Примечания
1. Сравнительная характеристика процедурного и объектного подхода при создании программных систем. Сущность и преимущества объектного подхода.
2. Понятие объекта. Возможные отношения между объектами.
3. Понятия класса. Возможные отношения между классами. (Гражевский)
4. Сущность абстрагирования. Объявление классов в C++. Создание и уничтожение объектов
5. Создание и начальная инициализация данных объекта. Назначение и особенности использования конструкторов и деструкторов
6. Сущность инкапсуляции. Реализация и сокрытие данных в C++. Особенности доступа к открытым и закрытым элементам и методам элементам класса
7. Понятие интерфейса класса. Принцип отделения интерфейса класса от его реализации. Классификация методов класса
8. Классификация конструкторов класса. Назначение и характеристика разных типов конструкторов. (Белоцерковский)
9. Отношения дружественности в объектно-ориентированном программировании. Дружественные функции и классы.
10. Статические элементы и методы класса.
11. Константные элементы и методы класса.
12. Понятия перегрузки операций. Основные особенности перегрузки операций в C++
13. Особенности перегрузки операций присваивания и индексирования
14. Особенности перегрузки операций >> и << как операций ввода и вывода
15. Отношения наследования между классами. Реализация в C++ открытого одиночного наследования
16. Отношения наследования и защищенные элементы класса. Сравнительная характеристика разных уровней доступа к элементам и методам класса
17. Базовые и производные классы. Использование и назначение методов базового класса в производных классах
18. Понятие и суть полиморфизма. Статическое (раннее) и динамическое (позднее) связывание.(Северина)
19. Поддержка полиморфизма в C++. Виртуальные функции.
20. Чисто виртуальные функции. Абстрактные базовые классы.
21. Особенности и порядок создания программ с визуальным интерфейсом в среде Visual Studio 2012/2013. (Карпенко)
22. Шаблоны классов. Правила создания шаблонов классов. (Макаревич)
Сущности, используемые в практических заданиях.
    1. Фигуры в пространстве (есть).

Примечания

Примечание. Функции-элементы везде заменила на методы. Это короче, и это признанное в ООП обозначение.

Когда пишем название темы билета, выделяем это название и выбираем формат - Heading 1 (Заголовок 1). Это нужно, чтобы потом можно было сделать автоматическое содержание и быстро перемещаться к нужным вопросам. Весь остальной текст самого билета - Normal text(Обычный текст)

1. Сравнительная характеристика процедурного и объектного подхода при создании программных систем. Сущность и преимущества объектного подхода.

Это тупо из конспекта лекций с ДО:

Процедурная декомпозиция - это разделение алгоритма на модули, при котором каждый из модулей выполняет один из этапов общего процесса.

ООП-декомпозиция - это представление программы в виде совокупности взаимодействующих объектов.

Суть процедурного программирования: все программы - набор процедур и функций,выполняя которые, компьютер модифицирует значения переменных в памяти. Основная программа - это тоже процедура, которая может вызывать другие процедуры и функции. Суть процедурного программирования: **данные отдельно, поведение отдельно**. Плюсы такого подхода:

- позволяет выделять повторно используемые фрагменты кода
- делает код структурированным

Процедурный стиль выражается в четком выделении в программе двух составляющих: непроцедурной (описание переменных, функций, процедур) и процедурной (описание действий). Существенный недостаток процедурного программирования — сложная реализация параллельного программирования. Также как недостаток можно отметить существенный объем программного кода.

Суть ООП: объект объединяет данные и поведение. В ООП объектом является все кроме операторов.То есть ресурсы операционной системы, сетевые протоколы, формы, кнопки интрфейса - это всё объекты.ООП является следующей ступенью развития процедурного подхода.
Главная проблема с процедурным подходом заключается в том, что программные модули не откражают в достаточной мере сущности реального мира, поэтому их трудно использовать повторно (каждый раз прогер начинат по сути с нуля).
Теперь Губинское:
ООП - это методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром соответствующего класса, а классы могут образовывать иерархию наследования, то есть находиться между собой в отношениях “являться”.
Для ООП-стиля концпетуальная база - это объектная модель. Объектная модель имеет следующие наиболее важные элементы:

- абстрагирование (sex)
- инкапсуляция (drugs)
- наследование, полиморфизм (rock’n’roll)

Преимущества ООП (из сети, в конспекте нет):
(+ можно добавить преимущества из преимуществ инкапсуляции)

- Классы позволяют проводить конструирование из полезных компонент (уже готовых классов - например тех, что есть в библиотеке), обладающих простыми инструментами, что дает возможность абстрагироваться от деталей реализации.
- Данные и операции вместе образуют определенную сущность и они не «размазываются» по всей программе, как это нередко бывает в случае процедурного программирования.
- Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
- Инкапсуляция информации защищает наиболее критичные данные от несанкционированного доступа.
- ООП дает возможность создавать расширяемые системы (extensible systems). Это одно из самых значительных достоинств ООП и именно оно отличает данный подход от традиционных методов программирования. Расширяемость (extensibility) означает, что существующую систему можно заставить работать с новыми компонентами, причем без

внесения в нее каких-либо изменений. Компоненты могут быть добавлены на этапе выполнения.

2. Понятие объекта. Возможные отношения между объектами.

Объект - это конкретный опознаваемый предмет или конкретная сущность, имеющая физические (пространственные и временные) или концептуальные границы. Конкретная сущность может быть реальной либо абстрактной.
Границы объекта позволяют отделить один объект от другого, при этом же они являются в той или иной мере размытыми.
Объект обладает **состоянием, поведением (функциональностью), и идентичностью (означает, что совпадение всех свойств двух объектов не влечёт за собой совпадение объектов).**
Состояние объекта-перечень (обычно статических) его свойств и текущие (обычно динамических) значений каждого из этих свойств.
Поведение объекта-то, как объект действует и реагирует на поведение других объектов.
Состояние объекта является суммарным результатом его поведения, то есть объект, действуя либо подвергаясь воздействию других объектов, меняет свое состояние.
Возможна ситуация, когда у двух объектов, с идентичной структурой, совпадают значения всех свойств.
Отношения между объектами:
Между объектами может существовать отношение «быть частью» «конкретная рука»-«конкретный человек»
Между объектами не может быть установлено отношение «являться» в силу их конкретности и идентичности.
Между объектами можно установить отношение **«быть частью», part of** (предполагает, что части не могут существовать отдельно от целого) (банкомат содержит большое количество узлов (атрибутов)) и **отношение ассоциации**(выражает отношение между несколькими равноправными объектами и может иметь направление, роли и кратность, а также изображаться в виде класса ассоциации. Например, пользователь использует компьютер, разработчик работает на компанию, а компания пользуется результатами разработчика).

3. Понятия класса. Возможные отношения между классами. (Гражевский)

Класс-нечто общее , присущее некоторому множеству объектов, обладающих общей структурой и общим поведением. Объект при этом является экземпляром класса.
Конкретный класс может рассматриваться как объект, причем у этого объекта могут быть свои объекты.

Классы могут находится между собой в некоторых отношениях, таких как
1) отношение «являться» “is-a” «общее-частное» «человек» и «млекопитающее». Это же отношение можно назвать иерархией наследования.
2) отношение «быть частью» “part-of” «колесо» и «машина»
3) отношение «ассоциации» –«книга»- «автор».

4. Сущность абстрагирования. Объявление классов в C++. Создание и уничтожение объектов

Абстагирование - упрощённое представление чего бы то ни было, при котором одни свойства и детали выделяются, а другие опускаются. Хорошая абстракция подчеркивает существенные для решения задачи, опуская те, которые на данный момент не существенны. Классом, применительно к программированию, называют абстрактный тип данных.

```
1   // объявление классов в C++
2   class /*имя класса*/
3   {
4     private:
5     /* список свойств и методов для использования внутри класса */
6     public:
7     /* список методов доступных другим функциям и объектам программы */
8     protected:
9     /*список средств, доступных при наследовании*/
10  };
```

Объявление класса начинается с зарезервированного ключевого слова **class**, после которого пишется имя класса. В фигурных скобочках, **строки 3 — 10** объявляется тело класса, причём после закрывающейся скобочки обязательно нужно ставить точку с запятой, **строка 10**.
При создании объекта, лучше не копировать память для него, а выделять ее в в куче с помощью указателя. И освобождать ее после того, как мы закончили работу с объектом.
При создании статического объекта, для доступа к его методам и свойствам, используют операция прямого обращения — «.» (символ точки). Если же память для объекта выделяется посредством указателя, то для доступа к его методам и свойствам используется оператор косвенного обращения — «->».

```
// Выделение памяти для объекта Students
 Students *student = new Students;
// Удаление объекта student из памяти
delete student;
```

Создание и уничтожение объектов

Конструкторы и деструкторы - особые члены класса, служащие для инициализации и уничтожения объекта. Конструктор предназначен для инициализации данных класса в момент создания объектов соответствующего класса. Имя конструктора совпадает с именем класса. Конструктор не может иметь типа возвращаемого значения, соответственно не может возвращать значения. В момент создания объекта конструктор вызывается автоматически. Конструктор может быть многократно перегружен. Деструктор – специальная функция, которую вызывает программа вызывает автоматически каждый раз при уничтожении объкта. Деструктор имеет тоже имя, что и имя класса, но перед именем ставится знак ~.

5. Создание и начальная инициализация данных объекта. Назначение и особенности использования конструкторов и деструкторов

ДАРИК ,НАДО ПРОВЕРИТЬ

Для поддержки инициализации служит конструктор – специальная функция, предназначенная для инициализации данных класса в момент создания объектов соответствующего класса.

-В языке C++ имя конструктора совпадает с именем класса

-Конструктор не может иметь типа возвращаемого значения, не может возвращать значения.

-В случае, если класс не имеет конструкторов, конструктор по умолчанию вызывается средой автоматически.

