Криворожский государственный педагогический институт
Кафедра информатики и прикладной математики

А.П. Полищук
С.А. Семериков

КОМПЛЕКСНЫЕ ЧИСЛА, ВЕКТОРЫ И ПОЛИНОМЫ В КЛАССАХ ЯЗЫКА C++

Кривой Рог

1998

Полищук А.П., Семериков С.А. Комплексные числа, векторы и полиномы в классах языка C++. – Кривой Рог: КГПИ, 1998. – 72 с.

Авторы:

Полищук А.П.	к. т. н., с. н. с., доцент кафедры информатики и прикладной математики.
Семериков С.А.	магистр математики.

Рецензенты:

Хрисанов В.А.	к. ф.-м. н., доцент кафедры информатики и прикладной математики
Рашевский Н.А.	к. ф.-м. н., доцент кафедры математики

Под общей редакцией доктора физико-математических наук, профессора В.Н. Соловьёва.

Рекомендовано к печати на заседании кафедры информатики КГПИ, протокол №1 от 31.08.98 г.

Підп. до друку 02.09.98
Друк №3. Друк офсетний
Умовн. фарбо-відб. 3,45
Тираж 300

Формат 80x84 1/16.
Умовн. друк. арк. 3,6
Зам. №8-3876

КДПІ, 324086, Кривий Ріг-86, пр. Гагаріна, 54

Криворізька міська друкарня
324050, Кривий Ріг-50, пр. Металургів, 28.

Оглавление

1. Введение

1.1. Приближенные вычисления

Методы вычислений делят на *аналитические* и алгоритмические (*приближенные*). Аналитические методы вычислений используются в тех очень редких на практике случаях, когда входные и выходные данные удается связать между собой точным аналитическим выражением (формулами). В подавляющем большинстве практических задач сделать это не удается - либо из-за отсутствия математической модели, связывающей требуемые результаты с исходными данными, либо из-за чрезмерной сложности этой модели, либо сами исходные данные представлены в форме, исключающей возможность прямых аналитических вычислений.

Приведем простой пример: в распоряжении работника, подсчитывающего размер оплаты предприятием за израсходованный в производстве природный газ, имеется диаграммная лента с записью расхода газа в кубометрах в час за расчетный период. Если бы нарисованная самопишущим прибором на ленте кривая была представлена аналитической функцией времени, осталось бы проинтегрировать ее на заданном интервале и вычислить полученное аналитическое выражение. Но в силу случайного характера изменений расхода газа во времени представить имеющуюся зависимость расхода от времени в аналитической форме невозможно и остается использовать алгоритмы приближенного интегрирования.

К такому же выводу придем каждый раз при попытке решения уравнений, содержащих тригонометрические функции вперемежку с алгебраическими - даже простейший случай уравнения с одним неизвестным

$$a \cdot x + b \cdot \cos(c \cdot x) = 0$$

не позволяет получить аналитические выражения для вычисления корней и вынуждает нас использовать приближенные численные методы.

1.2. Численные методы и программирование

Вычисления сопровождают те виды интеллектуальной деятельности людей, в которых необходима точная численная оценка

результата, и математика - это не столько «упражнение для ума», как считал генералиссимус Суворов, сколько наука, обеспечивающая инженеров, ученых, финансистов, военных, фермеров, метеорологов и многих других методической основой для выполнения вычислений, по возможности - с помощью компьютерных программ с удобным интерфейсом пользователя.

Современная прикладная математика неразрывно связана со спецификой машинных вычислений и представляет собой сплав математики и компьютерного программирования. Разработчик технологии вычислений должен учитывать неизбежную при компьютерных вычислениях замену непрерывных задач их дискретными аналогами, и не только представить процесс обработки данных в виде вычислительной схемы с обоснованием существования и единственности решения, но и позаботиться об эффективных способах ввода и хранения исходных данных, их внутримашинной организации, об эффективности реализующей программы (времени счета и погрешностях вычислений), о возможностях наглядного (например, графического) представления полученных результатов, проанализировать возможность достижения заданной пользователем точности вычислений с учетом представления чисел в конкретной инструментальной среде. Иными словами, прикладной математик сегодня неизбежно и программист, владеющий современной технологией программирования. Результат разработки вычислительного алгоритма должен быть представлен в виде работоспособной, надежно функционирующей, хорошо документированной, переносимой программы с удобным пользовательским интерфейсом и разнообразными возможностями анализа и интерпретации результатов.

Методика преподавания численных методов и соответствующие учебные пособия медленно эволюционируют от классических курсов, ориентированных на ручные вычисления, в направлении объединения вычислительных алгоритмов с их программной реализацией на ЭВМ.

Среди книг, объединяющих изложение вычислительных алгоритмов с их реализацией в компьютерных программах, следует отметить «Справочник алгоритмов на языке АЛГОЛ. Линейная алгебра» Уилкинсона и Райнша (М.: Машиностроение, 1976), охватывающий широкий класс алгоритмов решения систем линейных алгебраических уравнений, псевдообращения матриц, вычисления

собственных значений и собственных векторов с исследованием области применения, подробным описанием вычислительных процедур и их АЛГОЛ-программ с оценкой точности, результатами тестирования, внимательным отношением к проблемам экономии памяти и скорости сходимости алгоритмов.

Язык ФОРТРАН представлен наиболее обширным списком пособий по численным методам:

- Плис А.И., Сливина Н.А. Лабораторный практикум по высшей математике. - М.: Высшая школа, 1983.
- Шуп Т. Решение инженерных задач на ЭВМ. Практическое руководство. - М.: Мир, 1982.
- Мак-Кракен Д., Дорн У. Численные методы и программирование на Фортране. - М.: Мир, 1977.
- Форсайт Дж., Малькольм М., Моулер К. Машинные методы математических вычислений. - М.: Мир, 1980.

Вычислительные алгоритмы с реализацией на языке БЕЙСИК приведены в «Справочнике по алгоритмам и программам на языке Бейсик для персональных ЭВМ» В.П. Дьяконова (М.: Наука, 1987) и в книге Я.Т. Гринчишина, В.И. Ефимова, А.Н. Ломаковича «Алгоритмы и программы на Бейсике» (М.: Просвещение, 1988).

Изложение алгоритмов вычислений с программами на языке ПАСКАЛЬ приведены в книге А.Е. Мудрова «Численные методы для ПЭВМ на языках БЕЙСИК, ФОРТРАН И ПАСКАЛЬ» (Томск: Раско, 1991).

Все известные авторам пособия по численным методам и их программной реализации базируются на процедурной методологии программирования и либо на отмирающих языках программирования (АЛГОЛ, ФОРТРАН), либо на учебных языках типа БЕЙСИК, ПАСКАЛЬ. Создание и использование в учебном процессе непрофессиональных учебных языков следует отнести к «детскому возрасту» компьютерного программирования, когда эта работа представлялась непостижимо сложной и недоступной для понимания большинством пользователей. Сегодня мы уже знаем, что для создания эффективных компьютерных программ недостаточно владения лексикой алгоритмического языка - необходим еще минимум знаний по архитектуре аппаратного обеспечения используемого компьютера, возможностях операционной системы и взаимодействию с ней

языкового транслятора; набор этих знаний вполне доступен для освоения пользователю средних способностей, а использование учебных суррогатов алгоритмических языков (особенно на национальным языковом базисе типа известного русскоязычного школьного учебного языка) приносит больше вреда, чем пользы. Да собственно и такой первоначально учебный язык как ПАСКАЛЬ, пытаясь в эволюции своего развития достичь возможностей простого, лаконичного профессионального языка Си, перестал быть учебным, хотя и не стал профессиональным по удобству и возможностям, сохранив, тем не менее «учебное» многословие и громоздкость конструкций.

Численные методы - основа инженерного и научного программирования, решающего задачи создания и исследования математических моделей процессов, протекающих в объектах различной природы: финансовых потоков и ресурсов в экономике страны или отдельного предприятия, воздушных потоков в атмосфере, изменения напряжений в строительных конструкциях под действием внешних сил, траекторий движения летательных аппаратов, движения электронов на атомных орбитах под воздействием внешних электромагнитных полей, взаимодействия популяций видов в растительном и животном мире и т. п. Математические модели различных процессов принято представлять в общем случае в виде систем интегро-дифференциальных уравнений с обыкновенными или частными производными; разработка методов и программ для решения таких уравнений представляет собой основной комплекс задач инженерного и научного программирования, а, следовательно, и численного анализа в целом.

При решении дифференциальных уравнений приходится осуществлять различные операции с такими математическими объектами, как матрицы (в общем случае с комплексными элементами), числовые или функциональные векторы, полиномы (с вещественными или комплексными коэффициентами).

В стройном здании математики более сложные математические объекты строятся из более простых. Так, комплексные числа представляются в виде пары вещественных координат вектора по вещественной и мнимой осям комплексной плоскости. Многомерный числовой вектор может быть представлен совокупностью его вещественных или комплексных координат по осям многомерной

координатной системы (или как частный случай матрицы - одностолбцовой или однострочной), матрица может быть представлена как вектор векторов, полином известного порядка может быть задан как вектор его коэффициентов.

Для каждого класса математических объектов определен допустимый набор математических операций и способы их реализации; например, операции умножения определены для вещественных чисел, векторов и матриц, но имеют, естественно, различный смысл и алгоритмы реализации. Операция определения нулей специфична для полиномов, транспонирования и вычисления собственных значений и собственных векторов - для матриц, определения модуля - для векторов.

Представляется целесообразным построить курс вычислительных методов по принципу определения иерархии математических классов, объекты которых конструировались бы затем в программе путем объявления, то есть синтаксически так же, как и стандартные для используемого языка типы (целые, вещественные и пр.) с определением внутри класса всех необходимых для их использования в вычислениях операций. При этом под термином «операция» можно понимать как общепринятые для простых типов операции, например, арифметические, так и любые, базирующиеся на данном математическом классе вычисления, - например, решение системы линейных алгебраических уравнений или вычисление коэффициентов регрессии для заданной матрицы или вычисление корней полинома наряду с операциями полиномиальной арифметики - сложения, умножения, деления полиномов.

В больших алгоритмических языках (ADA, PL) предусматриваются сложные математические типы вроде матриц, но широко используемые малые языки высокого уровня (Си, Паскаль) не содержат этих возможностей и с точки зрения учебного курса по численным методам это хорошо, так как дает возможность глубоко изучить вычислительные алгоритмы, доводя их до программной реализации с сопутствующим вылавливанием алгоритмических ошибок.

Наиболее удобным инструментом для создания классов математических объектов является объектно-ориентированное программирование и его поддержка в языке C++. Этот язык дает возможность варьировать методы создания математических объектов путем определения в классе необходимого количества конструкто-

ров, осуществить переопределение стандартных операций для вновь созданных классов, использовать мощный механизм одиночного и множественного наследования свойств базовых классов в производных классах, создавать параметризованные классы и функции с подстановкой типов параметров в процессе конструирования соответствующих объектов. Последовательное наращивание иерархии математических типов на базе уже созданных позволяет существенно снизить трудоемкость программирования за счет исключения повторяющихся последовательностей действий и избежать внесения в программы новых ошибок.

Нам неизвестны пособия с систематическим изложением методов вычислений на базе объектно-ориентированного подхода в программной реализации на языке C++. Отчасти это объясняется известным консерватизмом нашей системы образования, ориентированной традиционно на использование Паскаля, а отчасти тем, что специалисты по программированию достаточно высокого профессионального уровня редко работают в наших учебных заведениях. Но уже сейчас во многих средних и высших учебных заведениях осуществляется преподавание языка C и C++ и предлагаемая работа может, по нашему мнению, оказать положительное влияние на эффективность учебного процесса в области вычислительной математики.

1.3. Особенности машинных вычислений

По-видимому, основной особенностью машинных вычислений по сравнению с ручными является ограниченная длина разрядной сетки и представление вещественных чисел в формате с плавающей десятичной точкой (экспоненциальный формат с нормализованной мантиссой). Это не позволяет осуществлять машинные операции вещественной арифметики так, как мы привыкли в чистой математике - численное представление вещественных значений требует, как правило, бесконечного количества цифр и не может быть осуществлено с абсолютной точностью. Проблема ошибок округления существует и в ручных вычислениях, но там она не так важна, потому что объем вычислений, которые могут быть выполнены вручную, невелик по сравнению с тем, что обычно выполняется на современных ЭВМ. Кроме того, при ручном счете эффекты округления непосредственно наблюдаемы и могут быть скорректированы

своевременным изменением длины числа или другими мерами предохранения от чрезмерных погрешностей результата, которая может быть достаточно легко оценена.

В машинных вычислениях получение такой оценки затруднительно - ранее разработанные теории оценок ошибок округления, как правило, непригодны, поэтому обычно прибегают к статистическим оценкам, основанным на допущении о независимости ошибок округлений; опыт показывает, что на самом деле эти ошибки часто коррелированны, и получаемые результаты сомнительны.

К сожалению, нет удовлетворительных теорий анализа ошибок округлений в вычислительных алгоритмах, поэтому в практике вычислений используют несколько эвристических программных методов диагностики наличия ошибок вычислений и программной оценки величины ожидаемой ошибки - прежде чем реализовать метод вычислений в программе, неплохо бы иметь представление об ожидаемой точности. Самым простым из них является определение допустимого диапазона для значений каждой переменной в виде максимальной и минимальной границ, внутри которых оно должно находиться. При выполнении действий над переменными новый диапазон вычисляется на основании определенного с учетом подходящих округлений и на каждой стадии вычислений определяются надежные границы, внутри которых должен размещаться верный ответ. Этот метод требует примерно удвоения объема вычислений и расходуемой памяти и позволяет получить только оценку попадания результата в область допустимых значений, но не позволяет оценить величину ошибки.

Другим распространенным методом является просчет задачи дважды - с одинарной и с двойной точностью представления чисел - предполагается, что верными можно считать совпадающие разряды в двух ответах.

Большие потери точности, как правило, происходят при вычитании двух близких по значению чисел - например, отличающихся только в последнем разряде; в начале числового результата образуется последовательность нулей, удаляемая при нормализации в операциях с плавающей точкой. Результат вычитания будет иметь значительно меньше значащих цифр, например одну, даже при отсутствии ошибок округления и использование результата вычитания в последующих вычислениях может привести к тому, что оконча-

тельный результат будет иметь только один верный знак. Если это возможно, необходимо исключить потерю знаков изменением последовательности вычислений или использовать метод так называемого счета со значащими разрядами. Он состоит в том, что нормализующий сдвиг мантиссы блокируется с сохранением ведущих нулей для сохранения количества значащих разрядов. Можно фактически осуществить нормализующий сдвиг, но с предварительным запоминанием числа ведущих нулей.

