реализовать следующие методы:

- вычисление площади сферы $S=4\pi r^2$;
- вычисление объёма сферы $V=\frac{4}{3}\pi r^3$;
- вычисление значения логического выражения «указанная точка лежит внутри сферы»;
- вычисления значения логического выражения «две указанные сферы имеют общие точки».

ВАРИАНТ 8

Класс материальных точек с равноускоренным движением по прямой.

Модель равноускоренного движения по прямой может быть описана следующим выражением:

$$x(t)=\frac{at^2}{2}+vt+x_0.$$

Реализовать следующие методы:

- вычисление координаты точки в заданный момент времени;
- вычисление момента времени совпадения двух точек (если точки никогда не пересекутся, вернуть значение NaN);
- вычисление координаты совпадения двух точек (если точки никогда не пересекутся, вернуть значение NaN).

ВАРИАНТ 9

Класс резисторов с постоянным значением электрического сопротивления.

Реализовать следующие методы:

- создание резистора, эквивалентного параллельному соединению двух указанных резисторов: $R = \frac{R_1 R_2}{R_1 + R_2}$;
- создание резистора, эквивалентного последовательному соединению двух указанных резисторов: $R = R_1 + R_2$;
- вычисление силы тока, текущего через резистор, при указанном напряжении: $I = \frac{U}{R}$;
- вычисление напряжения на резисторе при указанной силе тока: $U = IR$.

ВАРИАНТ 10

Класс значений угла, заданного в градусах, минутах, секундах.

Реализовать следующий функционал, с учетом целых оборотов и ограничений на значения градусов (0...359), минуты и секунды (0...59):

- Сложение двух углов
- Вычитание двух углов
- Умножение угла на вещественное число
- Вычисление обратного угла (дополнение до 360)
- Преобразование в радианы

ВАРИАНТ 11

Класс для работы с денежными суммами.

Рубли и копейки хранятся в двух отдельных целочисленных полях данных. Реализовать следующий функционал:

- Сложение двух сумм
- Вычитание двух сумм
- Умножение суммы на вещественное число
- Сравнение сумм
- Вывод на экран суммы в формате xxxxxxxx руб. xx коп.
- (*) Вывод на экран суммы прописью.

ВАРИАНТ 12

Класс точка на плоскости

Хранятся декартовы координаты. Реализовать следующий функционал:

- Перемещение по оси X
- Перемещение по оси X
- Определение расстояния до начала координат
- Определение расстояния до произвольной точки
- Преобразование в полярные координаты и из полярных координат
- Проверка на равенство

ВАРИАНТ 13

Класс дата, в формате год, месяц, день.

Реализовать следующий функционал:

- Конструирование объектов по дате, числам и строке
- Добавление к дате указанного числа дней
- Вычитание от даты указанного числа дней
- Вычисление разницы в днях между двумя датами
- Определение високосного года.
- Сравнение дат

ВАРИАНТ 14

Класс прямоугольник, заданный двумя точками

Хранятся декартовы координаты левого нижнего и правого верхнего угла. Реализовать следующий функционал:

- Расширение прямоугольника для включения точки с координатами
- Расширение прямоугольника для включения другого прямоугольника
- Проверка попадает ли заданная точка в прямоугольник
- Проверка пересекает ли другой прямоугольник
- Создать новый прямоугольник, который является пересечением с другим прямоугольником
- Проверка на равенство

Необходимо описать и использовать вспомогательный класс точек (x, y) на плоскости для реализации последних методов.

ВАРИАНТ 15

Модель работы принтера

Хранится максимальный объём лотка (в единицах листов), загруженное количество листов, а также текущий объём чернил (в условных единицах). Также задана вещественная константа расхода чернил на 1 лист. Реализовать следующий функционал:

- Добавить в лоток заданное количество листов, если не будет переполнения лотка
- Распечатать заданное количество листов, пока достаточно листов для печати и достаточно чернил, вернуть количество лисов, которые удалось распечатать
- Проверка можно ли распечатать заданное количество листов
- Очистить лоток (извлечь всю бумагу)
- Установить максимальный объем чернил (сменить картридж)
- Создать копию модели принтера (копия экземпляра класса)