-Если класс имеет хотя бы один конструктор - конструктор по умолчанию в классе должен быть объявлен явно, если предполагается его использование

-Функция «конструктор» может быть многократно перегружена

Типы конструкторов (с реализацией на примере сущбности рациональная дробь):

1)Конструктором по умолчанию называется конструктор, который можно вызывать, не задавая аргументов.

```
Drob ()
                { ch = 0; zn = 1; }
```

2)Инициализирующий конструктор

```
Drob (int p1, int p2)
                {if (p2==0)
{ch = 0;  zn = 1;}
else
 {ch=p1;zn=p2;}}
```

3)Конструктор преобразования – конструктор с единственным параметром. Он задает преобразование от типа параметра типу соответствующего класса

```
Drob ()
```

{ ch = p1; zn = 1; }

4)Копирующий конструктор – конструктор с параметром того же типа. Необходим для того, что бы создавать объекты на базе уже существующих объектов того же типа

```
Drob (Drob&d)
        { ch = d.ch;
          zn = d.zn; }
```

Парная по отношению к конструктору функция, деструктор, автоматически применяется к каждому объекту класса по окончании его использования и предназначена для освобождения ресурсов, захваченных либо в конструкторе класса, либо на протяжении его жизни. Имя этой функции образовано из имени класса с предшествующим символом "тильда" (~). Деструктор не возвращает значения и не принимает никаких параметров, а следовательно, не может быть перегружен. Хотя разрешается определять несколько таких функций-членов, лишь одна из них будет применяться ко всем объектам класса. Вот, например, деструктор для нашего класса Drob

```
~Drob ()
```

Но функции деструктора не ограничены только освобождением ресурсов. Он может реализовывать любую операцию, которая по замыслу проектировщика класса должна быть выполнена сразу по окончании использования объекта. Так, широко распространенным приемом для измерения производительности программы является определение класса Timer, в конструкторе которого запускается та или иная форма программного таймера. Деструктор останавливает таймер и выводит результаты замеров.

6. Сущность инкапсуляции. Реализация и сокрытие данных в C++. Особенности доступа к открытым и закрытым элементам и методам элементам класса

Инкапсуляция (сокрытие реализации) – отделение элементов объекта, определяющих их устройство от элементов определяющиъ его поведения. Устройство автомобиля инкапсулированно от водителя. В ООП инкапсуляция проявляется в разделении класса на интерфейс и реализацию. Интерфейс определяет внешнее поведение всех объектов данного класса, а реализация обеспечивает достижения желаемого поведения.
Преимущества инкапсуляции:
1. Позволяет той части программы, где используются объекты некоторого класса, никак не зависить от той части программы где класс реализован. Таким образом упрощается модификация приложения.
2. Пользователю класса нет необходимости перегружать себя информацией о том, как реализованы функции класса.
3. Закрытые данные класса защищены от некорректной работы с ними. Раздел public (открытый) предназначен для объявлений элементов и методов класса, которые доступны для внешнего использования. Раздел private (закрытый), содержит объявления

полей, процедур и функций, используемых только внутри данного класса. Раздел protected (защищенный) содержит объявления, доступные только для потомков объявляемого класса. Как и в случае закрытых элементов, можно скрыть детали реализации защищенных элементов от конечного пользователя. Однако в отличие от закрытых, защищенные элементы остаются доступны для программистов, которые захотят производить от этого класса производные объекты, причем не требуется, чтобы производные объекты объявлялись в этом же модуле.

```
 class A
{
public:
   int a, b; //данные открытого интерфейса
   int ReturnSomething(); //метод открытого интерфейса
private:
   int Aa, Ab; //скрытые данные
   void Do_Something(); //скрытый метод
};
```

Данные класса называются свойствами, полями, членами, элементами, атрибутами класса. Функции класса наз. функциями-членами, методами класса. Закрытые элементы класса видны только в области видимости класса. Вне класса доступа к закрытым элементам класса нет. Открытые элементы класса(обычно функции) видны и вне области видимости "класса". Для доступа к закрытым данным класса используют открытые функции доступа и инициализации(откр. функция Init).

Доступ к элементам или полям класса можно осуществить тремя способами:

- имя представителя класса.имя элемента;
- имя класса: :имя элемента;
- указатель на представитель класса->имя элемента

7. Понятие интерфейса класса. Принцип отделения интерфейса класса от его реализации. Классификация методов класса

ДАРИК ,НАДО ПРОВЕРИТЬ

Интерфейс класса - конструкция, определяющая методы и свойства, предоставляемые классом. Реализация класса - это способ осуществления работоспособности класса. Отделение интерфейса от реализации класса выполняется для того, чтобы скрыть способ осуществления работоспособности класса. Это – проявление инкапсуляции в ООП.

Отделение интерфейса от реализации выполняется за 5 шагов:

1.добавить в проект заголовочный файл *.h; 2.определить интерфейс класса в заголовочном файле 3.добавить в проект исполняемый файл *.cpp; 4.в исполняемом файле выполнить реализацию класса; 5.подключить заголовочный файл к программе.

Классификация:

1)конструкторы(специальная функция, предназначенная для инициализации данных класса в момент создания объектов соответствующего класса.)
2)деструктор(автоматически применяется к каждому объекту класса по окончании его использования и предназначена для освобождения ресурсов, захваченных либо в конструкторе класса, либо на протяжении его жизни.)
3)функции доступа. Имена этих функций нач. с get int getCh() {return ch;} 4)ф-ии инициализации(Init, set)

```
void set Ch(int p)
{ch=p;}
void set Zn(int p)
{if (p!=0) zn=p;
  else zn=1;}
void Init (int p1, int p2)
{set Ch(p1); set Zn(p2);}
Drob()
{Init (0,1);}
Drob (int p1; int p2)
{Init(p1, p2);}
```

- упрощающий конструктор.
5)набор функций реализующих фукциональность класса (Sum, Print).
6)функции перегружающие операции.
7)функции утилиты(вспомогательные функции, могут использоваться другими функциями класса.) обычно помещаются в раздел “private”

8. Классификация конструкторов класса. Назначение и характеристика разных типов конструкторов. (Белоцерковский)
Для поддержки инициализации служит конструктор – специальная функция, предназначенная для инициализации данных класса в момент создания объектов соответствующего класса.
-В языке C++ имя конструктора совпадает с именем класса
-Конструктор не может иметь типа возвращаемого значения, не может возвращать значения.
-В случае, если класс не имеет конструкторов, конструктор по умолчанию вызывается средой автоматически.
-Если класс имеет хотя бы один конструктор - конструктор по умолчанию в классе должен быть объявлен явно, если предполагается его использование
-Ф-я «конструктор» может быть многократно перегружена
Типы конструкторов:1)Конструктором по умолчанию называется конструктор, который можно вызывать, не задавая аргументов.

```
Drob ()
        { ch = 0; zn = 1; }
```

2)Инициализирующий конструктор

```
Drob (int p1, int p2)
                {if (p2==0)
{ch = 0;  zn = 1;}
else
 {ch=p1;zn=p2;}}
```

3)Конструктор преобразования – конструктор с единственным параметром. Он задает преобразование от типа параметра типу соответствующего класса

```
                Drob ()
                { ch = p1; zn = 1; }
```

4)Копирующий конструктор – конструктор с параметром того же типа. Необходим для того, что бы создавать объекты на базе уже существующих объектов того же типа

```
Drob (Drob&d)
                { ch = d.ch;
                  zn = d.zn; }
```

Копирующий конструктор неявно вызывается при передаче соответствующего объекта по значению в функцию

```
Int main()
{Drob d1(1,2),d2;
Drob d3(d1);
d2=d1;}
```

При присваивании объектов друг другу происходит побитовое копирование информации В классах, имеющих поля-ссылки, такое копирование приводит к искажению данных, поэтому в классах, имеющих поля-ссылки, наличие копирующего конструктора необходимо.

9. Отношения дружественности в объектно-ориентированном программировании. Дружественные функции и классы.

Функции-друзья — это функции, не являющиеся функциями-членами и тем не менее имеющие доступ к защищённым и закрытым членам класса. Они должны быть объявлены в теле класса как «friend»

Губин:

Дружественная функция - это функция объявленная вне области видимости class, но имеющая право доступа к закрытым элементам класса.

Дружественной может быть объявлен как весь класс, так и функция-член класса. Если класс «A» — объявлен в классе «B» как друг, то все собственные **(не унаследованные)** функции-члены класса «A» могут обращаться к любым членам класса «B»

Четыре важных ограничения, накладываемых на отношения дружественности в C++:

1)Если «A» объявляет другом «B», а «B», в свою очередь, объявляет другом «C», то «C» не становится автоматически другом для «A». Для этого «A» должен явно объявить «C» своим другом. (Нет транзитивности)
2)Если класс «A» объявляет другом класс «B», то он не становится автоматически другом для «B». Для этого должно существовать явное объявление дружественности «A» в классе «B». (Несимметричность)
3)Если «A» объявляет класс «B» своим другом, то потомки «B» не становятся автоматически друзьями «A». Для этого каждый из них должен быть объявлен другом «A» в явной форме.
4)Если класс «A» объявляет «B» другом, то «B» не становится автоматически другом для классов-потомков «A». Каждый потомок, если это нужно, должен объявить «B» своим другом самостоятельно.
В общем виде это правило можно сформулировать следующим образом: «Отношение дружественности существует только между теми классами (классом и функцией), для которых оно явно объявлено в коде, и действует только в том направлении, в котором оно объявлено».

+ из Губина:

Объявление дружественности может располагаться в любом месете описания класса.
Дружественность требует явного объявления, то есть не обладает ни свойство симметричности, ни транзитивности.

10. Статические элементы и методы класса.