Результирующая погрешность вычислений, помимо распространения ошибок округлений при выполнении операций, обусловлена также влиянием других источников:

- ошибки дискретизации непрерывных зависимостей:
- погрешности в задании исходных данных;
- погрешности моделирования зависимости между исходными данными и вычисляемыми переменными;
- методические погрешности, определяемые несовершенством вычислительных алгоритмов;

Совершенно очевидно, что бессмысленно выдвигать высокие требования к точности вычислений при наличии больших ошибок в задании исходных данных или грубом моделировании исследуемых зависимостей. Мы постарались рассеять возможную избыточную наивную доверчивость к результатам машинных вычислений, но не можем привести библиографию источников с описанием надежных методов оценок погрешностей как до, так и после реализации вычислений. Но отношение к результатам в сложных программах с сотнями тысяч повторяющихся арифметических операций должно быть тем более скептическим, чем больше расхождения в этих результатах при выполнении программы с различными длинами мантисс в представлении чисел с плавающей точкой.

1.4. Структура учебного пособия

Настоящее учебное пособие рассчитано на сравнительно небольшой двухсеместровый курс численных методов (2 часа в неделю лекций + 2 часа лабораторных работ в компьютерном классе), который читается студентам специальности «Математика и информатика» педагогических институтов. Изложение курса предполагает владение основами объектно-ориентированного программирования на языке C++.

В соответствии с уже изложенной концепцией объектно-ориентированной программной реализации многие методы вычислений инкапсулируются в классы математических объектов, с которыми они работают; например, метод наименьших квадратов и метод решения систем линейных алгебраических уравнений будут размещены в классе матриц, а полиномиальная арифметика и методы вычисления полиномиальных нулей - в классе полиномов.

Глава 2 посвящена рассмотрению специальных математических типов (и операций над ними) и определению соответствующих им классов в терминах языка C++. Вначале в качестве иллюстрации рассматривается необходимый при изучении последующего материала (например, методов вычисления корней полиномов при наличии среди них комплексных) предположительно знакомый слушателю и реализованный в библиотеке C++ класс комплексных чисел; его программная реализация взята прямо из среды разработки Borland C++ и по возможности откомментирована - этот материал служит своеобразным образцом в реализации других рассмотренных в этой главе математических классов - векторов, полиномов, матриц. Изучение матричного класса сопровождается изложением методов решения основных задач линейной алгебры - систем линейных уравнений, вычисления собственных значений и векторов матриц.

Глава 3 содержит изложение методов полиномиальной и экспоненциальной аппроксимации функций и их программную реализацию, также инкапсулированную в виде функций - членов специального класса.

Глава 4 является естественным прикладным продолжением предыдущей и содержит методы численного интегрирования и дифференцирования функций с использованием рассмотренных методов приближения функций.

Глава 5 посвящена методам решения обыкновенных дифференциальных уравнений; при этом рассматриваются не только численные методы, а иллюстрируется применение приближенных численных методов при программной реализации аналитических решений. Например, при решении дифференциальных уравнений методами операционного исчисления возникает задача вычисления корней характеристических уравнений, которая может быть решена численно.

Глава 6 содержит введение в поисковые методы определения экстремумов функций при отсутствии и наличии шумов в определении значения функции и методы программирования соответствующих задач.

Глава 7 посвящена некоторым методам оптимизации функционалов (методам решения вариационных задач) и их программной реализации на примере метода динамического программирования.

По-видимому, некоторый материал покажется избыточным повторением предположительно известного из других математических курсов, но авторы ориентировались на известный им контингент слушателей и уровень его математической подготовки; невозможно было игнорировать также несогласованность учебных планов по высшей математике, вычислительным методам и информатике - зачастую вместо общематематической поддержки прикладных курсов картина получается обратной, так как прикладные дисциплины стоят в плане раньше поддерживающих математических дисциплин.

2. Специальные классы математических объектов и операции над ними

2.1. Комплексные числа

Основные понятия

Мнимая единица i (или j) определена как число, которое дает -1 при возведении в квадрат и приводит к надмножеству действительных чисел - комплексным числам, имеющим действительную и мнимую части и в алгебраической форме (в декартовой системе координат) имеющим вид:

$$c=Re+Im\cdot i \text{ или } c=Re+Im\cdot j$$

где Re, Im - вещественные числа, причем Re - вещественная, $Im\cdot i$ или $Im\cdot j$ - мнимая части комплексного числа.

В геометрической интерпретации в декартовой системе Re, Im - суть координаты точки на числовой плоскости с вещественной и мнимой осями или координаты конца вектора, проведенного в эту точку из начала координат.

Если использовать полярную систему координат, то получим тригонометрическую форму записи комплексного числа:

$$c=r\cdot(\cos\varphi+i\cdot\sin\varphi)$$

где $r=sqrt(Re^2+Im^2)$ - модуль комплексного числа, $\varphi=\text{arctg}(Im/Re)$ - его аргумент, то есть угол между радиус-вектором и вещественной осью.

Используется также показательная форма записи комплексного числа

$$c=r\cdot\exp(i\cdot\varphi)$$

Два комплексных числа называют *сопряженными*, если они отличаются только знаками мнимых частей; на комплексной плоскости изображающие их точки размещены симметрично относительно вещественной оси, их модули равны, а аргументы имеют противоположные знаки.

Операции над комплексными числами

Операции сравнения

Два комплексных числа равны, если равны их вещественные и мнимые части. Понятия «больше» или «меньше» для комплексных чисел не определены.

Алгебраические операции

Сложение и вычитание

$$c_1+c_2=(Re_1+Re_2)+(Im_1+Im_2)\cdot i.$$

Умножение

$$c_1\cdot c_2=(Re_1\cdot Re_2-Im_1\cdot Im_2)+(Re_1\cdot Im_2+Re_2\cdot Im_1)\cdot i.$$

или в тригонометрической форме

$$c_1\cdot c_2=r_1\cdot r_2\cdot(\cos(\varphi_1+\varphi_2)+i\cdot\sin(\varphi_1+\varphi_2)).$$

Деление

$$c_1/c_2=(Re_1\cdot Re_2+Im_1\cdot Im_2)/(Re_2^2+Im_2^2)+i\cdot[(Re_2\cdot Im_1-Re_1\cdot Im_2)/(Re_2^2+Im_2^2)]$$

или в тригонометрической форме

$$c_1/c_2=(r_1/r_2)\cdot(\cos(\varphi_1-\varphi_2)+i\cdot\sin(\varphi_1-\varphi_2)).$$

Возведение в степень (формула Муавра)

$$c^n=r^n\cdot(\cos(n\cdot\varphi)+i\cdot\sin(n\cdot\varphi)).$$

Определение программного класса комплексных чисел.

Класс комплексных чисел с операциями над ними определен в среде разработки Borland C++в файле complex.h. Мы воспользуемся этим файлом с упрощениями, не искажающими смысла, и снабдим необходимыми, на наш взгляд, комментариями - это хороший образец для последующего самостоятельного определения класса векторов, являющегося по существу обобщением класса комплексных чисел на многомерное пространство.

Итак, файл complex.h - включаемый библиотечный файл определения комплексных чисел. Для экономии места мы опускаем директивы препроцессора условной компиляции и другие несущественные детали.

```
class  complex {
//Нам понадобятся рабочие переменные для хранения Re и Im
double re, im;
```

```
//А дальше определим общедоступные функции-члены класса и
//начнем с конструкторов
public:
/*конструктор с инициализацией вещественной и мнимой частей
при объявлении комлексного числа в прикладной программе*/
complex(double     __re_val,     double     __im_val=0){
re=__re_val; im=__im_val;}
//конструктор по умолчанию
complex()  {re=im=0;}
//текущие значения вещественной и мнимой частей возвратят
//функции
double  real(){ return re;}
double  imag(){  return im;}

//сопряженное данному комплексное число
complex  conj(){ return complex(re,-im);}

//Модуль и аргумент комплексного числа
double norm () { return(re*re+im*im) ;}
double arg(){return (!re&&!im)?0:atan2(im,re);}

/*Операции над комплексными объектами.
Бинарные операции могут быть определены в двух вариантах, различающиеся местом размещения результата-либо с изменением текущего значения объекта, либо без, с помещением результата во внешней по отношению к объекту переменной комплексного типа. Мы определим простые бинарные операции как внешние дружественные классу функции не члены класса. Поэтому сначала объявим их прототипы, а определения приведем вне тела класса.
*/

//Операции проверки равенства и неравенства комплексных чисел
friend int operator==(complex _FAR &, complex _FAR &);
friend int operator!=(complex _FAR &, complex _FAR &);
/*Бинарные алгебраические операции над парами комплексных чисел, комплексным и вещественным, вещественным и комплексным
*/
friend complex operator+(complex _FAR&,complex _FAR&);
friend complex operator+(double, complex _FAR &);
friend complex  operator+(complex _FAR &, double);
friend complex operator-(complex _FAR&,complex _FAR&);
```

```
friend complex operator-(double, complex _FAR &);
friend complex operator-(complex _FAR &, double);
friend complex operator*(complex _FAR&,complex _FAR&);
friend complex  operator*(complex _FAR &, double);
friend complex  operator*(double, complex _FAR &);
friend complex operator/(complex _FAR&,complex _FAR&);
friend complex  operator/(complex _FAR &, double);
friend complex  operator/(double, complex _FAR &);

/*
```
Унарные +, - и * комбинированные с присвоением арифметические операции сделаем функциями-членами, изменяющими текущее значение комплексного объекта. Мы объединим их объявления с определениями.*/
```
complex  operator+(){ return*this;}
complex  operator-(){ return complex(-re,-im);}
complex _FAR &  operator+=(complex _FAR &)
{re+=__z2.re; im+=__z2.im; return*this;}
complex _FAR & operator+=(double __re_val2)
{ re+=__re_val2; return*this;}
complex _FAR & operator-=(complex _FAR & __z2)
{re-=__z2.re; im-=__z2.im; return*this;}
complex _FAR & operator-=(double __re_val2)
{ re-=__re_val2;    return*this;}
complex _FAR & operator*=(complex _FAR & __z2)
{    re=re*z2.real()-im*z2.imag();    im=re*z2.imag()+
z2.real()*im.}; return*this;}
complex _FAR & operator*=(double __re_val2)
{ re*=__re_val2; im*=__re_val2;  return*this;}
complex _FAR & operator/=(complex _FAR & __z2)
{re=(re*z2.real()+im*z2.image())/(z2.real()*z2.real()+
z2.imag()*z2.imag());  im=(z2.real()*im-re*z2.imag())/
(z2.real()*z2.real()+z2.imag()*z2.imag());
return*this;}
complex _FAR & operator/=(double __re_val2)
{ re/=__re_val2;im/=__re_val2; return*this;}

//В заключение определим и функцию потокового вывода
//комплексного объекта
ostream _FAR& operator<<(ostream _FAR&,complex _FAR&)
{return os<<"("<<x.real()<<","<<x.imag()<<")";
```

```
};
```

/* Теперь приведем реализацию дружественных операторных функций, не являющихся членами класса и реализующих бинарные операции над комплексными объектами. */

```
inline complex  operator+(complex _FAR & __z1, complex
_FAR & __z2)
{ return complex(__z1.re+__z2.re, __z1.im+__z2.im);}
inline  complex  operator+(double  __re_val1,  complex
_FAR & __z2)
{return complex(__re_val1+__z2.re, __z2.im);}

inline complex  operator+(complex _FAR & __z1, double
__re_val2)
{ return complex(__z1.re+__re_val2, __z1.im);}
inline complex  operator-(complex _FAR & __z1, complex
_FAR & __z2)
{ return complex(__z1.re-__z2.re, __z1.im-__z2.im);}
inline  complex   operator-(double  __re_val1,  complex
_FAR & __z2)
{   return complex(__re_val1-__z2.re,-__z2.im);}

inline complex  operator-(complex _FAR & __z1, double
__re_val2)
{ return complex(__z1.re-__re_val2, __z1.im);}

inline complex  operator*(complex _FAR & __z1, complex
_FAR & __z2)
{ double r=z1.real()*z2.real()-z1.imag()*z2.imag();
double i=z1.real()*z2.imag()+z2.real()*z1.imag()};
return complex(r,i);}
inline complex  operator*(complex _FAR & __z1, double
__re_val2)
{return complex(__z1.re*__re_val2,__z1.im*__re_val2);}
inline  complex   operator*(double  __re_val1,  complex
_FAR & __z2)
{return complex(__z2.re*__re_val1,__z2.im*__re_val1);}
inline complex  operator/(complex _FAR & __z1, complex
_FAR & __z2)
{
double r=(z1.real()*z2.real()+z1.imag()*z2.image())/
(z2.real()*z2.real()+z2.imag()*z2.imag());
```

```
double i=(z2.real()*z1.imag()-z1.real()*z2.imag())/
(z2.real()*z2.real()+z2.imag()*z2.imag());
return complex(r,i);
}
```

2.2. Векторы.

Основные понятия.

При изучении комплексных чисел мы уже ввели понятие векторного объекта на комплексной плоскости с числовыми компонентами. Теперь расширим это понятие до векторного объекта в *n*-мерном пространстве, понимая под ним последовательность из *n* чисел (в общем случае комплексных), которые будем называть составляющими вектора.

Совокупность всех таких векторов образует *n*-мерное векторное пространство $\mathbf{R}^n$.

Два вектора будем считать *равными* тогда и только тогда, когда все их компоненты равны.

Умножение вектора на число определим как операцию умножения всех составляющих вектора на это число.

Сложение векторов будет сводиться к сложению их составляющих.

Нулевым будем считать вектор, у которого все составляющие равны нулю.

Линейно-зависимыми будем считать векторы

$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)}$$

если существуют такие не все нулевые константы $C_1, C_2, \ldots, C_n$, что

$$C_1 \cdot \boldsymbol{x}^{(1)} + C_2 \cdot \boldsymbol{x}^{(2)} + \ldots + C_l \cdot \boldsymbol{x}^{(n)} = 0$$

В противном случае векторы считаются *линейно-независимыми*.