Чисто Губинский конспект

Статические данные- данные, являющиеся общими для всех объектов данного класса. Память под них выделяется на этапе компиляции до создания первого объекта соответствующего класса. Статические данные объявляются с использованием слова static/

```
class myClass
{
        private:
                static int count;
        public:
                myClass()
                { count ++;}
…
};
```

Статические данные класса инициализируются посредством повторного объявления в глобальной области видимости **(вне класса)**.

```
int myClass:: count=0;
```

Класс модет содержать статические функции. **Статические функции могут оперировать только статическими данными класса.** В статические функции указатель this не передается, так как статические функции не могут обращаться к обычным данным класса. Обращение к статическим элементам класса может происходить либо через объекты соответствующего класса, либо с использованием операции расширения области видимости(более предпочтителен).

Реализация класса для которого можно создавать только 1 объект:

```
class Singl
{
        private:
                Singl ()
                {}
        public:
                static Singl * self;
                static void Init();
…
};
void Singl::Init()
{
        if(self==NULL)
        self=new Singl();
}
Singl * Singl:: self=NULL;
int main ()
{
        Singl::Init();
        Singl:: self.Print();
}
```

11. Константные элементы и методы класса.

Если мы объявим объект вот так: const Drob d;, то:
Компилятор блокирует вызов любых (обращение) ф-ций (к любым объектам) без изменений, потому что он считает , что любая ф-ция может изменить состояние объектов.
С константными объектами могут работать только константные ф-ции классов.
Если создать представитель класса с модификатором const, то компилятор будет проинформирован, что содержимое объекта не должно изменяться после инициализации.
Чтобы предотвратить изменение значений элементов константного объекта, компилятор

генерирует сообщение об ошибке, если объект используется с неконстантной функцией-элементом.
Константная функция-элемент, объявляемая с ключевым словом const после списка параметров, должна удовлетворять следующим правилам:

- она не может изменять значение элементов данных класса;
- не может вызывать неконстантные функции-элементы класса;
- может вызываться как для константных, так и неконстантных объ-ектов класса.

Для того чтобы сделать функцию константной, необходимо указать ключевое слово const после прототипа функции, но до начала тела функции. Если объявление и определение функции разделены, то модификатор const необходимо указать дважды – как при объявлении, так и при ее определении. Те методы, которые только лишь считывают данные из поля класса, имеет смысл делать константными, поскольку у них нет необходимости изменять значения по-лей объектов класса.
Пример объявления констатной ф-ции на примере сущности Drob:

```
int get ch() const
        {return ch;}
```

Константные ф-ции класса не могут менять значение данных объекта.(конспект)
Есть смысл объявлять ф-ции константными , даже не имея константных объектов в том случае , если мы хотим чтобы в них состояние объектов не менялось.
Можно в классе перегружать константные ф-ции константными аналогами.

```
Возможно: void Print() const;    //вызов в зависимости от объекта
                void Print() ;
```

Возможно объявление константных данных класса. Для инициализации константных данных класса необходимо использовать инициализаторы.

Пример сущности вектор постоянной размерности:

```
class Vect
{
        private:
                const int len;
                int * mas;
        public:
                Vect():len(2)
                {
                        mas=new int[len];
                }
Vect():len(p)
                {
                        if(len<=0)
```

```
                abort();
            mas=new int[len];
        }
…
};
```

12. Понятия перегрузки операций. Основные особенности перегрузки операций в C++

Операции перегружаются посредством включения в класс функций со спец. Структурой в имени начинающейся с «operator», а потом обозначения оператора.
Операции могут быть перегружены либо функциями класса либо функциями дружественными.
В функции класса при этом первый операнд операции передается неявно.
В дружественной функции оба операнда передаются явно.
В случае если операция перегружается для пары операндов первый из которых не является объектом того же класса, мы обязаны использовать друж. Функции.
Правила перегрузки операций C++
1. C++ не позволяет создавать новые операции
2. Старшинство операций нельзя изменить
3. С функциями перегружающими операции нельзя использовать параметры по умолчанию
4. Нельзя менять количество операндов в операции (унарные, бинарные)
5. Нельзя перегружать операции для встроенных типов.
6. Не рекомендуется менять интуитивно понятный смысл операции.
7. Операции присваивания или унарный амперсант могут перегружаться.
8. Не могут быть перегружены «.», «.*» , «::», «?:», “#”, “##”
9. Унарные и бинарные операции -,+,*,& требуют раздельной перегрузки
10. Пары операций + и +=, - и -= и т.д. требуют явной и раздельной перегрузки. Из-за того, что перегружен бинарный плюс, не следует неявная перегрузка +=
11. Операции инкремента и декремента (++,--) раздельно перегружаются для префиксной и постфиксной части.
12. Операции индексирования обращ. К функции [], (), =, -> могут быть перегружены только функциями класса (не могут быть перегружены дружественными функциями).
13. Функции перегружающие операции могут быть перегружены
14. При перегрузке операции преобразования типов, тип возвращаемого значения функции не указывается.
15. Пары операций (<=, >=, !=, ==) не обязаны перегружаться совместно

13. Особенности перегрузки операций присваивания и индексирования

```
class MyClass{
private:
	int data;
public:
	MyClass(){data = 0;}
	MyClass(int n){data = n;}
	MyClass& operator =(MyClass& ob){
		data = ob.data;
		return *this;
	 }
};

void main(){
	 MyClass ob1, ob2(3);
	 ob1 = ob2;
}
```

В вышеприведенном коде перегружается “=” (присваивание). Он копирует содержимое поля data одного класса в поле data другого класса. Для этого достаточно написать выражение ob1 = ob2. При использовании перегруженного “=” будет вызываться функция MyClass& operator =(MyClass& ob), определенная в классе. Первый параметр(левый операнд) передается в нее неявным образом по указателю this, второй же, находящийся справа, указан в списке параметров функции. Далее выполняется присвоение полю data левого операнда значения поля data правого операнда. Функция возвращает *this, объект с внесенными изменениями, что позволяет использовать цепочку присваиваний.

```
class MyClass{
private:
int* mas;
public:
	 MyClass(){mas = NULL;}
	 MyClass(int* arr, int sz){
mas = new int[sz];
for(int i = 0; i < sz; i++){mas[i] = arr[i];}
	 }
	 int& operator[](int k){return mas[k];}
};

void main(){
```

```
        int arr[] = {3, 4, 5};
        MyClass ob(arr, 3);
        cout << ob[2];
        ob[1] = 2;
}
```

Здесь перегружается операция индексирования. Имеет смысл ее перегружать, если данными-членами класса являются хранилища данных(массивы, списки). Для использования достаточно написать ob[число]. При этом вызовется функция int& operator[](int k). Имя вызывающего функцию объекта располагается перед квадратными скобками, а указанная в списке параметров переменная – непосредственно внутри квадратных скобок. Функция возвращает значение указанного элемента по ссылке, что позволяет не только использовать его в дальнейшем коде, но также и менять, как показано в строке ob[1] = 2.

14. Особенности перегрузки операций >> и << как операций ввода и вывода

Cout - объект, явл. экземпляром класса ostream, представляющий стандартный выходной поток. Cin - объект, экземпляр класса istream, представляющий входной поток. Для объектов cin и cout и для всех стандартных типов операции << и >> перегружены, как операции ввода и вывода соответственно.

```
class MyClass{
private:
        int data;
public:
        MyClass(){data = 0;}
        MyClass(int n){data = n;}
        friend istream& operator >>(istream& in, MyClass& ob){
                in >> ob.data;
                return in;
        }
        friend ostream& operator >>(ostream& out, MyClass& ob){
                out << ob.data;
                return out;
        }
};

void main(){
  MyClass ob1;
 cin >> ob1;
 cout << ob1;
```

}

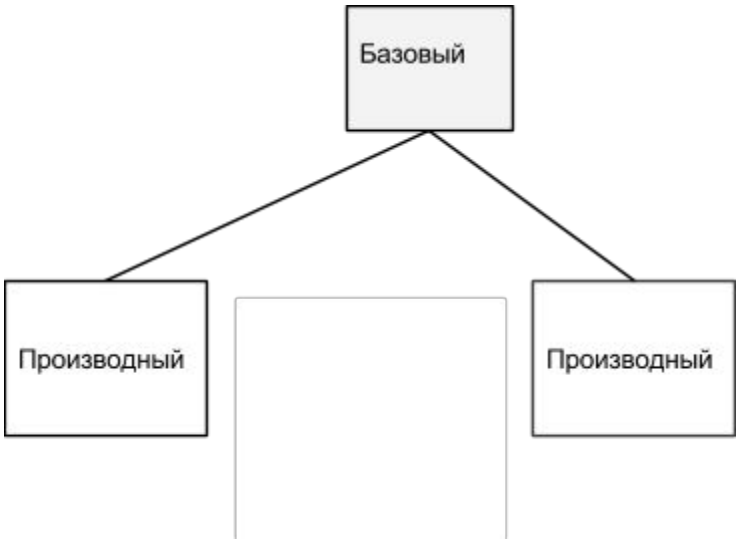
Здесь перегружаются операции консольного ввода и вывода. Важно отметить, что эти функции обязательно должны быть дружественными классу. Для использования достаточно написать cin >> объект для ввода и cout << объект для вывода. При этом вызовется функция friend istream& operator >>(istream& in, MyClass& ob) для ввода или friend ostream& operator >>(ostream& out, MyClass& ob) для вывода. Первый параметр – объект cin/cout, стоящий перед двумя стрелочками, передаваемый по ссылке, второй параметр – объект, находящийся за ними, также передаваемый по ссылке. Функции и принимают, и возвращают объект класса istream/ostream по ссылке, что позволяет использовать цепочку ввода/вывода

15. Отношения наследования между классами. Реализация в C++ открытого одиночного наследования

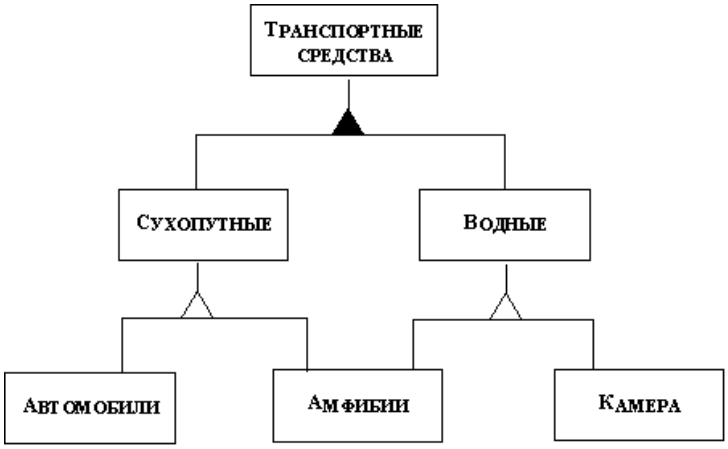
Наследование - способ повторного использования кода, при котором новые классы создаются из уже существующих путем заимствования их данных и функций и добавлением своих собственных.

В C++ возможно **одиночное и множественное** наследование:

-В **одиночном** производный класс наследуется от одного базового:



-В **множественном** от нескольких базовых классов:



В C++ возможно открытое, защищенное и закрытое наследование.

Реализация одиночного наследования

```
class Tbase { //Базовый класс
private:
        int count;
public:
        TBase() {count = 0;}
        void SetCount(int n) {count = n;}
        int GetCount(void) {return count;}
};
class TDerived: public TBase { //Производный класс
public:
        TDerived(): Tbase() {}
        void ChangeCount(int n) {SetCount(GetCount() + n);}};
```

Способ доступа	Спецификатор в базовом классе	Доступ в производном классе
private	private protected public	нет private private
protected	private protected public	нет protected protected

public	private protected public	нет protected public

16. Отношения наследования и защищенные элементы класса. Сравнительная характеристика разных уровней доступа к элементам и методам класса

Наследование - способ повторного использования кода, при котором новые классы создаются из уже существующих путем заимствования их данных и функций и добавлением своих собственных.
В C++ возможно одиночное и множественное наследование:
-В одиночном производный класс наследуется от одного базового
-В множественном от нескольких базовых классов
В C++ возможно открытое, защищенное и закрытое наследование.
Пусть D унаследован от B. Виды: открытое, закрытое, защищённое, виртуальное. Во всех случаях класс D получает доступ к открытым и защищённым членам B. Отличия:
1. Открытое(public) - Открытые члены B становятся открытыми в D. Защищённые B - защищёнными D. Говорят, что класс наследует как интерфейс, так и реализацию предка.
2. Закрытое(private) - Открытые и защищённые члены B становятся закрытыми в D. Применяют когда класс должен наследовать только реализацию.
3. Защищённое(protected) - открытые и защищённые члены B становятся защищёнными в D.
-При открытом наследовании public главного класса становится добавочным к public дочернего класса.
-Защищенные данные и функции базового класса становятся защищенными данными производного класса.
-Закрытые данные базового класса производому недоступны, но доступ к ним возможен через открытые\защищенные функции базового класса.
-Защищенные элементы базового класса ведут себя как открытые при обращениях из производн. классов и ведут себя как закрытые при внешних обращениях.

17. Базовые и производные классы. Использование и назначение методов базового класса в производных классах

Базовый класс — это класс, на основе которого создаются другие классы.

Производный класс — это класс, который наследует все свойства базового класса, и можно добавлять или переопределять методы и свойства в зависимости от необходимости.
Наследование - способ повторного использования программного кода, при котором производные классы создаются путём заимствования их данных и ф-ий базового класса и добавлением своих. Переопределение функции — одна из возможностей языка программирования, позволяющая подклассу обеспечивать специфическую реализацию метода, уже реализованного в одном из родительских классов. Реализация метода в подклассе переопределяет (заменяет) его реализацию в базовом классе, описывая метод с теми же названием, сигнатурой формальных параметров и типом возвращаемого результата, что и у метода базового класса. Если вызов метода происходит от объекта родительского класса, то выполняется версия метода родительского класса, если же объект подкласса вызывает метод, то выполняется версия дочернего класса.

```
class Rectangle // базовый класс
{public:
        Rectangle() {l=0, w=0;}
        Rectangle (int l1, int w1) {l=l1, w=w1;}
        void print()  // функция print() базового класса{
                cout << "Length = " << l << "; Width = " << w;}
private:
        int l;
        int w;
        };

class Box : public Rectangle // производный класс
{public:
        Box(): Rectangle(), h(0) {}
        Box (int l1, int w1, int h1): Rectangle(l1, w1), h(h1) {}
        void print()  // ф-я print() произв-го класса, переопределяющая родительскую ф-ю
print() {
          Rectangle::print();  // вызов родительской функции print()
        cout << "; Height= " << h;}
private:
int h;};

int main(){
        Rectangle myR (5, 3);
        myR.print(); // outputs: Length = 5; Width = 3
        Box myBox(6, 5, 4);
        myBox.print(); // outputs: Length = 6; Width = 5; Height = 4
}
```

18. Понятие и суть полиморфизма. Статическое (раннее) и динамическое (позднее) связывание

Полиморфизм — это переопределение в производном классе функций базового класса и динамическое соответствие между операцией и функцией её выполняющие.
Суть полиморфизма.
Концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Выбор же конкретного действия, в зависимости от ситуации, возлагается на компилятор.
Понятие динамического связывания. Если функция в базовом классе объявлена как virtual и если затем эта функция вызывается через указатель базового класса, указывающий на объект производного класса, то компилятор будет динамически связывать, то есть во время выполнения выбирать соответствующие функция производного класса. Этот процесс называется динамическим связыванием.
Статическое (раннее) связывание означает, что объект и вызов функции связываются между собой на этапе компиляции. Это означает, что вся необходимая информация для того, чтобы определить, какая именно функция будет вызвана, известна на эпате компиляции программы.

19. Поддержка полиморфизма в C++. Виртуальные функции.

Полиморфизм - переопределение в производном классеф-ий базового класса и динамическое соответствие между операцией и ф-ей еевыполняющей.
Виртуальная функция - это функция, объявленная с ключевым словом «virtual» в базовомклассе и переопределенная в одном или в несколькихпроизводных классах.
Пример:virtual void vFunc(){};
Правила использования виртуальных функций (поддержка полиморфизма):
1.Ф-я объявляется виртуальной посредством ключевого слова «virtual», предшествующего ее прототипу в базовом классе.
2.Если ф-я объявлена виртуальной, то она остается такой и после ее переопределения на более низком уровне иерархии.
3.Если в производном классе решено не описывать виртуальную ф-ю, то производный класс наследует описание виртуальной ф-ии из базового класса.
4.Переопределенная вирт. ф-я должна иметь тот же тип возвращаемого значения и ту же сигнатуру формальных параметров, что и виртуальная ф-ябазового класса.

20. Чисто виртуальные функции. Абстрактные базовые классы.

Чисто виртуальная функция - та, которая объявляетсяв базовом классе, но не имеет в нем определения. Поскольку она не имеет определения, то всякий производный класс обязан иметь свою собственную версию определения. Пример:
virtual возвращаемое значение имя_функции(список параметров) = 0;
Абстрактный класс – базовый класс, не предполагающий создание экземпляров.В C++ абстрактный класс объявляется включением хотя бы одной чисто виртуальной ф-ии.
Пример:

```
class Ab{
int a;
virtual void Abstr()=0;
};
```

Использование абстрактных классов:
1.В качестве базового в иерархии наследования.
2.Объявление указателей на абстрактные классы.
3.В качестве типа возвращаемого значения и типа формальных параметров ф-ий.

21. Особенности и порядок создания программ с визуальным интерфейсом в среде Visual Studio 2012/2013.

Объясним процесс работы с визуализацией приложений на примере окна с полем, содержащим текст. При выборе типа создаваемого проекта выбираем Windows Forms Application.
Прежде чем приступить непосредственно к программированию, следует понять, что такое событие. Событие это действие, которое происходит при определённых условиях.
Самыми простыми (и наиболее распространенными) можно считать:
Activated — событие, возникающее при активизации элемента.
Click — возникает при одиночном нажатии на элемент.
DoubleClick — двойной щелчок по элементу.
HelpRequested — срабатывает при нажатии клавиши <F1>.
Paint — возникает, когда элемент перерисовывается.
MouseLeave — событие срабатывает, когда курсор покидает границы элемента.
Не стоит забывать, что свойственные элементу события зависят от его типа. Чтобы просмотреть все доступные для объекта формы события следует выделить его и в окне свойств выбрать значок молнии.
При открытии файла форм (Form.h) можно будет встретить подобный код:

```
private: System::Void Form_Load(System::Object^ sender, System::EventArgs^ e){
}
```

Это та самая функция Form_Load, срабатывающая при загрузке формы. Далее добавим компонент TextBox.
Для этого откроем ресурс формы и выделяем его. Далее выбираем панель с инструментами Toolbox и перетаскиваем компонент TextBox на форму. Модифицирем функцию Form_Load следующим образом:

```
private: System::Void Form_Load(System::Object^ sender, System::EventArgs^ e) {
   textBox1->Text = "Поставьте пятерку по ООП!"; //textBox1 – имя добавленного текст бокса
}
```

Запустив проект, в итоге должно появиться окно с текстовым полем, содержащим "Поставьте пятерку по ООП!"

Рекомендации:

1)необходимо всегда поддерживать актуальным состояние объектов.
2)графика должна быть следствием внутреннего состояния объектов.
3) если «игровое поле» существует в единственном экземпляре, то ему может соответствовать класс только со статическими данными
4) необходимо разделять сущности «игровое поле» и «части игрового поля», «игровое поле» и «действующие персонажи». Сущн. «игр. поле» должна владеть и частями и персонажами
5)не существует четких критериев разделения функциональности между сущностями.
6)обработчики событий формы должны быть максимально облегчены
7)необходимо стараться минимизировать зависимость реализации ф-ё класса от деталей виз. интерфейса.