В последнем уравнении слева стоит вектор-сумма, который может по сделанному определению быть равен нулю при всех нулевых составляющих. Если количество векторов *n*, то это векторное уравнение равносильно системе из *n* уравнений с неизвестными C_1, C_2, …, C_n:

$$C_1 \cdot x_1^{(1)} + C_2 \cdot x_1^{(2)} + \ldots + C_l \cdot x_1^{(n)} = 0$$
$$C_1 \cdot x_2^{(1)} + C_2 \cdot x_2^{(2)} + \ldots + C_l \cdot x_2^{(n)} = 0$$
$$\ldots\ldots\ldots$$

$$C_1 \cdot x_n^{(1)} + C_2 \cdot x_n^{(2)} + \ldots + C_l \cdot x_n^{(n)} = 0$$

Если число векторов больше размерности пространства $n<l$, то есть число уравнений меньше числа неизвестных, то система видимо будет иметь отличные от нулевого решения для C_j и наши векторы будут соответственно линейно-зависимыми. Другими словами - число линейно-независимых векторов не больше размерности пространства.

При $n=l$ (число уравнений равно числу неизвестных) система может иметь ненулевое решение (а векторы быть линейно-зависимыми) только тогда, когда ее определитель равен нулю.

Таким образом, для линейной независимости векторов необходимо и достаточно, чтобы определитель, составленный из составляющих векторов, был отличен от нуля или чтобы ранг матрицы, составленной из векторов, был равен числу векторов.

Введем теперь новое понятие, которое в дальнейшем будет широко использоваться:

Скалярным произведением $(\boldsymbol{x},\boldsymbol{y})$ двух векторов

$$\boldsymbol{x}(x_1, x_2, \ldots, x_n) \text{ и } \boldsymbol{y}(y_1, y_2, \ldots, y_n)$$

называется число, равное следующей сумме: $\sum_{s=1}^{n} x_s \cdot y_s$.

Взаимно-ортогональными или взаимно-перпендикулярными будем называть два вектора, если их скалярное произведение равно нулю.

Корень квадратный из скалярного произведения $(\boldsymbol{x},\boldsymbol{x})$ вектора $\boldsymbol{x}(x_1, x_2, \ldots, x_n)$ на этот же вектор

$$\|\boldsymbol{x}\|^2 = (\boldsymbol{x},\boldsymbol{x}) = \sum_{s=1}^{n} x_s^2 ; \qquad \|\boldsymbol{x}\| = \sqrt{\sum_{s=1}^{n} x_s^2} .$$

называют *нормой*, длиной или модулем вектора $\boldsymbol{x}$.

Использование скалярного произведения позволяет записать систему однородных линейных алгебраических уравнений в виде:

$$(\boldsymbol{x},\boldsymbol{a_j}) = 0 \ (i=1, 2, \ldots, n; j=1, 2, \ldots, n)$$

откуда очевидно, что ее решение сводится к нахождению вектора $\boldsymbol{x}$, ортогонального ко всем векторам $\boldsymbol{a_j}$. Решением этой задачи мы займемся в разделе, посвященном матрицам и линейной алгебре, а пока ограничимся тем набором понятий и операций, которые уже нами определены и разместим соответствующие данные и методы в классе C++.

Определение программного класса многомерных векторов.

Итак, составим включаемый файл vector.h, который будет содержать все, о чем мы договорились в предыдущем теоретическом подразделе. Для этого вначале рассмотрим структуру класса для работы с векторами, алгоритмы для работы с объектами типа «вектор» и примеры методов векторного класса, реализующие эти алгоритмы:

```
template <class YourOwnFloatType> class vector
{
```

Класс для работы с векторными объектами- параметризованный, то есть мы определяем только шаблон, по которому для конкретных типов, подставляемых вместо YourOwnFloatType, компилятор автоматически генерирует класс.

```
 long m;
```

Вектор будет характеризоваться размерностью (длиной), которая может быть любым натуральным числом. Скаляры можно рассматривать как вектора размерности 1, комплексные числа - как вектора размерности 2 и т.д. В общем случае размерность вектора равна числу его составляющих, т.е. размерность вектора ($a_1, a_2, \ldots, a_n$) - n.

```
 YourOwnFloatType ·vec;
```

Данные, характеризующие вектор, мы будем хранить по этому указателю. В данном случае это - компоненты, или, как их ещё называют, координаты вектора. Это название не случайно: геометрические вектора размерности 2 - направленные отрезки на плоскости - могут быть связаны в декартовой системе координат с арифметическими координатами своих концов, если считать их всегда исходящими из начала координат.

```
 virtual void In(istream &);
 virtual void Out(ostream &);
```

Виртуальные функции чтения из потока и записи в поток; их необходимо переопределить в производных классах для обеспечения возможности использования единых перегруженных операторов << и >>. Явно использовать их необходимости нет, потому они и перенесены в приватную часть.

```
public:
```

В этом разделе - общедоступные методы класса, которые можно использовать для манипуляций с векторными объектами.

```
vector(char ·);
```

Конструктор класса «вектор», параметром которого является имя текстового файла. Предполагается, что первым числом в этом файле является размерность вектора, после которого следуют данные. Этот конструктор определяет, откуда мы можем взять данные о векторе - размерность и компоненты.

```
vector();
```

Это конструктор по умолчанию, создающий нулевой вектор единичной размерности. Необходим при динамическом создании массивов векторов. Геометрически такой вектор - это точка на числовой прямой в отметке «0»; такой подход связан с трудностью решения вопроса о размерности вектора по умолчанию и отнюдь не является единственно правильным. Основная причина появления этого конструктора здесь - это требование компилятора.

```
vector(long);
```

Параметром этого конструктора является размерность вектора; после создания компоненты вектора обнуляются. Если, к примеру, принимаемая размерность - 5, то мы получим вектор вида (0, 0, 0, 0, 0).

```
vector(long,YourOwnFloatType ·);
```

Этот конструктор пытается создать вектор размерности, заданной первым параметром, и заполнить его данными из массива. Полезен в том случае, когда компоненты вектора заранее известны.

Пусть, к примеру, параметры этого конструктора - размерность 3 и некий массив с числами 1, 2, 3, 4, 5, 6, …. В этом случае данный конструктор создаст вектор (1, 2, 3).

```
vector(vector<YourOwnFloatType> &);
```

Конструктор копирования (инициатор копии), создающий новый вектор из уже имеющегося. Размерность нового вектора устанавливается в размерность старого, а данные переписываются без изменений. В результате его выполнения мы получаем вектор, идентичный копируемому, но находящийся в другом участке памяти.

```
~vector();
```

Деструктор. Освобождает динамически распределённую память из-под компонент вектора.

```
friend vector<YourOwnFloatType> operator+ (vector<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

Дружественная функция сложения двух векторов. Складывает вектора только совпадающих размеров, вектор-результат конструируется и возвращается. При этом вектора одинаковых размерностей складываются по закону:

$$(a_1, a_2, ..., a_n) + (b_1, b_2, ..., b_n) = (c_1, c_2, ..., c_n),$$
$$c_i = a_i + b_i .$$

```
friend vector<YourOwnFloatType> operator+= (vector<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

Перегруженная операция сокращённого сложения складывает первый вектор со вторым, модифицируя при этом первый вектор (записывая результат сложения в него) и возвращает результат для того, чтобы он мог участвовать в других операциях над векторами типа a+(c=y) и т.п.

```
friend vector<YourOwnFloatType> operator-(vector<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

Дружественная функция для вычитания векторов одинаковой размерности; вычитает первый вектор из второго, конструируя и возвращая результат как новый вектор. При этом вектора одинаковых размерностей вычитаются по закону:

$$(a_1, a_2, ..., a_n) - (b_1, b_2, ..., b_n) = (c_1, c_2, ..., c_n),$$
$$c_i = a_i - b_i .$$

```
friend vector<YourOwnFloatType> operator-=(vector<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

Сокращённое вычитание, работающее так же, как и сокращённое сложение: из первого вектора вычитается второй, результат снова записывается в первый вектор, а его копия возвращается.

```
friend YourOwnFloatType operator·(vector<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

Скалярное произведение двух векторов одинаковой размерности **-** число того же типа, что и компоненты исходных векторов. Его можно получить по следующему закону:

$$(a_1, a_2, ..., a_n) \cdot (b_1, b_2, ..., b_n) = \sum_{i=1}^{n} a_i b_i$$

```
friend vector<YourOwnFloatType> operator·(YourOwnFloatType,
vector<YourOwnFloatType> &);
```

Ещё одна дружественная функция, перегружающая операцию умножения в несколько ином контексте, а именно **-** как умножение скаляра на вектор. Результатом является вектор той же размерности, что и исходный. Произведением вектора $\boldsymbol{a}(a_1, a_2, ..., a_n)$ на число α будет вектор $\alpha\boldsymbol{a}(\alpha a_1, \alpha a_2, ..., \alpha a_n)$, то есть вектор, каждая компонента которого умножена на это число.

```
friend vector<YourOwnFloatType> operator* (vec-
tor<YourOwnFloatType> &, YourOwnFloatType);
```

Умножение скаляра на вектор операция коммутативная, что не избавляет нас от необходимости снова перегрузить операцию умножения, с теми же аргументами, но в другом порядке. Действительно, из соотношения $\alpha\boldsymbol{a}=(\alpha a_1, \alpha a_2, ..., \alpha a_n)$ мы легко можем перейти к соотношению $(a_1\alpha, a_2\alpha, ..., a_n\alpha) = \boldsymbol{a}\alpha$.

```
friend vector<YourOwnFloatType> operator*= (vec-
tor<YourOwnFloatType> &, YourOwnFloatType);
```

Сокращённое умножение вектора на число. Первый аргумент этой функции – вектор **-** после выполнения данной операции модифицируется, копия результата возвращается.

```
friend ostream &operator<<(ostream &, vec-
tor<YourOwnFloatType> &);
```

Перегруженная операция вывода вектора в поток. При этом выводятся только компоненты вектора, разделяемые пробелами; размерность не указывается. Модифицированный поток возвращается.

```
friend istream &operator>>(istream &,
vector<YourOwnFloatType> &);
```

Ввод вектора из потока осуществляется путём приёма из потока количества чисел, равного размерности вектора. Модифицированный поток ввода возвращается для участия в дальнейших операциях ввода из него.

```
vector<YourOwnFloatType> opera-
tor=(vector<YourOwnFloatType> &);
```

Присвоение **-** операция, которая не может быть перегружена с использованием механизма дружественных функций. Первым, неявным параметром этой функции является текущий объект (*this),

вторым - объект, присваиваемый текущему. При этом, если размерности присваиваемого и текущего векторов совпадают, компоненты последнего просто переписываются. В противном же случае размерность текущего вектора устанавливается в размерность копируемого, а память под компоненты перераспределятся, и только тогда происходит перепись данных. Возвращаемым значением является сам текущий вектор.

```
vector<YourOwnFloatType> operator-();
```

Унарный минус. Эта операция создаёт вектор той же размерности, что и текущий, с компонентами, имеющими противоположный знак, то есть Вектор, противоположный вектору $\boldsymbol{a}$ с координатами ($a_1, a_2, \ldots, a_n$) - вектор ($-\boldsymbol{a}$) с координатами ($-a_1,-a_2, \ldots,-a_n$).

```
vector<YourOwnFloatType> operator+();
```

Эта функция-член включена только для полноты набора. Не выполняя ничего полезного, она просто возвращает копию текущего вектора.

```
vector<YourOwnFloatType> operator~();
```

Метод, для данного ненулевого вектора конструирующий вектор, имеющий то же направление и единичную длину, т.е. выполнятся нормирование данного вектора по модулю. Так как компонентами вектора-результата фактически являются косинусы углов данного вектора к координатным осям (а также длины проекций, т.к. вектор-результат - единичный), то данную операцию называют ещё и *определением направляющих косинусов*.

Модуль вектора ($a_1, a_2, \ldots, a_n$) определим как корень квадратный из скалярного произведения этого вектора на себя: $|(a_1,a_2,...,a_n,)|=\sqrt{\sum_{i=1}^{n} a_i^2}$, а операцию нормирования по модулю - как деление каждой составляющей вектора на его модуль, или же умножение ненулевого вектора на величину, обратную модулю:

$$\frac{1}{\sqrt{\sum_{i=1}^{n} a_i^2}}(a_1,a_2,...,a_n)=\left(\frac{a_1}{\sqrt{\sum_{i=1}^{n} a_i^2}},\frac{a_2}{\sqrt{\sum_{i=1}^{n} a_i^2}},...,\frac{a_n}{\sqrt{\sum_{i=1}^{n} a_i^2}}\right).$$

```
YourOwnFloatType operator!();
```

Метод, для вектора любой размерности определяющий его модуль. При этом делается, заметим, далеко не всегда корректное

допущение, что длина вектора выражается в единицах того же типа, что и компоненты вектора. Например, для целых векторов эта операция даст лишь приближённый результат, а для многомерных векторов (например, комплексных) данная операция имеет смысл нормы.

```
virtual long IsEqual(void *);
```

Эта виртуальная функция сравнивает текущий вектор с объектом, лежащим по адресу, передаваемому через обобщённый указатель. При этом делается неявное предположение о том, что указатель содержит адрес вектора, а не чего-либо более экзотического. Преобразуя получаемый указатель к указателю на вектор, эта функция пытается сравнить его с текущим. Заметим, что вектора будут считаться равными, если выполнятся два условия - одинаковая размерность и совпадающие компоненты:

$$(a_1, a_2, ..., a_n) = (b_1, b_2, ..., b_n) \Leftrightarrow a_1 = b_1 \wedge a_2 = b_2 \wedge ... \wedge a_n = b_n$$

```
friend long operator==(vector<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

Используя описанную выше функцию, вводим операторное сравнение двух векторов - проверка на равенство…

```
friend long operator!=(vector<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

…и проверка на неравенство.

```
YourOwnFloatType &operator[](long a);
```

Индексирование элементов вектора - это операция, которая возвращает ссылку на компоненту вектора с заданным номером. Если этот номер выходит за границы размерности вектора, после диагностики возвращается специальный код ошибки. Так как данная функция ссылочная, то её можно использовать в операторах присваивания как слева, так и справа - ведь модифицируя ссылочный объект, мы фактически воздействуем на то, что под ним скрывается, то есть на саму векторную компоненту. Заметим, что вектор размерности *a* можно индексировать от 0 до (*a*-1), а не от 1 до *a* (!).

```
long getm() { return m; }
```

Размерность вектора можно узнать, используя этот метод.