22. Шаблоны классов. Правила создания шаблонов классов.

Суть:

Шаблоны классов используются для того, чтобы избежать бессмысленного и беспощадного дублирования кода. Например, мы хотим создать класс Stack (стэк), и нам хочется, чтобы он работал и для int-ов, и для стрингов, и для float и т.п. Можно было бы написать по классу для каждого, но в C++ есть решение получше - шаблоны. Мы пишем один шаблон для некого типа данных T, и когда нам нужен специлизированный стэк для интов, например, компилятор по шаблону нам его запиливает, подставляя вместо T нужный нам тип. По сути шаблон пишется как самый обычный класс, но перед словами class ClassName имеет строчку: template<typename T>, и в самом классе мы вместо типа данных, который должен меняться пишем T. Методы класса по факту являются шаблонами функций. ЗЫ. Вместо “T” может быть любая буква.

Обрезанный пример из инета, который отражает:

```
template <typename T>
class Stack
```

```
{
private:
    T *stackPtr; // указатель на стек
    int size; // размер стека
    T top; // вершина стека
public:
    Stack(int = 10);// по умолчанию размер стека равен 10 элементам
    ~Stack(); // деструктор
    void printStack();
};

int main()
{
    Stack <int> myStack(5);
    myStack.printStack(); // вывод стека на экран
    return 0;
}
// конструктор
template <typename T>
Stack<T>::Stack(int s)
{
    size = s > 0 ? s: 10;   // инициализировать размер стека
    stackPtr = new T[size]; // выделить память под стек
    top = -1; // значение -1 говорит о том, что стек пуст
}

// деструктор
template <typename T>
Stack<T>::~Stack()
{
    delete [] stackPtr; // удаляем стек
}
```

Правила объявления шаблонов (конспект):

1. Параметры представляют собой типы и константы, перечисленные через запятую. Могут иметь значения по умолчанию. Пример:

```
template<class T1, class T2> struct Pair {
        T1 first;
        T2 second;
}
```

2. Типы могут быть как стандартными, так и определенными пользователем. В последнем случае необходимо будет перегрузить некоторые операции.
3. Имя параметра-типа может быть любым, но принято начинать его с префикса T. Внутри класса-шаблона параметр может появляться в тех местах, где разрашено указывать конкретный тип.

4. Параметр-тип можно рассматривать как параметр, на место которого при компиляции будет подставлен конкретный тип данных.
5. Шаблоны могут быть приозводными как от шаблонов, так и от обычных классов, а также являться базовыми и для шаблонов, и для обычных классов.
6. Методы шаблоноа класса автоматичести становятся шаблонами функции. При реализации функции вне класса описание параметра шаблона в заголовке функции должно соответствовать шаблону класса. См. пример выше, где конструктор запилен отдельно.

Сущности, используемые в практических заданиях.

1. Фигуры в пространстве (есть).
Норм версия (на основе губинской): http://pastebin.com/0rA0uk8Q
Версия ниже (от Владоса) не очень правильная, т.к. нет смысла в базовом классе еще и радиус добавлять. На хиба?

```
using namespace std;
class Figura
{
protected:
double x;
double y;
double z;
double R;
public:
Figura()
{
SetXY(0, 0, 0,0);
}
Figura(double p1, double p2, double p3,double p4)
{
SetXY(p1, p2, p3,p4);
}
Figura(Figura &s)
{
SetXY(s.x, s.y,s.z, s.R);
}
void SetXY(double p1, double p2, double p3,double p4)
{
x = p1;
```

```
y = p2;
z = p3;
R = p4;
}
double getX()
{
return x;
}
double getY()
{
return y;
}
double getZ()
{
return z;
}
double getR()
{
return R;
}
virtual bool Print() = 0;
virtual double Ob()=0;

};

class Kyb : public Figura
{
public:
double a;
Kyb() :Figura()
{
SetA(0);
}
Kyb(double p1, double p2, double p3, double p4, double p5) :Figura(p1, p2, p3,p4)
{
SetA(p5);
}
Kyb(Kyb& r1) :Figura(r1.getX(), r1.getY(),r1.getZ(), r1.getR())
{
SetA(r1.a);
}
void SetA(double p1)
```

```
{
if (p1<0)
a = 0;
else
a = p1;
}
double Ob()
{double V;
                V=a*a*a;
return V;
}
bool Print()
{
                cout <<Ob()<<endl;
        if ((sqrt(2)*a>R) )
return false;
else
return true;
}
};
class Sfera : public Figura
{

public:
double r;
Sfera() :Figura()
{
SetR(0);
}
Sfera(double p1, double p2, double p3, double p4,double p5) :Figura(p1, p2, p3,p4)
{
SetR(p5);
}
Sfera(Sfera& r1) : Figura(r1.getX(), r1.getY(),r1.getZ(), r1.getR())
{
SetR(r1.R);
}
void SetR(double p1)
{
if (p1<0)
r = 0;
else
```

```
r = p1;

}
double Ob()
{
	double V=0.75*3.14*r*r*r;
return V;
}
bool Print()
{
	cout <<Ob()<<endl;
if (r>R)
return false;
else
return true;
}

};
class Piramida : public Figura
{

public:
double a;
double b;
Piramida() :Figura()
{
SetABC(0, 0);
}
Piramida(double p1, double p2, double p3, double p4, double p5, double p6) :Figura(p1, p2,
p3,p4)
{
SetABC( p5, p6);
}
Piramida(Piramida & r1) : Figura(r1.getX(), r1.getY(), r1.getZ(), r1.getR())
{
SetABC(r1.a, r1.b);
}
void SetABC(double p1, double p2)
{
if (p1<0)
a = 0;
else
```

```
a = p1;
if (p2<0)
b = 0;
else
b = p2;
}
double Ob()
{
	double V;
		V=0.33*a*a*b;
return V;
}
bool Print()
{
		cout <<Ob()<<endl;
		if (R<b && R<sqrt(2)*a)
		return false;
else
return true;
}

};
struct Fixit
{
	Figura* item;
		Fixit* next;
};

class Cont
{
private:
	Fixit* first;
	public:
	Cont()
	{
		first = NULL;
	}
	Cont(Figura *p)
	{
		first = NULL;
		Add(p);
	}
```

```
	void Add(Figura *p)
	{
		Fixit *temp = new Fixit;
		temp->item = p;
		temp->next = first;
		first = temp;
	}

	bool Ekran(){
Fixit *temp = first;
while (temp != NULL)
{
if (temp->item->Print())
cout << "yes\n"; //проверка фигуры и вывод обьема
else
cout << "no\n";
temp=temp->next;

}
return true;
	}};


int main()
{
Kyb g(3.0, 5.0, 1.0, 11.0,10.0);
Sfera o;
Piramida k;
Cont p;
p.Add(&g);
p.Add(&o);
p.Add(&k);
p.Ekran();
return 0;
};
```

2. Фигуры на плоскости (есть)
Либо Губинская версия: http://pastebin.com/6XBAV262

```
#include "stdafx.h"
#include <iostream>
```

```
#include "math.h"

using namespace std;
class Figurka
{
protected:
	double x;
	double y;
public:
	Figurka()
	{
		SetXY(0, 0);
	}
	Figurka(double p1, double p2)
	{
		SetXY(p1, p2);
	}
	Figurka(Figurka &s)
	{
		SetXY(s.x, s.y);
	}
	void SetXY(double p1, double p2)
	{
		x = p1;
		y = p2;
	}
	double getX()
	{
		return x;
	}
	double getY()
	{
		return y;
	}
	virtual void Print() = 0;
	virtual double Sfig() = 0;


};
class Triangle : public Figurka
{
private:double a;
```

```
        double b;
        double c;
public:

        Triangle() :Figurka()
        {
                SetABC(0, 0, 0);
        }
        Triangle(double p1, double p2, double p4, double p5, double p6) :Figurka(p1, p2)
        {
                SetABC(p4, p5, p6);
        }
        Triangle(Triangle& r1) :Figurka(r1.getX(), r1.getY())
        {
                SetABC(r1.a, r1.b, r1.c);
        }
        void SetABC(double p1, double p2, double p3)
        {
                if (p1<0)
                        a = 0;
                else
                        a = p1;
                if (p2<0)
                        b = 0;
                else
                        b = p2;
                if (p3<0)
                        c = 0;
                else
                        c = p3;
        }
        double Sfig()
        {
                double S, p;
                p = (a + b + c) / 2;
                S = sqrt(p*(p - a)*(p - b)*(p - c));
                return S;
        }
        void Print()
        {
                cout << Sfig() << endl;
```

```
	}
};
class Circle : public Figurka
{
private:
	double r;
public:
	Circle() :Figurka(){
		SetR(0);
	}
	Circle (double x, double y, double r) :Figurka(x, y) {
		SetR(r);
	}
	Circle(Circle& c) :Figurka(c.getX(), c.getY()){
		SetR(c.r);
	}
	void SetR(double p1) {
		if (p1 < 0) r = 0;
		else r = p1;
	}
	double getR() {
		return r;
	}
	void Print() {
		cout << Sfig() << endl;
		cout << "[" << getX() << "," << getY() << "," << r << "]" << endl;
	}
	double Sfig()
	{
		double S = 3.14*(r^2);
		return S;
	}
};
class Kvad : public Figurka
{
private:double a;
public:

	Kvad() :Figurka()
	{
		SetA(0);
	}
```

```
	Kvad(double p1, double p2, double p4) :Figurka(p1, p2)
	{
		SetA(p4);
	}
	Kvad(Kvad & r1) : Figurka(r1.getX(), r1.getY())
	{
		SetA(r1.a);
	}
	void SetA(double p1)
	{
		if (p1<0)
			a = 0;
		else
			a = p1;
	}
	double Sfig()
	{
		double S;
		S = a*a;
		return S;
	}
	void Print()
	{
		cout << Sfig() << endl;
	}
};

struct Och
{

	Figurka *data;
	Och *next;
};
class kont
{
private:
	Och *first;
	Och *last;
public:
	kont()
	{
		first = NULL;
```

```
        last = NULL;
}
void add(Figurka* qq)
{
        if (first == NULL){
                Och *temp = new Och;
                temp->data = qq;
                first = temp;
                last = temp;
                first->next = NULL;
                last->next = NULL;
        }
        else
        if (first->next == NULL)
        {
                Och *temp = new Och;
                temp->data = qq;
                first->next = temp;
                last = temp;
                last->next = NULL;
        }
        else
        {
                Och *temp = new Och;
                temp->data = qq;
                last->next = temp;
                last = temp;
                last->next = NULL;
        }
}


virtual bool Print1(){
        Och *temp = first;
        while (temp != NULL)
        {
                if (temp->data->Print())
                        cout << "yes\n";
                else
                        cout << "no\n";
                temp = temp->next;
```

```
            }
            return true;
        }
};
int main()
{
        Triangle z(1, 2, 3, 4, 6);
        Circle c( 2, 7, 6);
        Kvad d(2, 7, 8);
        kont q;
        q.add(&z);
        q.add(&c);
        q.add(&d);
        q.Print1();
        return 0;
};
```

3. Элементы графа (есть)
Вот еще моя версия графа, и я иду спать: http://pastebin.com/hFUai29K

```
// Severina3.cpp: определяет точку входа для консольного приложения.
//

#include "stdafx.h"
#include "iostream"
using namespace std;


class Base{
protected:
        int x;
public:

        Base ()
        {
                SetX(1);
        }

        Base (int a)
        {
                SetX(a);
```

```
	}

	Base (Base& b)
	{
		SetX(b.x);
	}

	virtual void Print ()
	{
		cout << "[" << x << "]" << endl;
	}

	virtual int GetY ()
	{
		return 0;
	}

	virtual void PrintName(){}

	virtual char GetName ()
	{
		return 0;
	}

	void SetX (int x1)
	{ if (x1<1) abort();
	else x=x1;
	}

	int GetX ()
	{
		return x;
	}

	virtual int Type () = 0;
};
	class Vert: public Base
	{
	public:
		Vert (): Base()
		{}
```

```
        Vert (int x1): Base (x1)
        {}

        Vert (Vert& x1): Base(x1.x)
        {}

        int GetX ()
        {
        return x;
        }

        virtual int Type ()
        {
                return 0;
        }
};

class Edge: public Base
{
private:
        int y;
        char name;
public:
        Edge (): Base()
        {
                SetY(2);
                SetName ('a');
        }

        Edge (int x1, int y1, char name1): Base (x1)
        {
                SetY(y1);
                SetName(name1);
        }

        Edge (Edge& e1): Base(e1.x)
        {
                SetY(e1.y);
                SetName(e1.name);
        }

        virtual int Type ()
```

```
	{
		return 1;
	}

	void SetName (char a1)
	{
		name=a1;
	}

	void SetY(int y1)
	{
		if (y1<1) abort();
		else y=y1;
	}

	int GetY()
	{
		return y;
	}

	char GetName()
	{
		return name;
	}

	void PrintName()
	{
		cout << name;
	}

	virtual void Print ()
	{
		PrintName();
		cout << "("<< x <<";" << y<< ")" << endl;
	}
};
struct Element
{
		Base *elem;
		Element *next;
	};
```

```
class Graph{
private:
	Element *first;
	int **A;
	int p;
	int ver;
	int reb;
	int **B;
public:
	Graph()
	{
	first=NULL;
	}

	void Push (Base *el)
	{
		Element *temp = new Element;
		temp->elem = el;
		temp->next = first;
		first = temp;
	}

	void Print()
	{
		Element *temp = first;
		while (temp!=NULL)
		{
			temp->elem->Print();
			temp=temp->next;
		}
	}

	void SetMatrix()
	{
		Element *temp = first;
		p=0;
		while (temp!=NULL)
		{
			if (temp->elem->Type() ==1){
			if (p<temp->elem->GetX())
			{
				p=temp->elem->GetX();
```

```
                }
                if (p<temp->elem->GetY()){
                        p=temp->elem->GetY();
                }}
                else {
                        if (p<temp->elem->GetX())
                {
                        p=temp->elem->GetX();
                        }}
                temp=temp->next;
        }
        int **A= new int *[p];
        for (int i = 0;i<p; i++){
                A[i]= new int[p];};
        for(int i=0;i<p;i++){
        for (int j=0; j<p; j++){
                A[i][j]=0;}};
        Element *temp1 = first;
        while (temp1!=NULL)
        {
                if (temp1->elem->Type() ==1){
        A[temp1->elem->GetX()-1][temp1->elem->GetY()-1]=1;
        A[temp1->elem->GetY()-1][temp1->elem->GetX()-1]=1;
                }
        temp1=temp1->next;
        }
        cout << "Матрица смежности" << endl;
        for(int i=0;i<p;i++){
        for (int j=0; j<p; j++){
                cout << A[i][j]<< ' ';}
        cout << endl;};
}

void InMatrix ()
{
        cout << "Матрица инцидентности" << endl;
        Element *temp = first;
        ver=0;
        reb=0;
        cout << "  ";
        while (temp!=NULL)
        {
```

```
        if (temp->elem->Type() == 1)
        { reb++;
        temp->elem->PrintName();
        cout << " ";
        if (ver<temp->elem->GetX())
        {
                ver=temp->elem->GetX();
        }
        if (ver<temp->elem->GetY()){
                ver=temp->elem->GetY();
        }
        }
        else {
                if (ver<temp->elem->GetX())
                        ver = temp->elem->GetX();}
        temp=temp->next;
}
cout << endl;
int **B= new int *[ver];
for (int i = 0;i<ver; i++){
        B[i]= new int[reb];};
for(int i=0;i<ver;i++){
for (int j=0; j<reb; j++){
        B[i][j]=0;}};
Element *temp1 = first;
int r=0;
while (temp1!=NULL)
{
        if (temp1->elem->Type() ==1)
        {
                B[temp1->elem->GetX ()-1][r]=1;
                B[temp1->elem->GetY ()-1][r]=1;
                r++;
        }
        temp1=temp1->next;
}
int n=1;
for(int i=0;i<ver;i++){
        cout << n << " ";
        n++;
        for (int j=0; j<reb; j++){
                cout << B[i][j]<< ' ';}
```

```
                        cout << endl;};
        }

        ~ Graph ()
        {
                Element *temp = first;
                while (temp!=NULL)
                {first=first->next;
                delete temp;
                temp=first;}
        }
};

int _tmain(int argc, _TCHAR* argv[])
{       setlocale(LC_ALL,"Russian");
        Graph g;
        Vert s(5);
        Edge d;
        Edge e1(2,3,'b');
        Edge e2(4,6,'c');
        g.Push(&d);
        g.Push (&e2);
        g.Push (&s);
        g.Push (&e1);
        g.Print();
        g.SetMatrix();
        g.InMatrix();
        system ("pause");
        return 0;
}
```

4. Вектор в многомерном пространстве (есть)

```
// lb22.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "iostream"
using namespace std;

class Vector
{
```

```
private:
	const int size;
	double *mas;
public:
	Vector();
	Vector(int);
	Vector(double *,int);
	Vector(Vector&);
	friend istream& operator >> (istream&, Vector&);
	friend ostream& operator << (ostream&, Vector);
	double& operator [] (int);
	double& operator () (int);
	double* getMas() const;
	int getSize() const;
	Vector operator + (Vector) const;
	Vector operator - (Vector) const;
	Vector operator * (double) const;
	Vector operator = (Vector);
	double operator * (Vector) const;
	bool operator == (Vector) const;
	bool operator != (Vector) const;

};
int _tmain(int argc, _TCHAR* argv[])
{
	double M1[2], n=5;
	M1[0]=1;
	M1[1]=2;
	Vector v1,v2 ,v3 (M1,2), v4 (v3), v5(2) ;
	cin>> v1 >> v2;
	cout <<"\nV1-" << v1 <<"\nV2-" << v2 << "\nV3-" << v3 << "\nV4-" << v4 << "\nV5-" << v5 ;
	v3=v1+v2;
	cout <<"\n Summa v3=v1+v2= " <<  v3;
	v5=v3-v4;
	cout <<"\n Raznost v5=v3-v4= " << v5;
	v5=v4*n;
	cout <<"\n Proizved v5=v4*n= " << v5;
	n=v1*v2;
	cout <<"\n Proizved v1*v2= " << n;
	if (v1==v2)
		cout <<"\n  v1=v2 " ;
	if (v3!=v5)
```

```
                cout <<"\n  v3!=v5 " ;
        cout <<"\nv4[0]=" << v4[0];
        system ("pause");
        return 0;
}

Vector :: Vector() : size(2)
{
        mas= new double [size];
        mas[0]=0;
        mas[1]=0;
}
Vector :: Vector(int p) : size(p)
{
        if(p<=0)
                abort();
        mas= new double [size];
        for (int i=0; i<size; i++)
                mas[i]=0;
}
Vector :: Vector(double* M, int p) : size(p)
{
        if(p<=0)
                abort();
        mas= new double [size];
        for (int i=0; i<size; i++)
                mas[i]=M[i];
}
Vector :: Vector(Vector& v) : size(v.size)
{
        mas= new double [size];
        for (int i=0; i<size; i++)
                mas[i]=v.mas[i];
}
istream& operator >> (istream& in, Vector& v)
{
        cout <<"Vvedite " << v.size << " elementa\n";
        for (int i=0; i<v.size; i++)
                cin >> v.mas[i];
        return in;
}
ostream& operator << (ostream& out, Vector v)
```

```
{
	cout<<" ( ";
	for (int i=0; i<v.size-1; i++)
		cout << v.mas[i] << " ; ";
	cout << v.mas[v.size-1] << " ) ";
	return out;
}
double& Vector :: operator [] (int p)
{
	if (p<0||p>=size)
		abort();
	return mas[p];
}
double& Vector :: operator () (int p)
{
	if (p<0||p>=size)
		abort();
	return mas[p];
}
double* Vector :: getMas() const
{
   return mas;
}
 int Vector  :: getSize() const
{
	return size;
}

Vector  Vector ::  operator + (Vector v2) const
{
	if (size!= v2.size)
		abort();
	Vector v1((Vector&) *this);
	for (int i=0; i< v1.size; i++)
		v1.mas[i]+=v2.mas[i];
	return v1;
}
Vector  Vector ::  operator - (Vector v2) const
{
	if (size!= v2.size)
		abort();
	Vector v1((Vector&) *this);
```

```
	for (int i=0; i< v1.size; i++)
		v1.mas[i]-=v2.mas[i];
	return v1;
}
Vector  Vector ::  operator * (double p) const
{
	Vector v1((Vector&) *this);
	for (int i=0; i< v1.size; i++)
		v1.mas[i]*=p;
	return v1;
}
double  Vector ::  operator * (Vector v) const
{
	if (size!= v.size)
		abort();
	double sum=0;
	for (int i=0; i< size; i++)
		sum+=mas[i]*v.mas[i];
	return sum;
}
Vector Vector :: operator = (Vector v)
{
	if (size!= v.size)
		abort();
	if (this ==&v)
		return *this;
	for (int i=0; i< size; i++)
		mas[i]=v.mas[i];
	return *this;
}
bool Vector :: operator == (Vector v)const
{
	if (size!= v.size)
		abort();
	for (int i=0; i< size; i++)
		if (mas[i]!=v.mas[i])
			return false;
	return true;
}
bool Vector :: operator != (Vector v)const
{
	return (!((*this)==v));
```

```
}
```