```
};
```

По приведенным описаниям можно составить, к примеру, такой интерфейс для данного класса:

```
#ifndef __VECTOR_H
#define __VECTOR_H
#ifndef __FSTREAM_H
#include <fstream.h>
#endif
#ifndef __IOMANIP_H
#include <iomanip.h>
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#endif
#ifndef __MATH_H
#include <math.h>
#endif
#ifndef __EXCEPT_H
#include <except.h>
#endif
#ifndef __CSTRING_H
#include <cstring.h>
#endif
```

//параметризованный класс для работы с векторными объектами

```
template <class YourOwnFloatType>//подставьте свой тип
class vector
{ //приватные данные
  long m;                //размерность (длина) вектора
  YourOwnFloatType *vec;//указатель на элементы вектора
 //виртуальные функции чтения из потока и записи в поток;
 //их необходимо переопределить в производных классах
 //для обеспечения возможности использования единых
 //перегруженных операторов << и >>
  virtual void In(istream &);
  virtual void Out(ostream &);
 public://общедоступные данные и функции
 //загрузка вектора из файла: dimension data1 data2...
  vector(char *);
  vector();//создание пустого вектора единичной размерности
  vector(long);//создание пустого вектора заданной размерности
  vector(long,YourOwnFloatType *);
```

```
  //создание вектора заданной размерности,
  //заполняемого данными из массива
//конструктор копирования
  vector(vector<YourOwnFloatType> &);
  ~vector();//деструктор
  friend      vector<YourOwnFloatType>      operator+
(vector<YourOwnFloatType> &, vector<YourOwnFloatType>
&);//сложение двух векторов
  friend  vector<YourOwnFloatType>  operator+=  (vec-
tor<YourOwnFloatType>   &,   vector<YourOwnFloatType>
&);//сложение с присвоением
  friend      vector<YourOwnFloatType>      operator-
(vector<YourOwnFloatType> &, vector<YourOwnFloatType>
&);//вычитание
  friend  vector<YourOwnFloatType>  operator-=  (vec-
tor<YourOwnFloatType>   &,   vector<YourOwnFloatType>
&);//вычитание с присвоением
  friend     YourOwnFloatType     operator*     (vec-
tor<YourOwnFloatType>   &,   vector<YourOwnFloatType>
&);//скалярное умножение
  friend vector<YourOwnFloatType> operator* (YourOwn-
FloatType, vector<YourOwnFloatType> &); //умножение
числа на вектор
  friend  vector<YourOwnFloatType>  operator*  (vec-
tor<YourOwnFloatType>    &,    YourOwnFloatType    );
//умножение вектора на число
  friend  vector<YourOwnFloatType>  operator*=  (vec-
tor<YourOwnFloatType>    &,    YourOwnFloatType    );
//умножение вектора на число с присвоением
  friend   ostream   &operator<<(ostream   &,   vec-
tor<YourOwnFloatType> &);//вывод вектора в поток
  friend   istream   &operator>>(istream   &,   vec-
tor<YourOwnFloatType> &);//ввод вектора из потока
  vector<YourOwnFloatType>      operator=      (vec-
tor<YourOwnFloatType> &);//присвоение
  vector<YourOwnFloatType> operator-();//унарный минус
  vector<YourOwnFloatType> operator+();//унарный плюс
  vector<YourOwnFloatType> operator~();//нормирование
              //(определение направляющих косинусов)
  YourOwnFloatType operator!();//модуль вектора
  virtual long IsEqual(void *);
```
/*эта виртуальная функция сравнивает текущий вектор с объектом, лежащим по адресу, передаваемому через обобщённый указатель*/

```
  friend long operator==(vector<YourOwnFloatType> &,
  vector<YourOwnFloatType> &);//проверка на равенство
  friend long operator!=(vector<YourOwnFloatType> &,
 vector<YourOwnFloatType> &);//проверка на неравенство
  YourOwnFloatType &operator[](long a);
     //индексирование элементов вектора
  long getm() { return m; }//размерность вектора
};
```

//Теперь реализация объявленных методов класса «vector».

```
//индикатор ошибки - некая константа
const long double MAX_LONGDOUBLE=1.7976931348e308;
```

/*

Создавать вектор можно по-разному. Например, если он находится на внешнем устройстве в формате m d1 d2... dm, где m - размерность вектора, a di - его компоненты, то имеем следующий конструктор:

*/

```
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector(char *f)//имя файла
{
  long i;

  ifstream fp=f;//пытаемся открыть файл
  if(!fp)//если не удалось
    throw xmsg(«Не могу открыть файл «+string(f)+»\n»);
  fp>>m;//вводим размерность
  if(m<=0)//проверка на корректность
    throw xmsg("Размерность вектора некорректна\n");
    //диагностика
  try
  {
   vec=new YourOwnFloatType[m];//попытка выделения памяти
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти\n");
  }
  for(i=0;i<m&&fp>>vec[i];i++);//считывание из файла
}
```

```
/*
```
В случае, когда нам известна лишь размерность вектора, но неизвестны его составляющие, предполагаем, что данный вектор является нулевым:
```
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector(long a):m(a)
                                //размерность вектора
{
  long i;

  if(m<=0)//проверка размерности
    throw xmsg("Размерность вектора некорректна\n");
      //диагностика
  try
  {
   vec=new YourOwnFloatType[m];//попытка выделения памяти
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти\n");
  }
  for(i=0;i<m;vec[i++]=0);//обнуление компонент вектора
}


/*
  Наконец, нам могут быть известны как размерность,
так и компоненты вектора:
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector(long a,
YourOwnFloatType *v):m(a)
//этот конструктор принимает размер и указатель на данные
{
  long i;

  if(m<=0)//проверка размерности
    throw xmsg("Размерность вектора некорректна\n");
      //диагностика
  try
```

```
  {
   vec=new YourOwnFloatType[m];//попытка выделения памяти
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти\n");
  }
  for(i=0;i<m;i++)
    vec[i]=v[i];//копирование из внешнего массива в вектор
}
```

/*

Есть ещё один случай, когда мы ничего не можем сказать о размерности и компонентах вектора - при создании массива векторов, то есть матрицы, когда для оператора new требуется конструктор без параметров или когда размер вектора заранее неизвестен.

*/

```
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector():m(1)
{
  try
  {
   vec=new YourOwnFloatType[m];//попытка выделения памяти
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти\n");
  }
  *vec=0;//обнуляем единственный имеющийся элемент
}
```

/*

В реальных расчетах могут использоваться векторы больших размерностей, поэтому размещаются они в свободной памяти компьютера, а когда необходимость в них отпадает - уничтожаются.

*/

```
template <class YourOwnFloatType>
vector<YourOwnFloatType>::~vector()
{
```

```
  delete []vec;//уничтожение динамического массива
}


/*
 Необходимость в индексации вектора возникает в двух случаях:
 - при получении составляющей вектора по её номеру и
 - при изменении не всего вектора, а только одной его составляющей.
 При этом, конечно, следует учитывать возможность ошибочного задания номера составляющей: допустимый диапазон значений [0,m).
*/
template <class YourOwnFloatType> YourOwnFloatType &
vector<YourOwnFloatType>::operator[](long a)
{
  static YourOwnFloatType error=MAX_LONGDOUBLE;
  if(a>=0&&a<m)//если всё ОК
    return vec[a];
  else//при выходе за пределы вектора ругаемся
  {
    cerr<<"Индекс "<<a<<" вне диапазона вектора\n";
    return error;
  }
}


/*
  Создавая вектор, можно попутно инициализировать его данными из уже существующего:
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector(                    vec-
tor<YourOwnFloatType> &ex) : m(ex.m)
//это конструктор копирования, принимающий ссылку на вектор
{
  try
  {
   vec=new YourOwnFloatType[m];//попытка выделения памяти
  }
  catch(xalloc)
  {
```

```
    throw xmsg("Не хватает памяти\n");
  }
  for(long i=0;i<m;i++)
    vec[i]=ex[i];
//здесь при копировании ex используется уже индексация
}


/*
 Сложение векторов является алгебраической операцией только тогда, когда вектора одинаковой размерности. Результатом сложения является вектор той же размерности, что и исходные, компонентами которого является сумма соответствующих компонент исходных векторов.
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator+
(vector<YourOwnFloatType>  &f,vector<YourOwnFloatType>
&s)
{
  if(f.m!=s.m)//проверка на равенство размерностей
    throw xmsg("Слагаемые вектора имеют различные длины\n");
      //диагностика
  vector<YourOwnFloatType> temp(f.m);
      //создаём временный вектор
 //здесь работают операции индексирования для всех трёх векторов
  for(long i=0;i<f.m;i++)
    temp[i]=f[i]+s[i];
  return temp;//возвращаем результирующий вектор
}


//сокращённая операция "сложение с присвоением"
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator+=
(vector<YourOwnFloatType>  &f,vector<YourOwnFloatType>
&s)
{
  return f=f+s;
}
```

```
/*
  Введём несколько вспомогательных унарных операций:
- "минус":
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType>    vector<YourOwnFloatType>::
operator-()
{
  vector<YourOwnFloatType> temp(m);
      //создаём временный вектор
```
/*Если this - это указатель на текущий объект векторного класса, то this - это сам текущий объект класса vector, то есть тот, с которым мы сейчас работаем. А к любому векторному объекту мы можем применить операцию индексирования */
```
  for(long i=0;i<m;i++)
    temp[i]=-(*this)[i];
  //temp[i]=-vec[i];
  //temp.vec[i]=-vec[i];
  //temp.operator[](i)=-operator[](i); etc...
  return temp;//возвращаем результирующий вектор
}


//унарный плюс
template <class YourOwnFloatType>
vector<YourOwnFloatType>    vector<YourOwnFloatType>::
operator+()
{
  return *this;//возвращаем самого себя
}


/*
  Операция, которую алгебраической назвать нельзя -
это, скорее, пример очень распространённого тернарного
отношения "скалярное произведение двух векторов":
*/
template <class YourOwnFloatType>
YourOwnFloatType    operator*(vector<YourOwnFloatType>
&f, vector<YourOwnFloatType> &s)
{
  if(f.m!=s.m)
```

```
    throw xmsg("Умножение векторов с несовпадающими "
               "размерами невозможно\n");//диагностика
  YourOwnFloatType temp=0;
  for(long i=0;i<f.m;i++)
    temp+=f[i]*s[i];
      //суммируем произведения составляющих векторов
  return temp;
}


/*
  Модуль вектора как квадратный корень скалярного про-
изведения вектора на самого себя:
*/
template  <class  YourOwnFloatType>  inline  YourOwn-
FloatType vector<YourOwnFloatType>::operator!()
{
  return sqrt((*this)*(*this));
}


/*
- нормирование вектора по модулю
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType>                              vec-
tor<YourOwnFloatType>::operator~()
{
  vector<YourOwnFloatType> temp(m);
 //скалярное произведение текущего объекта на самого себя
  YourOwnFloatType modul=!(*this);
  for(long i=0;i<m;i++)
    temp[i]=(*this)[i]/modul;//направляющие косинусы
  return temp;
}


/*
      умножение числа на вектор":
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator*
(YourOwnFloatType ld,vector<YourOwnFloatType> &v)
```

```
{
  vector<YourOwnFloatType> temp=v;
  for(long i=0;i<v.getm();i++)
    temp[i]=temp[i]*ld;//скорее, это даже "удлинение" вектора
  return temp;
}


/*
  умножение вектора на число:
*/
template <class YourOwnFloatType>
inline vector<YourOwnFloatType> operator*
(vector<YourOwnFloatType> &v,YourOwnFloatType ld)
{
  return ld*v;//очень просто - вызвали другую функцию
}


//операция сокращённого умножения вектора на число
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator*=
(vector<YourOwnFloatType> &v,YourOwnFloatType ld)
{
  return v=v*ld;
}
```

/*

Имея определённые бинарную операцию сложения векторов и унарную получения вектора, противоположного к данному, можно на векторном языке, не обращаясь к компонентам векторов, определить операцию вычитания:

*/

```
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator-
(vector<YourOwnFloatType>  &f,vector<YourOwnFloatType>
&s)
{
  return f+(-s);
  //return operator+(f,-s);
  //return operator+(f,s.operator-());
```

```
}


//операция сокращённого вычитания
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator-=
(vector<YourOwnFloatType>   &f,vector<YourOwnFloatType>
&s)
{  return f=f-s;}


/*
 При переписывании одного вектора в другой возможны два случая:
 1. если размерность обоих векторов совпадает, то просто заменяем составляющие первого вектора компонентами второго;
 2. в противном случае безжалостно уничтожаем первый вектор и создаём снова, используя второй как строительный материал.
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType>    vector<YourOwnFloatType>::
operator=(vector<YourOwnFloatType> &x)
{
  if(m!=x.m)//если размеры не совпадают
  {
    delete []vec;//уничтожаем содержимое текущего вектора
    m=x.m;//устанавливаем новый размер
    try
    {
   vec=new YourOwnFloatType[m];//попытка выделения памяти
    }
    catch(xalloc)
    {
      throw xmsg("Не хватает памяти\n");
    }
  }
  for(long i=0;i<m;i++)
    vec[i]=x[i];//копируем данные из вектора x в текущий
```
/*присвоение - это бинарная операция, первым параметром которой является объект, которому присваивают, вторым - объект, который присваивают. При этом первый объект, в отличие от всех остальных

бинарных операций, меняется, и он же возвращается в качестве результата (это бывает необходимым для операций вида a=b=c;)*/
```
  return *this;
}
```

/*Эта функция сравнивает текущий вектор с вектором, лежащим по адресу x. Для каждого класса, производного от векторного, не имеет смысла переопределять операторные функции проверки на равенство и неравенство - достаточно переопределить эту виртуальную функцию*/
```
template <class YourOwnFloatType>
long vector<YourOwnFloatType>::IsEqual(void *x)
{
  if(m!=((vector<YourOwnFloatType>*)x)->m)
    //при несовпадении размерностей
    return 0; //констатируем несовпадение векторов
  for(long i=0;i<m;i++)
    if((*this)[i]!=(*(vector<YourOwnFloatType>*)x)[i])
      return 0;//если хоть один элемент не совпал
  return 1;
}
```

/*
Сравнение векторов является тернарным отношением, результатом которого является число нуль, если векторы не равны и единица в противном случае.

Два вектора будем считать равными, если они имеют одинаковые длины и их соответствующие составляющие совпадают:
*/
```
template <class YourOwnFloatType>
long operator==(vector<YourOwnFloatType> &f,
vector<YourOwnFloatType> &s)
{
  return f.IsEqual(&s);
}
```

```
/*
```

```
  Неравенство векторов определим через равенство и
операцию отрицания:
*/
template <class YourOwnFloatType> inline long opera-
tor!=(vector<YourOwnFloatType> &f,
vector<YourOwnFloatType> &s)
{
  return !(f==s);//логично
}
```

/*Мощный I/O-механизм C++ позволяет в естественной форме выводить (вводить) векторы на любое устройство отображения информации. Для универсализации считывания и записи вектора в поток снова прибегнем к механизму виртуальных функций. С этой целью, по аналогии с printOn, определим две функции - одну для ввода, другую - для вывода*/

```
template <class YourOwnFloatType>
void vector<YourOwnFloatType>::In(istream &is)
{
  for(long i=0;i<m;i++)
    is>>(*this)[i];
}

template <class YourOwnFloatType>
void vector<YourOwnFloatType>::Out(ostream &os)
{
  for(long i=0;i<m;i++)
  {
    os.precision(100);
    os<<(*this)[i]<<" ";//компоненты разделяем пробелами
  }
}

//вывод в поток
template <class YourOwnFloatType>
ostream &operator<<(ostream &os,
vector<YourOwnFloatType> &x)
{ x.Out(os);  return os;}

/*  - ввод из потока*/
```

```
template <class YourOwnFloatType>
istream &operator>>(istream &is,
vector<YourOwnFloatType> &x)
{  x.In(is);
  return is;//принимаем и возвращаем ссылку на поток ввода
}

#endif
```

2.3. Полиномы.

Общие сведения.

Многочлен с целочисленными степенями переменных (полином) в общем виде записывается так:

$$f(z)=a_n\cdot z^n+a_{n-1}\cdot z^{n-1}+\ldots+a_{n-k}\cdot z^{n-k}+\ldots+a_1\cdot z+a_0,$$

где a_0, a_1, …, a_k, …, a_n - заданные числа (в общем случае - комплексные), z - переменная (в общем случае - комплексная). Старший коэффициент a_0 в соответствии со здравым смыслом мы будем считать отличным от нуля.

Значения z, при подстановке которых полином обращается в нуль, называются *корнями* (или нулями) этого полинома, то есть корни полинома есть решения уравнения

$$f(z)=a_n\cdot z^n+a_{n-1}\cdot z^{n-1}+\ldots+a_{n-k}\cdot z^{n-k}+\ldots+a_1\cdot z+a_0=0$$

Это уравнение называют *алгебраическим уравнением n-й степени*.

При делении $f(z)$ на двучлен $(z-a)$ частное $Q(z)$ будет многочленом n-1-й степени со старшим коэффициентом a_0, остаток R не будет содержать z, то есть имеет место тождество:

$$f(z)=(z-a)\cdot Q(z)+R$$

После подстановки в него $z=a$ получим $R=f(a)$ - остаток при делении полинома на $(z-a)$ равен $f(a)$ (теорема Безу). При делении без остатка (с нулевым остатком) $f(a)=0$, то есть $z=a$ должно быть корнем полинома. Зная этот корень, можно выделить из полинома множитель $(z-a)$:

$$f(z)=(z-a)\cdot f_1(z),$$

где

$$f_1(z)=b_{n-1}\cdot z^{n-1}+b_{n-2}\cdot z^{n-2}+\ldots+b_1\cdot z+b_0\ (b_{n-1}=a_0)$$

и для нахождения остальных корней надо решить уравнение на один порядок ниже:

$$b_{n-1}\cdot z^{n-1}+b_{n-2}\cdot z^{n-2}+\ldots+b_1\cdot z+b_0=0$$

В соответствии с основной теоремой алгебры, всякое алгебраическое уравнение имеет хотя бы один вещественный или комплексный корень (например, z_1) и делится на $(z-z_1)$, полином-частное тоже будет иметь корень и делится на $(z-z_2)$ и т.д. - таким образом, всякий полином степени n разлагается на $n+1$ множителей, один из которых равен старшему коэффициенту, а остальные есть двучлены вида $(z-a)$.

$$f(z)=a_n\cdot(z-z_1)\cdot(z-z_2)\cdot\ldots\cdot(z-z_n)$$

Это разложение на множители единственно.

Среди корней полинома могут быть кратные - необходимым и достаточным условием того, что значение $z=a$ является корнем кратности k является обращение в нуль при этом значении полинома и всех его производных до $(k-1)$-й включительно и необращение в нуль k-й производной. Корень кратности k некоторого полинома является корнем кратности $(k-d)$ для d-й производной этого полинома, то есть если имеет место разложение полинома

$$f(z)=a_n\cdot(z-z_1)^{k1}\cdot(z-z_2)^{k2}\cdot\ldots\cdot(z-z_m)^{km},$$

где $z_1, z_2,\ldots z_m$ - различны и $k1+k2+\ldots+km=n$,
то разложение производной будет

$$f'(z)=(z-z_1)^{k1-1}\cdot(z-z_2))^{k2-1}\cdot\ldots\cdot(z-z_m)^{km-1}\cdot w(z),$$

где $w(z)$ - полином, уже не имеющий общих с $f(z)$ корней.

Наибольший общий делитель двух полиномов есть произведение всех общих для них двучленных множителей с меньшими из двух вариантов показателями степени. Если полиномы не имеют общих корней, то они - взаимно-простые. Составление полинома-наибольшего общего делителя двух других полиномов можно выполнить известным в арифметике методом определения НОД двух целых чисел: полином со степенью не меньше степени второго делим на второй, затем второй делим на остаток при первом делении, этот первый остаток делим на остаток при втором делении и т.д. до получения нулевого остатка. Последний ненулевой остаток и есть НОД и если он не содержит z, то полиномы взаимно-простые. Разделив полином на его НОД и НОД его производной, получим полином, имеющий все простые корни, совпадающие с различными кор-

нями исходного полинома - так можно освободиться от кратных корней без решения уравнения $f(z)=0$.

Операции с полиномами.

Основные операции с полиномами хорошо известны из элементарной алгебры, поэтому здесь мы остановимся подробно только на операции деления многочленов и процедурах вычисления значений и корней полиномов. Эти операции и основанные на них основные результаты будут нами часто использоваться и поэтому заслуживают подробного рассмотрения - насколько это возможно в кратком курсе численных методов. Методы определения корней многочленов, как правило (за исключением решений, получаемых в квадратурах для полиномов порядка не выше четвертого), пригодны и для вычисления корней уравнений неалгебраического типа (трансцендентных).

Деление многочленов осуществляют по правилам, принятым для целочисленного деления, при котором в качестве результата получают частное и остаток. Проще всего реализовать его по известным правилам деления "в столбик" по методу Евклида.

В качестве первого элемента частного от деления полинома $f_1(z)$ на полином $f_2(z)$ берут переменную z в степени, равной разности порядков полиномов делимого и делителя с коэффициентом, равными частному от деления коэффициента при старшей степени делимого на коэффициент делителя при старшей степени делителя; этот элемент умножают на делитель и результат вычитают из делимого.

С полученным разностным полиномом пониженного порядка все действия повторяют, получая второй и последующие элементы частного, пока его порядок не станет ниже порядка делителя - этот полином представляет собой остаток от деления.

Вычисление значений полиномов.

Вспомнив теорему Безу о том, что остаток от деления многочлена на двучлен $(z-a)$ равен значению полинома при $z=a$, можно использовать для вычисления значения полинома операцию деления на двучлен. Схему деления для этого частного случая можно упростить, используя a вместо $(z-a)$ и суммирование вместо вычитания, а также используя запись только коэффициентов без степеней z; от-

сутствующие степени обозначаются нулевыми коэффициентами. Такая схема вычислений известна как *схема Горнера*.

Вычисление корней полиномов.

Для решения этой часто возникающей на практике задачи предложено много методов и их количество говорит об отсутствии одного, обладающего убедительными преимуществами над другими. Наиболее общим естественно считать метод, позволяющий находить комплексные корни многочленов; при этом можно ограничиться подклассом многочленов с вещественными коэффициентами, так как именно они обычно возникают в практических расчетах.

Численные методы решения этой задачи строятся по одному и тому же шаблону: выбирается первое (при отсутствии априорных данных - произвольное) значение корня и вычисляется значение функции при этом значении; теперь стоит задача скорректировать текущее значение x_k так, чтобы соответствующее значение полинома оказалось ближе к нулю.

$$x_{k+1}=x_k+d.$$

Приходится определять направление и величину шага коррекции d.

Для нашего случая, когда левая часть нелинейного уравнения - полином $P(z)$, то есть легко дифференцируемая функция, наиболее подходящим будет видимо классический метод Ньютона или его модификации. Алгоритм Ньютона получается из простых геометрических соотношений - шаг коррекции можно определить как катет прямоугольного треугольника, другим катетом которого является значение полинома в текущей точке, а тангенс противолежащего ему угла есть производная полинома в той же текущей точке:

$$d=-P(z_k)/P^{(1)}(z_k),$$

$$z_{k+1}=z_k-P(z_k)/P^{(1)}(z_k), \quad \text{где } k \text{ - номер итерации.}$$

Можно вычислить производную только в точке первого приближения и использовать ее значение на всех последующих шагах, пожертвовав некоторым снижением скорости сходимости:

$$z_{k+1}=z_k-P(z_k)/P^{(1)}(z_0).$$

Неприятности могут ожидать нас на пологих участках функции, когда первая производная близка к нулевому значению - в этом случае слишком большой шаг коррекции «выбросит» нас в далекую

от корня область. Это особенно неприятно, если мы используем только однократное дифференцирование в точке первого приближения - случайное попадание на пологий участок в этом первом приближении вызовет эффект «рыскания» поисковой процедуры с большим шагом. Для устранения этого эффекта можно ограничить величину шага некоторым допустимым значением, например, делая его пропорциональным не первой производной, а той, которая достаточно далека от нулевого значения или использовать другие приемы ухода с пологого участка кривой. В этом случае алгоритм может выглядеть например так:

$$z_{k+1} = z_k + t\left(-\frac{P(z_k)}{P^{(j)}(z_k)}\right)^{\frac{1}{j}},$$

где j - порядок очередной производной с достаточно удаленным от нуля значением, t - коэффициент шага коррекции, выбираемый так, чтобы $P(z_{k+1})$ было меньше $P(z_k)$.

Перед началом поиска корней можно избавиться от кратных делением исходного полинома на его НОД с первой производной, но потом придется определять кратность корней, пока их количество не станет равным порядку полинома.

Методы определения корней для полиномов степеней ниже пятой, известные из курса элементарной алгебры, рассмотрены кратко при описании программной реализации.

Определение программного класса полиномов.

Рассмотрим структуру класса для работы с многочленами, алгоритмы для работы с объектами типа «полином» и примеры методов полиномиального класса, реализующие эти алгоритмы:

```
template <class YourOwnFloatType> class polynom:public vec-
            tor<YourOwnFloatType>
{
```

Если мы запишем коэффициенты полинома в порядке убывания, включая нулевые, мы получим упорядоченный кортеж длиной $n+1$: $(a_n, a_{n-1}, \ldots, a_2, a_1, a_0)$, то есть не что иное, как вектор размерности $n+1$, полностью характеризующий заданный полином. Поэтому вполне естественным является то, что наш полином будет базироваться на векторе.

Как и векторный класс, он будет параметризованным. При этом тип-параметр будет у нас относиться как к коэффициентам многочлена, так и к подставляемым в него значениям. Внутренний формат для хранения нашего полинома будет $a_0+a_1x+a_2x^2+a_3x^3+a_4x^4+a_5x^5+...+a_{n-1}x^{n-1}$, что обусловлено требованием удобства его индексирования, а внешнее представление будет в канонической форме - ввод и вывод будет производиться, начиная с коэффициента при наивысшей степени.

```
void optimize();
```

В процессе работы с полиномом мы можем прийти к ситуации, когда его порядок необходимо уменьшить в связи с тем, что обнулился коэффициент при старшей степени (или несколько подряд идущих коэффициентов). В связи с этим после каждой операции, которая может привести к изменению степени полинома, производится проверка, записан ли полином в канонической форме. Если нет, мы с помощью этой внутренней функции понижаем его порядок до тех пор, пока полином не будет преобразован в каноническую форму.

```
polynom<YourOwnFloatType> reverse();
```

В связи с различием внутреннего и внешнего представления полинома иногда бывает необходимо записать полином в обратном порядке.

```
void In(istream &);
void Out(ostream &);
```

По аналогии с векторами переопределим две виртуальные функции ввода и вывода. При этом отпадает необходимость в перегрузке операций потокового ввода-вывода: однажды определённые в векторном классе, они используют именно эти виртуальные функции, а определение, из какого именно класса необходимо их вызывать, осуществляется уже на этапе выполнения, в зависимости от того, к какому типу преобразуется базовый указатель. Так, любой полиномиальный объект можно преобразовать к векторному.

```
public:
```

В этом разделе мы разместим общедоступные дружественные функции и методы полиномиального класса.

```
polynom(char ·);
```

Полином из файла? Почему бы и нет! Параметром этого конструктора является имя файла, в котором находятся данные в виде [Степень Многочлена-1] [Свободный Член] [Коэффициент При Первой Степени] … [Коэффициент При Старшей Степени].

```
polynom(long,YourOwnFloatType ·);
```

Конструктор, принимающий два параметра - уменьшенную на единицу степень многочлена и указатель на данные - коэффициенты многочлена, начиная со свободного члена. Почему бы не передавать в этот конструктор просто степень многочлена? Просто показалось удобнее пользоваться понятием не «степень многочлена», а «размерность полинома», то есть количество коэффициентов, его составляющих.

```
polynom(polynom<YourOwnFloatType> &);
```

Конструктор копирования делает слепок с заданного полинома.

```
polynom();
```

Конструктор по умолчанию создаёт полином особого вида - нуль-полином, то есть многочлен размерности 1 (соответственно степени 0), свободный член которого равен нулевому элементу типа-параметра полиномиального класса.

```
polynom(long);
```

В некоторых случаях бывает полезно задать сначала степень полинома, но временно оставить неопределёнными его коэффициенты. Этот конструктор создаёт полином заданной размерности и обнуляет коэффициенты, нарушая каноническую форму записи многочлена. Это бывает необходимо редко и только в тех случаях, когда коэффициенты многочлена становятся известными после того, как задана его степень. Разумеется, и в этом случае параметром конструктора является степень многочлена-1.