5. Вектор в двумерном пространстве (есть)

```
class Vector
{
        private:
double x;
double y;
public:
Vector();
Vector(double,double);
Vector(double);
Vector(Vector&);
                double getX ();
                double getY ();
                void setx (double);
                void sety (double);
                void Read ();
void Print() const;
Vector Add(Vector);
Vector Vich(Vector);
Vector Proizved(double);
double Modul();
double ScalPro(Vector);
bool Sravn(Vector) ;

};


void main ()
{
        Vector v1(1,2), v2(2), v3;
        Vector v;
        v1.Print();
        v2.Print();
        v3.Print();
        v3 = v1.Add(v2);
        v3.Print();
        v.Read();
        v.Print();
```

```
        v3=v.Vich(v1);
        v3.Print();
        v3=v2.Proizved(2);
        v3.Print();
        cout << " \nModul v= " << v.Modul() << " \nSkalarnoe Proisvedenie v*v1= " <<
v.ScalPro(v1);

if (v. Sravn(v2)) cout << "\nv=v2\n"; else cout << "\nv!=v2\n";
        system("pause");
}



Vector::Vector()
{x=0; y=0;}
Vector::Vector(double p1,double p2)
{x=p1; y=p2; }
Vector::Vector(double p1)
{ x=p1; y=0;}
Vector::Vector (Vector &v)
{ x=v.x; y=v.y;}
double Vector:: getX ()
{return x;}
double Vector:: getY ()
{return y;}
void Vector:: setx (double p)
{x=p;}
void Vector:: sety (double p)
{y=p;}
void Vector::Read ()
{ cout << " Vvedite koordinaty: \n" ;
cin >> x >> y;}
void Vector::Print() const
{ cout<< "(" << x << ";" << y << ")" << endl;
}
Vector Vector :: Add( Vector v2)
{ Vector v;
v.x=x+v2.x;
v.y=y+v2.y;
return v;
}
Vector Vector :: Vich ( Vector v2)
```

```
{ Vector v;
v.x=x-v2.x;
v.y=y-v2.y;
return v;
}
Vector Vector :: Proizved (double p)
{ Vector v;
v.x=x*p;
v.y=y*p;
return v;
}
double Vector ::Modul()
{ double p;
p=sqrt(pow(x,2)+pow(y,2));
return p;
}
double Vector :: ScalPro (Vector v2)
{
        double p;
p= (x*v2.x) + (y*v2.y);
return p;
}
bool Vector :: Sravn (Vector v2)
{ if ( x==v2.x && y==v2.y)
return true;
else
return false;
}
```

6. Полином (есть)

```
#include "stdafx.h"
#include "iostream"
#include "math.h"
using namespace std;
class Poli {
private:
        const int size;
        double* mas;
public:
    Poli ():size(2)// Конструктор по умолчанию
    {
        mas = new double [2];
```

```
        mas[0]=1;
        mas[1]=1;
}
Poli (int n):size (n)// Преобразующий конструктор
{       if (n<=0) abort();
        mas = new double [size];
        for (int i=0; i<size; i++){
        mas[i]=i+1;}
}
Poli (double* m, int n): size (n)// Инициализирующий конструктор
{       if (n<=0) abort();
        mas = new double [size];
        for (int i=0; i<size; i++){
                mas[i]=m[i];}
                }

Poli (Poli& p): size (p.size) // Копирующий конструктор
{mas = new double [size];
for (int i=0; i<size; i++){
        mas[i] = p.mas[i];}
}

bool operator == (Poli p) const// Проверка на равенство
{if (size!=p.size) return false;
for (int i=0; i<size; i++){
        if (mas[i]!=p.mas[i]) return false;}
return true;
}

bool operator != (Poli p)// Проверка на неравенство
{ if (size!=p.size) return true;
for (int i=0; i<size+1; i++){
        int d = (*this)(i); //this->operator ()(i);
        int r = p(i);
        if (d!=r) return true;};
 return false;}


Poli operator = (Poli p) //Присваивание
{
        if (size!=p.size) abort ();
        if (this == &p) return *this;
```

```
        for (int i=0; i<size; i++){
                mas[i]=p.mas[i];}
        return *this;
}

double& Getmas() const// Функция доступа mas
{return *mas;};

int Getsize() const // Функция доступа size
{ return size;};

void Setmas (double *A) //Функция инициализации
{for (int i=0; i<size; i++){
        mas[i]=A[i];}};

Poli operator + (Poli p)// Сумма полиномов
{  if (size>=p.size){
        Poli temp(size);
        for (int i=0; i<p.size; i++){
                temp.mas[i]=mas[i]+p.mas[i];}
        for (int i=p.size; i<size;i++){
                temp.mas[i]=mas[i];}
return temp;}
if (p.size>size){
        Poli temp(p.size);
        for (int i=0; i<size; i++){
                temp.mas[i]=mas[i]+p.mas[i];}
        for (int i=size; i<p.size;i++){
                temp.mas[i]=p.mas[i];}
return temp;}}

Poli operator - (Poli p)// Разность полиномов
{ if (size>=p.size){
        Poli temp(size);
        for (int i=0; i<p.size; i++){
                temp.mas[i]=mas[i]-p.mas[i];}
        for (int i=p.size; i<size;i++){
                temp.mas[i]=mas[i];}
return temp;}
if (p.size>size){
        Poli temp(p.size);
        for (int i=0; i<size; i++){
```

```
                temp.mas[i]=mas[i]-p.mas[i];}
        for (int i=size; i<p.size;i++){
                temp.mas[i]=(-1)*p.mas[i];}
return temp;}}

double& operator () (double k) const// Значение полинома
{ double temp = 0;
        for (int i=0; i<size; i++){
                temp+=mas[i]*pow(k,i);}
        return temp;}

double& operator [] (int k) const //Доступ в коэффициенту полинома
{ return mas[k];}

Poli operator * (double k)//Умножение на число
{ Poli temp (size);
for (int i=0; i<size; i++){
        temp.mas[i]=mas[i]*k;}
return temp;}

Poli operator + (double k)//Сложение с числом
{ Poli temp(*this);
temp.mas[0]+=k;
return temp;};

bool operator > (Poli p) const// Больше
{int n, k;
        for(int i=size-1; i>=0; i--){
                if (mas[i]!=0)
                {n=i;
                break;}
                else continue;}
        for(int i=p.size-1; i>=0; i--){
                if (p.mas[i]!=0)
                {k=i;
                break;}
                else continue;}
        if (n>k) return true;
        else return false;};

bool operator < (Poli p) const//Меньше
{
```

```
		int n, k;
		for(int i=size-1; i>=0; i--){
			if (mas[i]!=0)
			{n=i;
			break;}
			else continue;}
		for(int i=p.size-1; i>=0; i--){
			if (p.mas[i]!=0)
			{k=i;
			break;}
			else continue;}
		if (k>n) return true;
		else return false;};


friend ostream& operator << (ostream& out, Poli& p)// Вывод на консоль
{   int s = p.size;
if (p.mas[0]!=0){
	cout << p.mas[0];};
	if(p.mas[1]>0 && p.mas[0]!=0){
			cout << "+" << p.mas[1]<< "*x";}
	else if(p.mas[1]>0 && p.mas[0]==0){
			cout << p.mas[1]<< "*x";}
		else if (p.mas[1]< 0){
			cout << p.mas[1] << "*x";}
	for (int i=2; i<s; i++){
		if(p.mas[i]>0){
			cout << "+" << p.mas[i]<< "*x^" <<i;}
		else if (p.mas[i]< 0){
			cout << p.mas[i] << "*x^" << i;}
		else continue;}
	cout << endl;
	return cout;}

friend istream& operator >> (istream& cin, Poli& p) // Ввод с консоля
{   int s = p.size;
	for (int i=0; i<s; i++){
		cin >> p.mas[i];}
	return cin;}
};

int _tmain(int argc, _TCHAR* argv[])
```

```
{	setlocale(LC_ALL, "Russian");
	Poli P1(10), P2(9);
	double k=3;
	double p=P1(k);
	cout <<"P1: " << P1;
	cout <<"Значение P1 при x=3: "<< p << endl;
	cout << "Введите P2: ";
	cin >> P2;
	Poli P4 = P1*k;
	bool l= P1<P2;
	cout << "P2: " << P2;
	Poli P6 = P2+k;
	cout << "P2+k: " << P6;
	cout << "P1*3: " << P4;
	cout <<"P1<P2: " << l << endl;
	Poli P3 = P2-P1;
	cout << "P2-P1: " << P3;
	Poli P5=P1+P2;
	bool a=(P5==P4);
	cout << "(P1+P2)==(P2-P1): " << a << endl;
	double t =P1[1];
	cout  << "P1[1]: " << t << endl;
	system ("pause");
	return 0;
}
```