```
polynom<YourOwnFloatType> operator-();
```

Полином, противоположный полиному $P(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots+a_2x^2+a_1x+a_0$, определяется как $-P(x)=-a_nx^n-a_{n-1}x^{n-1}-\ldots-a_2x^2-a_1x-a_0$, то есть знак коэффициентов при степенях многочлена меняется на противоположный.

```
polynom<YourOwnFloatType> operator+();
```

Унарный плюс - операция, просто возвращающая копию текущего многочлена.

```
friend polynom<YourOwnFloatType> operator+(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

Дружественная функция сложения многочленов принимает два параметра - многочлены-слагаемые, и возвращает результирующий многочлен-сумму. Суммой двух многочленов, $f(x)=a_nx^n+a_{n-1}x^{n-1}+...+a_2x^2+a_1x+a_0$ и $g(x)=b_mx^m+b_{m-1}x^{m-1}+...+b_2x^2+b_1x+b_0$ степеней n и m соответственно ($n \geq m$) называют многочлен $c_nx^n+c_{n-1}x^{n-1}+...+c_2x^2+c_1x+c_0$, где $c_i=a_i+b_i$ (i=0, 1, …, m), $c_i=a_i$ (i=m+1, …, n). Разумеется, после получения многочлена-суммы его необходимо записать в канонической форме.

```
friend polynom<YourOwnFloatType> operator+=(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

Дружественная функция, реализующая операцию сокращённого сложения двух многочленов, складывает первый многочлен со вторым и записывает результат в первый, возвращая его копию для дальнейших преобразований.

```
friend polynom<YourOwnFloatType> operator-(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

Так как операция сложения многочленов обратима, то мы можем определить вычитание многочленов, используя операции сложения и получения многочлена, противоположного к данному. Таким образом, разностью двух полином $f(x)$ и $g(x)$ является полином $P(x)$ такой, что

$$P(x)=f(x)+(-g(x)).$$

```
friend polynom<YourOwnFloatType> operator-=(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

По аналогии с сокращённым сложением, мы можем определить сокращённое вычитание как дружественную функцию, первый параметр которой модифицируется.

```
friend polynom<YourOwnFloatType> operator*(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

Произведением двух многочленов, $f(x)=a_nx^n+a_{n-1}x^{n-1}+...+a_2x^2+a_1x+a_0$ и $g(x)=b_mx^m+b_{m-1}x^{m-1}+...+b_2x^2+b_1x+b_0$, называют многочлен $c_n+_mx^n+^m+...+c_1x+c_0$, где

$$c_i = \begin{cases} a_i b_0 + a_{i-1} b_1 + \ldots + a_0 b_i, & i = 0, 1, \ldots, m; \\ a_i b_0 + a_{i-1} b_1 + \ldots + a_{i-m} b_m, & i = m+1, \ldots, n; \\ a_n b_{i-n} + a_{n-1} b_{i-n+1} + \ldots + a_{i-m} b_m, & i = n+1, \ldots, \mathrm{n}+m. \end{cases}$$

Кроме того, мы можем находить произведение многочленов по правилу умножения сумм:

$$f(x) \cdot g(x) = \sum_{i=0}^{n} a_i x^i \cdot \sum_{k=0}^{m} b_k x^k .$$

```
friend polynom<YourOwnFloatType> operator*=(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

Сокращённое умножение полинома на полином.

```
friend polynom<YourOwnFloatType> operator*( YourOwn-
FloatType, polynom<YourOwnFloatType> &);
friend polynom<YourOwnFloatType> operator*(polynom
<YourOwnFloatType> &, YourOwnFloatType);
```

Умножение числа на полином, равно как и умножение полинома на число - это операция, воздействующая на коэффициенты полинома, и соответствует умножению числа на вектор (вектора на скаляр), то есть:

$$\alpha f(x) = \alpha a_n x^n + \alpha a_{n-1} x^{n-1} + \ldots + \alpha a_2 x^2 + \alpha a_1 x + \alpha a_0.$$

```
friend polynom<YourOwnFloatType> operator*=(polynom
<YourOwnFloatType> &, YourOwnFloatType);
```

Сокращённое умножение полинома на число.

```
friend long operator<(polynom<YourOwnFloatType> &, poly-
    nom<YourOwnFloatType> &);
friend long operator>(polynom<YourOwnFloatType> &, poly-
    nom<YourOwnFloatType> &);
friend long operator<=(polynom<YourOwnFloatType> &, poly-
    nom<YourOwnFloatType> &);
friend long operator>=(polynom<YourOwnFloatType> &, poly-
    nom<YourOwnFloatType> &);
```

Набор операций для сравнения полиномов. Два полинома считаются равными, если они одинаковой степени и коэффициенты при соответствующих степенях x равны. В противном случае полномы не равны. При этом считается, что первый полином меньше второго, если в канонической форме его степень ниже (размерность меньше). Если же эти полиномы одинаковой размерности (a, соот-

ветственно, и степени), то мы последовательно сравниваем их коэффициенты, начиная со старшей степени. Первое различие в коэффициентах и определяет, какой знак необходимо поставить между этими многочленами.

```
YourOwnFloatType &operator[](long);
```

Индексация полинома - операция, принимающая в качестве параметра степень одночлена, коэффициент при котором необходимо вернуть.

```
YourOwnFloatType operator()(YourOwnFloatType);
```

Получить значение полинома в заданной точке мы можем, используя эту функцию. Её задача – либо просто просуммировать произведения коэффициентов многочлена на соответствующую степень аргумента данной функции, либо вічислить єто значение по теореме Безу.

```
friend long div(polynom<YourOwnFloatType> &, polynom<YourOwnFloatType> &, polynom<YourOwnFloatType> &,polynom<YourOwnFloatType> &);
```

Из курса алгебры известна теорема, гласящая о том, что для любых двух многочленов *f(x)* и *g(x)* существует единственная пара многочленов *l(x)* и *r(x)*, удовлетворяющих условию

$$f(x)=g(x)l(x)+r(x),$$

где степень *r(x)* меньше степени *g(x)* или-нуль-многочлен.

Для определения вида этих многочленов рассмотрим многочлены

$$f(x)=a_n x^n+a_{n-1}x^{n-1}+\ldots+a_2x^2+a_1x+a_0 \text{ и}$$
$$g(x)=b_m x^m+b_{m-1}x^{m-1}+\ldots+b_2x^2+b_1x+b_0,\ g(x)\neq 0.$$

1. Пусть $n<m$. Тогда справедливо равенство $f(x)=g(x)0+f(x)$, удовлетворяющее искомую пару многочленов.
2. Пусть $n\geq m$. В этом случае составим вспомогательный многочлен $f_1(x)=f(x)-g(x)\frac{a_n}{b_m}x^{n-m}$ степени n_1. Если $n_1\leq m$, то ограничиваемся одним этим многочленом, если же $n_1\geq m$, то строим $f_2(x)=f_1(x)-g(x)\frac{a'_{n_1}}{b_m}x^{n_1-m}$ (где a'_{n_1} - старший коэффициент многочлена $f_1(x)$). Если степень n_2 многочлена $f_2(x)$ меньше m, то ограничиваемся построенными двумя многочленами, если же

$n_2 \geq m$, то продолжаем (аналогично предыдущему) строить вспомогательные многочлены $f_3(x)$, $f_4(x)$, ..., $f_s(x)$, ... При этом степень каждого следующего многочлена меньше степени предыдущего, поэтому на определённом k-м шаге имеем многочлен $f_k(x) = f_{k-1}(x) - g(x)\frac{a^*_{n_{k-1}}}{b_m} x^{n_{k-1}-m}$, степень которого меньше m (или $f_k(x)=0$).

Сложим почленно полученные равенства:

$$f_1(x) + f_2(x) + ... + f_k(x) = f(x) + f_1(x) + ... + f_{k-1}(x) - g(x)\left(\frac{a_n}{b_m} x^{n-m} + \frac{a'_{n_1}}{b_m} x^{n_1-m} + ... + \frac{a^*_{n_{k-1}}}{b_m} x^{n_{k-1}-m} \right).$$

Выражение в скобках представляет собой сумму многочленов, а потому (по определению суммы) также является многочленом. Обозначим его через $l(x)$. Многочлен $f_k(x)$ обозначим $r(x)$. По построению, степень $r(x)$ меньше степени $f(x)$, или $r(x)=0$.

По аналогии, назовём $f(x)$ делимым, $g(x)$ - делителем, $l(x)$ - неполным частным, а $r(x)$ - остатком от деления $f(x)$ на $g(x)$.

```
friend polynom<YourOwnFloatType> operator/(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
friend polynom<YourOwnFloatType> operator%(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

Используя описанную выше функцию, определим две вспомогательные операторные функции - для нахождения целой части и остатка от деления.

```
long IsEqual(void ·);
```

Виртуальная функция с таким же именем, определённая в векторном классе, предназначена для сравнения текущего вектора с вектором, адрес которого передаётся через обобщённый указатель. Та же операция проделывается и для полиномов с одной единственной целью - использовать единожды, в векторном классе, определённые операторные функции == и !=, что увеличивает гибкость и универсальность данного класса за счёт использования механизма виртуальных функций.

```
polynom<YourOwnFloatType> operator=(polynom<YourOwnFloatType> &);
```

Присвоение полинома-параметра текущему.

```
friend polynom<YourOwnFloatType> derive(polynom <YourOwnFloatType>,long);
```

Полином, как функция аналитическая, удобен тем, что для него всегда можно найти аналитическую производную. Как известно, производной степенной функции является степенная же функция, поэтому производная от полинома будет тоже полином:

$$\frac{d}{dx}\left(a_n x^n + a_{n-1}x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0\right) = \\ na_n x^{n-1} + (n-1)a_{n-1}x^{n-2} + \ldots + 2a_2 x + a_1 .$$

Параметром данной функции является порядок производной - количество раз, которое полином дифференцируется.

```
friend polynom<YourOwnFloatType> integral(polynom <YourOwnFloatType>, long);
```

По аналогии с аналитической производной для полинома можно определить аналитический интеграл. Первообразная степенной функции - степенная же функция степени на единицу выше, поэтому интеграл от полинома также будет полиномом:

$$\int\left(a_n x^n + a_{n-1}x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0\right)dx = \\ \frac{a_n}{n+1}x^{n+1} + \frac{a_{n-1}}{n}x^n + \ldots + \frac{a_2}{3}x^3 + \frac{a_1}{2}x^2 + a_0 x .$$

Параметром данной функции является кратность интегрирования - количество раз, которое полином интегрируется.

```
friend polynom<YourOwnFloatType> pow(polynom <YourOwn-FloatType>, unsigned int);
polynom<YourOwnFloatType> operator^(unsigned int);
```

Используя введенную ранее операцию умножения, мы можем определить степень полинома как перегруженную операцию или как дружественную функцию.

```
polynom<YourOwnFloatType> operator^=(unsigned int);
```

Сокращённая степень.

```
};
```

От параметризованных полиномов, тип которых определяется подстановкой в классовые уголки, перейдём к комплексным полиномам вида

$$f(z)=a_nz^n+a_{n-1}z^{n-1}+...+a_2z^2+a_1z+a_0$$

подставив тип «комплексное число»:

```
typedef polynom<complex> cpolynom;
typedef vector<complex> cvector;
typedef matrix<complex> cmatrix;
```

Значения z, при подстановке которых многочлен обращается в нуль, называются *корнями* этого многочлена. Таким образом, корни $f(z)$ - это решения уравнения:

$$f(z)=a_nz^n+a_{n-1}z^{n-1}+...+a_2z^2+a_1z+a_0=0,$$

называемого алгебраическим уравнением n-й степени.

По теореме Безу, остаток, получаемый при делении многочлена $f(z)$ на $(z-a)$, равен $f(a)$. В частности, для того чтобы многочлен $f(z)$ делился на $(z-a)$ без остатка, необходимо и достаточно условие $f(a)=0$, т. е. для того чтобы многочлен делился на двучлена $(z-a)$ без остатка, необходимо и достаточно, чтобы $z=a$ было корнем этого многочлена.

Таким образом, зная корень $z=a$ многочлена $f(z)$, мы можем выделить из этого многочлена множитель $(z-a)$: $f(z)=(z-a)f_1(z)$, где $f_1(z)=b_{n-1}z^{n-1}+b_{n-2}z^{n-2}+...+b_2z^2+b_1z+b_0$ ($b_{n-1}=a_{n-1}$); нахождение остальных корней приводит к решению уравнения

$$b_{n-1}z^{n-1}+b_{n-2}z^{n-2}+...+b_2z^2+b_1z+b_0=0$$

$(n-1)$-й степени.

Продолжая этот процесс, мы получим окончательно следующее разложение $f(z)$ на множители: $f(z)=a_0(z-z_1)(z-z_2)...(z-z_n)$, т.е. всякий многочлен n-й степени разлагается на $(n+1)$ множителей, один из которых равен старшему коэффициенту, а остальные - двучлены первой степени вида $(z-a)$. При подстановке $z=z_s$ (s=1, 2,..., n) по крайней мере один из множителей в разложении обратится в нуль, т. е. значения $z=z_s$ суть корни $f(z)$.

Рассмотрим частные методы решения алгебраических уравнений в радикалах, а также метод Ньютона для нахождения всех корней комплексного полинома.

```
cvector square(complex,complex,complex);
cvector kardano(double,double,double,double);
cvector ferrary(double,double,double,double,double);
```

Пусть задано алгебраическое уравнение степени n коэффициентами определяющего его многочлена. Количество корней этого уравнения будет равно степени многочлена, и в общем случае эти

корни будут комплексными, поэтому возвращаемым значением функции, предназначенной для нахождения корней полинома, будет вектор размерности *n*, компоненты которого суть корни полинома.

Первая из рассматриваемых функций предназначена для решения алгебраического уравнения второй степени, или, как его ещё называют, квадратного уравнения. Параметрами его являются коэффициенты при второй, первой и нулевой степенях соответственно. Алгоритм решения такого уравнения определяется следующими соотношениями:

$$a_2x^2+a_1x+a_0=a_2\left(x^2+\frac{a_1}{a_2}x+\frac{a_0}{a_2}\right)=a_2\left(x^2+2\frac{a_1}{2a_2}x+\frac{a_1^2}{4a_2^2}-\frac{a_1^2}{4a_2^2}+\frac{a_0}{a_2}\right)=$$

$$=a_2\left(\left(x^2+2\frac{a_1}{2a_2}x+\frac{a_1^2}{4a_2^2}\right)-\frac{a_1^2}{4a_2^2}+\frac{a_0}{a_2}\right)=a_2\left(\left(x+\frac{a_1}{2a_2}\right)^2-\frac{a_1^2}{4a_2^2}+\frac{4a_0a_2}{4a_2^2}\right)=$$

$$=a_2\left(x+\frac{a_1}{2a_2}\right)^2+a_2\frac{4a_0a_2-a_1^2}{4a_2^2}=0\Rightarrow\left(x+\frac{a_1}{2a_2}\right)^2=-\frac{4a_0a_2-a_1^2}{4a_2^2}\Rightarrow$$

$$\Rightarrow\left(x+\frac{a_1}{2a_2}\right)^2=\frac{a_1^2-4a_0a_2}{4a_2^2}\Rightarrow\left|x+\frac{a_1}{2a_2}\right|=\sqrt{\frac{a_1^2-4a_0a_2}{4a_2^2}}\Rightarrow$$

$$\Rightarrow\left|x+\frac{a_1}{2a_2}\right|=\frac{\sqrt{a_1^2-4a_0a_2}}{2a_2}\Rightarrow x+\frac{a_1}{2a_2}=\pm\frac{\sqrt{a_1^2-4a_0a_2}}{2a_2}\Rightarrow$$

$$\Rightarrow x=-\frac{a_1}{2a_2}\pm\frac{\sqrt{a_1^2-4a_0a_2}}{2a_2}\Rightarrow x=\frac{-a_1\pm\sqrt{a_1^2-4a_0a_2}}{2a_2}$$

Для решения уравнений третьей и четвёртой степеней с комплексными коэффициентами общих методов нет. Однако существуют два частных метода для решения уравнений этих степеней с действительными коэффициентами, метод Кардано-Тартальи для решения уравнений третей степени и метод Феррари для решения уравнений четвёртой степени. Так как эти методы не охватывают все уравнения данного класса, мы на них останавливаться не будем, однако приведём ниже пример их реализации.

```
cvector newton(cpolynom);
```

Пусть необходимо найти все корни уравнения $P(z)=0$. Для численного решения можно воспользоваться итерационным методом Ньютона. Замечательной особенностью этого метода является его гарантированная сходимость на всей комплексной плоскости для выпуклых функций (а полиномы степени выше первой - функции выпуклые), причём скорость сходимости не зависит от начального приближения. Для начала итерации зададимся точностью ε, с которым нам необходимо найти решение, и начальным приближением к некоторому корню $z_0=x_0+iy_0$. Алгоритм вычислений может быть таким:

1. Находим производную полинома в точке z_k (k-номер итерации). Если она равна нулю, дифференцируем $P(z)$ дальше до тех пор, пока $P^{(j)}(z_k)$ не станет отличной от нуля.
2. Следующее приближение к корню находим по формуле $z_{k+1} = z_k + t\left(-\frac{P(z_k)}{P^{(j)}(z_k)}\right)^{\frac{1}{j}}$, где параметр t подбирается так, чтобы выполнялось условие: $|P(z_{k+1})|<|P(z_k)|$.
3. Проверяем выполнение условия $|P(z_{k+1})|<\varepsilon$; если условие не выполняется, переходим к пункту 1. Если условие выполняется, то z_{k+1} считаем корнем уравнения, полином $P(z)$ делится на двучлен $(z-z_k)$ и получаем полином $(n-1)$-ой степени, для которого повторяем все вычисления, начиная с п. 1.

Описанные алгоритмы, к примеру, можно реализовать так:

```
#ifndef __EQUATION_H
#define __EQUATION_H

//Этот файл включения содержит объявление двух типов-
//комплексного полинома и комплексного вектора, а также
//заголовки четырёх функций для решения уравнений 2-ой,
//3-ей, 4-ой и высших степеней+ функции нахождения
//собственных значений и собственных векторов


#ifndef __COMPLEX_H
#include <complex.h>
#endif
#ifndef __POLYNOM_H
```

```
#include "polynom.h"
#endif
#ifndef __VECTOR_H
#include "vector.h"
#endif
#ifndef __MATRIX_H
#include "matrix.h"
#endif

typedef polynom<complex> cpolynom;
typedef vector<complex> cvector;
typedef matrix<complex> cmatrix;

cvector square(complex,complex,complex);
cvector kardano(double,double,double,double);
cvector ferrary(double,double,double,double,double);
cvector newton(cpolynom);
cvector eigenval(cmatrix&);
cmatrix eigenvec(cmatrix&,cvector&);

#endif


#ifndef __EQUATION_H
#include "equation.h"
#endif
```
//Решение квадратного уравнения с комплексными коэффициентами
```
cvector square(complex a2,complex a1,complex a0)
{
  complex a=a2,b=a1,c=a0;
  //Найденные комплексные корни должны быть записаны
  //в двухкомпонентный комплексный вектор-результат
  cvector res(2);
  //проверим, не нулевой ли коэффициент при x^2
  if(a!=complex(0,0))
  {
    //если да, ищем корни через квадратичный дискриминант
    complex D=b*b-4*a*c;
    res[0]=(-b+sqrt(D))/(2*a);
      //и заносим их в соответствующие
    res[1]=(-b-sqrt(D))/(2*a);
      //компоненты вектора-результата
  }
```

```
  else
    res[0]=res[1]=-c/b;//иначе решаем уравнение первой степени
  return res;
}


/*Решение кубического уравнения с действительными ко-
эффициентами методом Кардано-Тартальи*/
cvector kardano(double a3,double a2,double a1,double
a0)
{
  //Общий вид уравнения, корни которого мы ищем -
  //a3*x^3+a2*x^2+a1*x+a0=0
  //так как уравнение имеет третью степень, то и корней у него три,
  cvector res=3;//что и определяет размерность вектора-результата
  if(a3==0)//если коэффициент при x^3 нулевой,
  {
    //решаем соответствующее квадратное уравнение
    cvector res2=square(a2,a1,a0);
    //в этом случае принимаем, что два корня являются совпадающими
    res[0]=res[1]=res2[0];
    res[2]=res2[1];
  }
  else
  {
    //иначе - приводим кубическое уравнение к каноническому
    //виду, когда коэффициент при третьей степени равен 1
    double a=a2/a3,b=a1/a3,c=a0/a3;
    //находим слагаемые кубического дискриминанта
    double p=b-a*a/3,q=2*a*a*a/27-a*b/3+c;
    //проведя переобозначение x=y-a/3, решаем в дальнейшем
    //уравнение y^3+p*y+q=0
    //находим кубический дискриминант
    double diskr=pow(q/2,2)+pow(p/3,3);
    //если дискриминант равен 0, то имеем 3 действительных
    //корня, из них два совпадающих
    if(diskr==0)
    {
      res[0]=3*q/p;
      res[1]=res[2]=-res[0]/2;
    }
    //один действительный корень и два комплексно сопряжённых
    if(diskr>0)
```

```
    {
      double what=-q/2+sqrt(diskr);
      double  u0=(what>0)?pow(what,1.0/3.0):-pow(-what,
1.0/3.0);
      double v0=-p/(3*u0);
      res[0]=u0+v0;
      res[1]=res[2]=-res[0]/2;
      complex k3(0,sqrt(3)*(u0-v0)/2);
      res[1]+=k3;
      res[2]-=k3;
    }
    //три различных действительных корня
    if(diskr<0)
    {
      cvector zkub12=square(1,q,-p*p*p/27);
      complex u0=pow(zkub12[0],1/3.);
      res[0]=2*real(u0);
      res[1]=-real(u0)-imag(u0)*sqrt(3);
      res[2]=-real(u0)+imag(u0)*sqrt(3);
    }
    //переходим обратно от y к x
    for(int i=0;i<3;i++)
      res[i]-=a/3;
  }
  return res;//возвращаем вектор-результат
}


//Метод Феррари для решения уравнений четвёртой степени
//с действительными коэффициентами
cvector  ferrary(double  a4,double  a3,double  a2,double
a1,double a0)
{
  cvector res=4;//всего корней будет 4
  if(!a4)//если коэффициент при четвёртой степени x нулевой,
         //пытаемся решить уравнение третьей степени
  {
    cvector res3=kardano(a3,a2,a1,a0);
    res[0]=res[1]=res3[0];
    for(int i=1;i<3;i++)
      res[i+1]=res3[i];
  }
  else
```

```
  {
    double a=a3/a4,b=a2/a4,c=a1/a4,d=a0/a4;
    //составляем и находим корни кубической резольвенты
    cvector    cy0=kardano(1,-b,-4*d+a*c,-d*a*a+4*b*d-
c*c);
    double y0;
    //ищем хотя бы один действительный корень,
    //с помощью которого сводим уравнение четвёртой
    //степени к совокупности квадратных
    for(int i=0;i<3;i++)
      if(!imag(cy0[i]))
      {
        y0=real(cy0[i]);
        break;
      }
    double A=a*a/4-b+y0,B=a*y0/2-c;
    double x12=-B/(2*A);
    //решаем квадратные уравнения
    cvector      sq1=square(1,a/2-sqrt(A),y0/2+sqrt(A)
*x12);
    cvector      sq2=square(1,a/2+sqrt(A),y0/2-sqrt(A)
*x12);
    for(long i=0;i<2;i++)
      res[i]=sq1[i],res[i+2]=sq2[i];
  }
  return res;//возвращаем результат
}


/*модифицированный метод Ньютона поиска комплексных
корней полинома*/
cvector newton(cpolynom p)
{
  double t=1,eps=1e-8,j;

  p*=1;//вместо p.optimize(); выполняем умножение на 1 для того,
       //чтобы отбросить ведущие нули
  //результирующий вектор будет иметь размерность,
  //равную порядку полинома
  cvector result=p.getm()-1;
  //если вектор-результат одномерный, то имеем дело с
  //полиномом первой степени
  if(result==cvector())
```

```
  {
    result[0]=-p[0]/p[1];
    return result;
  }
  //если двумерный - с квадратным уравнением и т.д.
  if(result.getm()==2)
    return square(p[2],p[1],p[0]);
  if(result.getm()==3&&!imag(p[3])&&!imag(p[2])&&
!imag(p[1])&&!imag(p[0]))
    return   kardano(real(p[3]),real(p[2]),real(p[1]),
real(p[0]));
  if(result.getm()==4&&!imag(p[4])&&!imag(p[3])&&
!imag(p[2])&&!imag(p[1])&&
     !imag(p[0]))
    return   ferrary(real(p[4]),real(p[3]),real(p[2]),
real(p[1]),real(p[0]));
  complex z=0;/*начальное приближение к корню положим
равным (0,0)*/
  do
  {
    complex dz=0;
    //находим порядок и значение ненулевой производной в точке z
    for(j=1;dz==complex(0);dz=derive(p,j++)(z));
    z+=t*pow(-p(z)/dz,1/(--j));//модифицируем приближение
    t*=(1-eps);
  }while(abs(p(z))>=eps);
 //повторяем до достижения заданной точности
 //В этом цикле находится только один корень z. Для нахождения
 //остальных делим исходный полином на x-z, понижая тем самым степень
 //на единицу, и находим ещё один корень, и т.д. до первой степени
  result[0]=z;
  cpolynom z1=2;
  z1[0]=-z,z1[1]=1;
  cvector more=newton(p/z1);
  for(long i=1;i<result.getm();i++)
    result[i]=more[i-1];
  return result;//возвращаем вектор результата
}
```

По приведённым алгоритмам работы с полиномиальными объектами можно составить, к примеру, такой интерфейс для данного класса:

```
#ifndef __POLYNOM_H
#define __POLYNOM_H
#ifndef __VECTOR_H
#include "vector.h"
#endif
#ifndef __IOSTREAM_H
#include <iostream.h>
#endif
#define max(a,b)    (((a) > (b)) ? (a) : (b))
#define min(a,b)    (((a) < (b)) ? (a) : (b))

/*
 Параметризованный класс для полиномов
 Внутренний формат:
 a0+a1*x+a2*x^2+a3*x^3+a4*x^4+a5*x^5+...+a(n-1)*x^(n-1)
 Ввод и вывод производится, начиная с коэффициента при
 наивысшей степени
*/
//Наш полином будет базироваться на векторе
template <class YourOwnFloatType>
class polynom:public vector<YourOwnFloatType>
{
  //эти функции являются внутренними
  void optimize();/*преобразование полинома в каноническую форму*/
  polynom<YourOwnFloatType> reverse();/*запись полинома в обратном порядке*/
  void In(istream &);//по аналогии с векторами - функции ввода
  void Out(ostream &);//                и вывода
 public:
  polynom(char *);//полином из файла
  polynom(long,YourOwnFloatType *);//полином из массива
  polynom(polynom<YourOwnFloatType>    &);/*конструктор копирования*/
  polynom();//конструктор по умолчанию
  polynom(long);//полином заданной степени
  polynom<YourOwnFloatType> operator-();//унарный минус
  polynom<YourOwnFloatType> operator+();//унарный плюс
```

```
  friend polynom<YourOwnFloatType> operator+(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
//сложение
  friend polynom<YourOwnFloatType> operator+=(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
//сокращённое сложение
  friend polynom<YourOwnFloatType> operator-(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
//вычитание
  friend polynom<YourOwnFloatType> operator-=(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
//сокращённое вычитание
  friend polynom<YourOwnFloatType> operator*(polynom
<YourOwnFloatType> &,  polynom<YourOwnFloatType> &);
//умножение полинома на полином
  friend polynom<YourOwnFloatType> operator*=(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
//сокращённое умножение на полином
  friend polynom<YourOwnFloatType> operator* (YourOwn-
FloatType, polynom<YourOwnFloatType> &); //умножение
числа на полином
  friend polynom<YourOwnFloatType> operator*(polynom
<YourOwnFloatType> &, YourOwnFloatType); //умножение
полинома на число
  friend polynom<YourOwnFloatType> operator*=(polynom
<YourOwnFloatType> &, YourOwnFloatType);
//сокращённое умножение на число
  //набор операций для сравнения полиномов
  friend long operator<(polynom<YourOwnFloatType> &,
      polynom<YourOwnFloatType> &);
  friend long operator>(polynom<YourOwnFloatType> &,
      polynom<YourOwnFloatType> &);
  friend long operator<=(polynom<YourOwnFloatType> &,
      polynom<YourOwnFloatType> &);
  friend long operator>=(polynom<YourOwnFloatType> &,
      polynom<YourOwnFloatType> &);
  YourOwnFloatType &operator[](long);//индексация полинома
  YourOwnFloatType       operator()(YourOwnFloatType);
//значение полинома в заданной точке
  friend long div(polynom<YourOwnFloatType> &, poly-
nom<YourOwnFloatType>&, polynom<YourOwnFloatType> &,
polynom<YourOwnFloatType> &);//деление
```

```
  friend  polynom<YourOwnFloatType>  operator/(polynom
<YourOwnFloatType>  &,  polynom<YourOwnFloatType>  &);
//целая часть от деления
  friend  polynom<YourOwnFloatType>  operator%(polynom
<YourOwnFloatType>  &,  polynom<YourOwnFloatType>  &);
//остаток от деления
  long IsEqual(void *);//проверка на равенство
  polynom<YourOwnFloatType> operator=
      (polynom<YourOwnFloatType> &);//присвоение
  friend polynom<YourOwnFloatType> derive
      (polynom<YourOwnFloatType>,long);//производная
  friend polynom<YourOwnFloatType> integral
      (polynom<YourOwnFloatType>,long);//интеграл
  friend polynom<YourOwnFloatType> pow
      (polynom<YourOwnFloatType>,unsigned
      int);//степень
  polynom<YourOwnFloatType>  operator^(unsigned  int);
//степень как операция
  polynom<YourOwnFloatType>  operator^=(unsigned  int);
//сокращённая степень
};
```