7. Бином (есть)

```
#include <iostream>
using namespace std;
class Binomial
{
private:
	int a,b;
public:
	Binomial()
	{
		a = 0;
		b = 0;
	}
	Binomial(int x,int y)
```

```
{
        a = x;
        b = y;
}
Binomial(Binomial&B)
{
        a = B.a;
        b = B.b;
}
void setA(int x)
{
        a = x;
}
void setB(int y)
{
        b = y;
}
int getA()
{
        return a;
}
int getB()
{
        return b;
}
friend Binomial operator+ (const Binomial& B, const Binomial& C)
{
        Binomial D(B.a+C.a,B.b+C.b);
        return D;
}
friend Binomial operator- (const Binomial& B, const Binomial& C)
{
        Binomial D(B.a-C.a,B.b-C.b);
        return D;
}
friend Binomial operator* (const Binomial& B, const int C)
{
        Binomial D(B.a*C,B.b*C);
        return D;
}
friend ostream& operator << (ostream&out, Binomial C)
{
```

```
            if (C.b > 0)
            out << C.a << "x + " << C.b << "\n";
            else
            out << C.a << "x - " << -1*C.b << "\n";
            return out;
        }
};

int main()
{
        Binomial A(3,4), B(1,2), C;
        C = A+B;
        cout << C << endl;
        C = A - B;
        cout << C << endl;
        C = A*-3;
        cout << C << endl;
}
```

8. Матрица 3x3 целых чисел (есть)

```
class Matr33
{
private:
   int ** arr;
public:
   Matr33();
   Matr33(int**);
   Matr33(const Matr33&);
   ~Matr33();
   int def();
   Matr33 operator +(Matr33);
   Matr33 operator -(Matr33);
   Matr33 operator *(Matr33);
   Matr33 operator *(int);
   Matr33& operator =(Matr33);
   int* operator [](int);
   friend ostream& operator << (ostream&, Matr33);
   friend istream& operator >> (istream&, Matr33&);
};
```

```
Matr33::Matr33()
{
    arr = new int *[3];
    for (int i = 0; i < 3; i++)
        arr[i] = new int[3];
    for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        arr[i][j] = 0;
}


Matr33::Matr33(int ** brr)
{
    arr = new int *[3];
    for (int i = 0; i < 3; i++)
        arr[i] = new int[3];
    for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        arr[i][j] = brr[i][j];
}


Matr33::Matr33(const Matr33& b)
{
    arr = new int *[3];
    for (int i = 0; i < 3; i++)
        arr[i] = new int[3];
    for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        arr[i][j] = b.arr[i][j];
}


Matr33::~Matr33()
{
    for (int i = 0; i < 3; i++)
        delete[] arr[i];
    delete[] arr;
}


int Matr33::def()
```

```
{
    return
          arr[0][0] * arr[1][1] * arr[2][2]
        + arr[0][1] * arr[1][2] * arr[2][0]
        + arr[0][2] * arr[1][0] * arr[2][1]
        - arr[0][0] * arr[1][2] * arr[2][1]
        - arr[0][1] * arr[1][0] * arr[2][2]
        - arr[0][2] * arr[1][1] * arr[2][0];
}


Matr33 Matr33::operator +(Matr33 b)
{
    Matr33 temp(*this);
    for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        temp.arr[i][j] += b.arr[i][j];
    return temp;
}


Matr33 Matr33::operator *(int k)
{
    Matr33 temp(*this);
    for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        temp.arr[i][j] *=k;
    return temp;
}


Matr33 Matr33::operator -(Matr33 b)
{
    Matr33 temp(*this);
    for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        temp.arr[i][j] -= b.arr[i][j];
    return temp;
}

Matr33 Matr33::operator *(Matr33 b)
{
```

```
    Matr33 temp;
    for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
    for (int k = 0; k < 3; k++)
         temp.arr += arr[i][k] * b.arr[k][j];
    return temp;
}


Matr33& Matr33::operator =(Matr33 b)
{
    if (this == &b) return *this;
    for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
         arr[i][j] = b.arr[i][j];
    return *this;
}
int* Matr33::operator [](int i)
{
    if (i<0 || i>2) return NULL;
    return arr[i];
}
ostream& operator << (ostream& out , Matr33 m)
{
    for (int i = 0; i < 3; i++)
        {
        for (int j = 0; j < 3; j++)
        cout << m.arr[i][j] << " ";
        cout << endl;
        }
    return out;
}
istream& operator >> (istream& in, Matr33& m)
{
    for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
         cin >> m.arr[i][j];
    return in;
}
```

9. Время (есть)

class clock

```
{
        private :
                int h;
                int m;
                int s;
public :
        clock ()
  {h=1;
        m=1;
        s=1;}
clock (int p1, int p2, int p3)
{if (p2>=0 && p2<=59)
        m=p2;
else abort ();
if (p3>=0 && p3<=59)
   s=p3;
else abort ();
}

clock (clock &b)
{h=b.h;
 m=b.m;
 s=b.s;
}

int operator-( clock c2){
  clock temp; int r;
        temp.h = h*3600 + m*60 + s;
        temp.m = c2.h*3600 + c2.m*60 + c2.s;
        temp.s = abs(temp.h-temp.m);
        r=temp.s;
        cout<<r<<endl;
        return r;
}

void operator<( clock c2){
  clock temp; int r;
        temp.h = h*3600 + m*60 + s;
        temp.m = c2.h*3600 + c2.m*60 + c2.s;
        temp.s = abs(temp.h-temp.m);
        if (temp.h > temp.m)
                cout<<"pervoe chislo bolshe"<< endl;
```

```
	else
		cout<<"vtoroe chislo bolshe"<< endl;

}

clock operator++() {
	s++;
	if (s>=60){
		m++;
		s=0;}
	if (m>=60){
		h++;
		m=0;}
	return *this;
}

clock operator--() {
	s--;
	if (s<0){
		m--;
		s=59;}
	if (m<0){
		h--;
		m=59;}
	return *this;
}


void input ()
{	cin>>h>>m>>s;
}
void print  ()
{
	cout <<" h="<<h<<" m="<<m<<" s="<<s<<endl;
}



friend istream& operator >> (istream& in, clock& c)
{in >>c.h>>c.m>>c.s;
return in;}
```

```
friend ostream& operator << (ostream& out, clock& c)
{out << c.h<<" "<<c.m<<" "<<c.s<<endl;
return out;}

};



void main()
{       clock b1,b2;

    cin>>b1;
        cin>>b2;
        b1++;
        b2--;
        cout<< b1;
        cout<< b2;

        b1<b2;
        b1-b2;
}
```

10. Комплексное число (есть)

```
// ООП-лаба1.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <iostream>
#include <math.h>
using namespace std;
class Komp {
private:
        double ch,ab;
public:
Komp()
{
        ch=0.0;
        ab=0.0;
}
Komp (double n, double m){
        ch=n;
```

```
        ab=m;
if (m==0)
abort();
}
Komp (double r)
{
        ch=8.0;
        ab=r;
}
Komp (Komp &t)
{
        ch=t.ch;
        ab=t.ab;
}
bool srav (Komp t){
        if (ch==t.ch && ab==t.ab)
                return true;
        else
                return false;
        }
Komp add1(Komp t)
{
ch+=t.ch;
ab+=t.ab;
return Komp (ch,ab);
}
Komp add2(double a)
{
ch=ch+a;
return Komp (ch,ab);
}
Komp razn (Komp t)
{
ch-=ch*t.ch;
ab-=t.ab;
return Komp (ch,ab);
}
Komp ymnogenie1 (Komp t)
{
ch=ch*t.ch-ab*ab;
ab= -(ch*t.ab+ab*t.ch);
return Komp (ch,ab);
```

```
}
Komp ymnogenie2 (double a)
{
ch*=a;
ab*=a;
return Komp (ch,ab);
}
double modyl ()
{
double a;
a=sqrt(ch*ch+ab*ab);
return a;
}


void vivod () {
	cout <<ch<<" + "<<ab<<"i\n";
}
void vvod () {
	cout <<"vvedite rac i komp chast\n";
cin >>ch>>ab;
}

};


void main()
{

	Komp t1 (6,8);
	Komp t2  (3,13);
	Komp t3;
	t3=t1.add2(6);
	t3.vivod();
	t3=t1.add1(t2);
	t3.vivod();
	t3=t1.razn(t2);
	t3.vivod();
	t3=t1.ymnogenie1(t2);
	t3.vivod();
	t3=t1.ymnogenie2(5);
	t3.vivod();
```

```
        cout<<"modyl "<<t1.modyl()<<"\n";
        t3.vvod();
        t3.vivod();
        t2.vivod();
        t1.vivod();
}
```

11. Рациональная дробь

12. Точка в двумерном пространстве