//конструкторы полинома будут аналогичны конструкторам вектора

```
//полином из файла
template <class YourOwnFloatType>
polynom<YourOwnFloatType>::polynom(char
*f):vector<YourOwnFloatType>(f)
{
}


//полином степени a-1
template <class YourOwnFloatType>
polynom<YourOwnFloatType>::polynom(long
a):vector<YourOwnFloatType>(a)
{
}


//нуль-полином
template <class YourOwnFloatType>
```

```
polynom<YourOwnFloatType>::polynom():                vec-
tor<YourOwnFloatType>()
{
}
```

//полином (a-1)-ой степени с коэффициентами из массива v

```
template  <class  YourOwnFloatType>  polynom  <YourOwn-
FloatType>::polynom(long              a,YourOwnFloatType
*v):vector<YourOwnFloatType>(a,v)
{
}
```

```
//конструктор копирования
template  <class  YourOwnFloatType>  polynom  <YourOwn-
FloatType>::polynom(polynom<YourOwnFloatType>
&ex):vector<YourOwnFloatType>(ex)
{
}
```

```
/*для индексации полинома вызываем соответствующий ме-
тод векторного класса*/
template  <class  YourOwnFloatType>  YourOwnFloatType
&polynom<YourOwnFloatType>::operator[](long a)
{
  return (*(vector<YourOwnFloatType>*)this)[a];
}
```

```
//вычисление значения полинома в точке x
template  <class  YourOwnFloatType>  YourOwnFloatType
polynom<YourOwnFloatType>::operator()(YourOwnFloatType
x)
{
  YourOwnFloatType temp=0,px=1;
  for(long i=0;i<getm();i++,px*=x)
    temp+=(*this)[i]*px;
  return temp;
}
```

```
//унарный минус
template <class YourOwnFloatType>
polynom<YourOwnFloatType>                              poly-
nom<YourOwnFloatType>::operator-()
{
  return    *(polynom*)&(-(*(vector<YourOwnFloatType>*)
this));
}


//унарный плюс - просто возвращаем себя
template <class YourOwnFloatType>
polynom<YourOwnFloatType>                              poly-
nom<YourOwnFloatType>::operator+()
{
  return *this;
}


//сложение двух полиномов
template  <class  YourOwnFloatType>  polynom  <YourOwn-
FloatType>          operator+(polynom<YourOwnFloatType>
&f,polynom<YourOwnFloatType> &s)
{
  long ms=max(f.getm(),s.getm());
  polynom<YourOwnFloatType> temp(ms);
  //создаём временный полином
  //здесь работают операции индексирования для всех трёх полиномов
  for(long i=ms-1;i>=min(f.getm(),s.getm());i--)
    temp[i]=(f.getm()>s.getm())?f[i]:s[i];
  for(long i=min(f.getm(),s.getm())-1;i>=0;i--)
    temp[i]=f[i]+s[i];
  temp.optimize();
  //пока есть, удаляем 0-коэффициент при старшей степени
  return temp;
}


//сокращённое сложение, определяемое через обычное
template  <class  YourOwnFloatType>  polynom  <YourOwn-
FloatType>  operator+=(polynom  <YourOwnFloatType>  &f,
polynom <YourOwnFloatType> &s)
{
```

```
  return f=f+s;
}


//вычитание полиномов, выраженное через сложение и отрицание
template      <class      YourOwnFloatType>      poly-
nom<YourOwnFloatType>    operator-(polynom    <YourOwn-
FloatType> &f, polynom<YourOwnFloatType> &s)
{
  return f+(-s);
}


//сокращённое вычитание
template      <class      YourOwnFloatType>      poly-
nom<YourOwnFloatType>   operator-=(polynom   <YourOwn-
FloatType> &f, polynom<YourOwnFloatType> &s)
{
  return f=f-s;
}


//умножение полиномов
template   <class   YourOwnFloatType>   polynom<class
YourOwnFloatType>  operator*(polynom<YourOwnFloatType>
&f,polynom<YourOwnFloatType> &s)
{
  polynom<YourOwnFloatType> temp(f.getm()+s.getm());
  for(long i=0;i<f.getm();i++)
    for(long j=0;j<s.getm();j++)
      temp[i+j]+=f[i]*s[j];
  temp.optimize();
  return temp;
}


//сокращённое умножение
template   <class   YourOwnFloatType>   polynom<class
YourOwnFloatType> operator*=(polynom<YourOwnFloatType>
&f,polynom<YourOwnFloatType> &s)
{
  return f=f*s;
}
```

```
//умножение числа на полином
template      <class      YourOwnFloatType>      poly-
nom<YourOwnFloatType>       operator*(YourOwnFloatType
ld,polynom<YourOwnFloatType> &v)
{
  polynom<YourOwnFloatType> temp=v;
  temp=*(polynom<YourOwnFloatType>*)
       &((*(vector<YourOwnFloatType>*)(&temp))*ld);
  temp.optimize();
return temp;
}


//умножение полинома на число
template  <class  YourOwnFloatType>  polynom  <YourOwn-
FloatType>          operator*(polynom<YourOwnFloatType>
&v,YourOwnFloatType ld)
{
  return ld*v;
}


//сокращённое умножение на число
template  <class  YourOwnFloatType>  polynom  <YourOwn-
FloatType>    operator*=(polynom    <YourOwnFloatType>
&v,YourOwnFloatType ld)
{
  return v=v*ld;
}


//присвоение
template  <class  YourOwnFloatType>  polynom  <YourOwn-
FloatType>          polynom<YourOwnFloatType>::operator
=(polynom<YourOwnFloatType> &x)
{
  //присваиваем как вектора
  (*(vector<YourOwnFloatType>*)this)=(*(vector
<YourOwnFloatType>*)&x);
  optimize();//приводим к нормальному виду
  return *this;//и возвращаем результат
```

```
}


//установление актуальной степени многочлена
template <class YourOwnFloatType>
void polynom<YourOwnFloatType>::optimize()
{
  if(getm()!=1)//если полином не нулевой степени
  {
    if((*this)[getm()-1]==(YourOwnFloatType)0)/*если
коэффициент при наивысшей степени нулевой*/
    {
      polynom<YourOwnFloatType>        temp(getm()-1);
//создаём временный полином степени на 1 меньше
      for(long i=0;i<getm()-1;i++)
        temp[i]=(*this)[i];
      *this=temp;//переписываем его в текущий
      optimize();//снова проверяем полином
    }
  }
}


//в очень редких случаях может быть необходимым реверсировать полином
template <class YourOwnFloatType>
polynom<YourOwnFloatType>                          poly-
nom<YourOwnFloatType>::reverse()
{
  polynom temp=*this;
  for(long i=0;i<getm();i++)
    temp[i]=(*this)[getm()-i-1];
  return temp;
}


//неотрицательная степень полинома как функция
template <class YourOwnFloatType>
polynom<YourOwnFloatType>
pow(polynom<YourOwnFloatType> x,unsigned int p)
{
  polynom<YourOwnFloatType> temp;
  if(p==0)
    temp[0]=1;//полином в нулевой степени - это просто число 1
```

```
  else
  {
    temp=x;
    for(long i=0;i<p-1;i++)
      temp*=x;
  }
  return temp;//возвращаем результирующий полином
}


//производная j-го порядка
template <class YourOwnFloatType>
polynom<YourOwnFloatType>                                    de-
rive(polynom<YourOwnFloatType> p, long j)
{
  if(j==0)//нулевая производная полинома есть сам полином
    return p;
  if(j<0)//отрицательная производная интерпретируется как интеграл
    return integral(p,-j);
  if(p.getm()==1)
    return polynom<YourOwnFloatType>();
    //если более понижать некуда
  polynom<YourOwnFloatType> result=p.getm()-1;
  for(long i=0;i<result.getm();i++)
  //вычисляем первую производную
    result[i]=(i+1)*p[i+1];
  if(j==1)//если её и надо было найти -
    return result;//возвращаем результат
  else//иначе
    return derive(result,j-1);
    //вычисляем производную от данной
}


//интеграл от полинома
template <class YourOwnFloatType>
polynom<YourOwnFloatType>                                  inte-
gral(polynom<YourOwnFloatType> p, long j)
{
  if(j<=0)
  //отрицательная кратность интегрирования трактуется как производная
    return derive(p,-j);
  polynom<YourOwnFloatType> result=p.getm()+1;
```

```
  for(long i=0;i<p.getm();i++)
    result[i+1]=p[i]/(i+1);
  if(j==1)
    return result;
  else
    return integral(result,j-1);
}


//степень как операция
template <class YourOwnFloatType>
polynom<YourOwnFloatType>
polynom<YourOwnFloatType>::operator^(unsigned int p)
{
  return pow(*this,p);
}


//сокращённая степень
template <class YourOwnFloatType>
polynom<YourOwnFloatType>
polynom<YourOwnFloatType>::operator^=(unsigned int p)
{
  return (*this)=pow(*this,p);
}
```

/*Перегружать операторы ввода из потока и вывода в поток необходимости нет - достаточно перегрузить две виртуальные функции ввод полинома из потока, начиная с коэффициентов при старших степенях*/

```
template <class YourOwnFloatType>
void polynom<YourOwnFloatType>::In(istream &is)
{
  for(long i=getm()-1;i>=0;i--)
    is>>(*this)[i];
  optimize();
}
```

//вывод полинома в поток, начиная с коэффициентов при старших степенях

```
template <class YourOwnFloatType>
void polynom<YourOwnFloatType>::Out(ostream &os)
{
```

```
  for(long i=getm()-1;i>=0;i--)
  {
    os.precision(100);
    //устанавливаем огромную точность вывода
    os<<(*this)[i]<<" ";//компоненты разделяем пробелами
  }
}


//сравнение текущего полином с лежащим по адресу x
template <class YourOwnFloatType>
long polynom<YourOwnFloatType>::IsEqual(void *x)
{
  polynom<YourOwnFloatType>
test1=*(polynom<YourOwnFloatType>*)x, test2=(*this);
  test1.optimize();
  test2.optimize();
  return ((vector<YourOwnFloatType>*)&test1)->
      vector<YourOwnFloatType>::IsEqual(&test2);
}


/*
  Из двух многочленов одинаковой длины меньше тот, у
которого коэффициент при старшей степени меньше. При
равенстве рассматриваем более низкую степень и т.д.
*/
template <class YourOwnFloatType>
long   operator<(polynom<YourOwnFloatType>   &f,polynom
<YourOwnFloatType> &s)
{
  polynom<YourOwnFloatType> test1=f,test2=s;
  test1.optimize();
  test2.optimize();
  if(test1.getm()<test2.getm())
    return 1;
  if(test1.getm()>test2.getm()||test1==test2)
    return 0;
  //точно не равны - выясняем, кто же из них меньше
  for(long i=test1.getm()-1;i>=0;i--)
    if(test1[i]<test2[i])
      return 1;
    else
```

```
      if(test1[i]>test2[i])
        return 0;
  return 0;
}
```

//все остальные операции определяем через уже известные

```
template <class YourOwnFloatType>
long   operator>(polynom<YourOwnFloatType>   &f,polynom
<YourOwnFloatType> &s)
{
  return (!(f<s))&&(f!=s);
}


template <class YourOwnFloatType>
long  operator<=(polynom<YourOwnFloatType>  &f,polynom
<YourOwnFloatType> &s)
{
  return (f<s)||(f==s);
}


template <class YourOwnFloatType>
long  operator>=(polynom<YourOwnFloatType>  &f,polynom
<YourOwnFloatType> &s)
{
  return (f>s)||(f==s);
}


/* деление полинома на полином по алгоритму Евклида */
template  <class  YourOwnFloatType>  long  div(polynom
<YourOwnFloatType>  &f,polynom<YourOwnFloatType>  &g,
polynom<YourOwnFloatType> &l,polynom<YourOwnFloatType>
&r)
{
  polynom<YourOwnFloatType> tf=f,tg=g;
  tf.optimize();
  tg.optimize();
  if(tg==polynom<YourOwnFloatType>())
  //попытка деления на ноль
    throw xmsg("Попытка деления на 0\n");
```

```
  if(tf.getm()<tg.getm())
  {
    l=polynom<YourOwnFloatType>();
    r=tf;
    return 1;
  }
  for(l=polynom<YourOwnFloatType>();tf.getm()>=
tg.getm();)
  {
    polynom<YourOwnFloatType> x(2);
    x[1]=1;
    x^=tf.getm()-tg.getm();
    x*=tf[tf.getm()-1]/tg[tg.getm()-1];
    r=tf-=tg*x;
    l+=x;
  }
  return 1;
}


//целая часть от деления
template <class YourOwnFloatType> polynom <YourOwn-
FloatType>   operator/(polynom<YourOwnFloatType>   &f,
polynom<YourOwnFloatType> &s)
{
  polynom<YourOwnFloatType> l,r;
  div(f,s,l,r);
  return l;
}


//остаток от деления
template <class YourOwnFloatType> polynom <YourOwn-
FloatType>   operator%(polynom<YourOwnFloatType>   &f,
polynom<YourOwnFloatType> &s)
{
  polynom<YourOwnFloatType> l,r;
  div(f,s,l,r);
  return r;
}


#endif
```