Лабораторная работа №6.
Классы.

Основные сведения.

Класс – абстрактный тип данных, определяемый пользователем, и представляющий собой модель реального мира в виде данных и функций для работы с ними. Данные, содержащиеся в классе, называются полями, а функции – методами. Общий формат объявления класса выглядит следующим образом:

```
class имя
{
        private:
        // описание скрытых компонентов класса
        public:
        // описание доступных компонентов
};
```

Точка с запятой в конце описания – обязательный элемент.

Ключевые слова (спецификаторы) private, public управляют видимостью компонентов класса. Слово private (для класса этот спецификатор по умолчанию) запрещает обращение к полям класса из вне, а public (обязательный спецификатор) - объявляет поля как общедоступные. Все компоненты, описанные после public, часто называют интерфейсом класса. Количество и порядок следования спецификаторов стандартом не оговаривается.

Поля класса имеют следующие специфические особенности:

- могут иметь любой тип, кроме типа объявляемого класса, но могут быть указателями или ссылками на данный класс;
- могут быть объявлены с модификатором const, при этом инициализируются с помощью конструкторов специального вида;
- могут быть объявлены с модификатором static, но никак не auto, register или extern.

Инициализация полей класса при описании не допускается.

Пример:

```
class Small                                  //определение класса
{
        private:
           int small;                        // поле класса
        public:
           void setdata(int s)               //методы класса
             { small = s; }
           void showdata()
              { cout << "  Значение поля: " << small << endl; }
};
```

Этот простой класс содержит всего одно поле и два метода для доступа к нему. Первый метод позволяет присвоить полю некоторое значение, второй – выводит это значение на экран. Объединение данных и функций (инкапсуляция) является стержневой идеей объектно-ориентированного программирования.

Объявление объекта класса аналогично объявлению переменных стандартных типов, например,

```
int var_int;
double var_double;          // объявление переменных стандартных типов
Small small_1, small_2;     // объявление переменных (объектов) типа Small
```

Объект класса Small находится в таком же отношении к своему классу, в каком переменная находится по отношению к своему типу. Объект является экземпляром класса так же, как автомобиль является экземпляром класса средств передвижения.

Обращение к объектам класса допускается только через компоненты, описанные после спецификатора public. Например, для объявленного объекта small_1.setdata(38); производится присваивание значения 38 полю small. Непосредственное обращение к скрытому полю, например, small_1.small = 664;, запрещено. Сокрытие данных класса защищает данные класса от несанкционированного доступа через операцию присваивания или через функции, не принадлежащие данному классу.

Число полей и методов класса теоретически может быть любым.

Определение методов класса Small – setdata(int) и showdata() осуществлено непосредственно в самом классе. По умолчанию функции, определенные в теле класса, считаются как встроенные, то есть со спецификатором inline. Подобным образом желательно определять функции небольшого объема (2-5 строк), а функции, имеющие больший объем лучше выносить за пределы класса, оставляя внутри класса их прототипы. Общий формат определения функции за пределами класса выглядит следующим образом:
тип_результата имя_класса :: имя_функции(список параметров)

```
{
        // тело функции;
}
```

Для рассмотренного класса Small:

```
class Small
{
        private:
           int small;
        public:
           void setdata(int);
           void showdata();
};

void Small :: setdata(int s)
      { small = s; }
void Small :: showdata()
       { cout << "  Значение поля: " << small << endl; }
```

Подобный прием позволяет существенно уменьшить объем объявляемого класса, а вынесенные методы записать в заголовочные файлы и файлы реализации. Следует отметить, что методы, определение которых вынесено за пределы класса не являются встраиваемыми. Перед их определением необходимо указывать явно ключевое слово inline при необходимости.

Задание №1.

Определить простой класс, представляющий описание членов семьи, объявив поля для хранения фамилии, имени, отчества, возраста, пола, статуса (учащийся, рабочий, пенсионер). Объявить методы, позволяющие вводить данные для каждого член семьи, а также обеспечить вывод этих данных на экран. Попытайтесь обратиться к скрытым полям класса не через методы класса, а напрямую, объясните реакцию компилятора.

Локальные классы.

Класс может быть глобальным, то есть объявленным вне какого-либо блока и локальным, объявленным в пределах блока (например, функции или другого класса).

Некоторые особенности локальных классов:

- внутри локального класса можно использовать статические и внешние переменные, внешние функции из области (блока), в которой он описан. Автоматические переменные из данной области использовать нельзя;
- локальный класс не может иметь статических компонентов;

- методы локального класса могут быть описаны только в теле самого класса;

- если некоторый класс описан в теле другого, то доступ к элементам внешнего класса осуществляется по общим правилам.

Пример:

```
int func(int val)                    // определение функции
{
    class Test                       // локальный класс Test
    {
        int test;
    public:
        Test(){};
        Test(int t)
        {
            test = t;
        }
        int ret()
        {
            return test;
        }
};                                   // конец определения класса Test
    Test tst(val);                   // объявление объекта типа Test
    return(tst.ret());
    }
```

Необходимо отметить, что областью видимости класса Test является блок функции funct.

Аналогично можно определить класс в теле другого класса, общая схема подобного действия описана ниже.

```
class Extern    // определение внешнего класса
{
  private:
     //  приватные поля внешнего класса;
  public:
     // открытые поля и функции
     // …
     class Inter      // определение внутреннего класса
     {
                      // определение полей и методов локального класса
     };               // конец определения локального класса Inter
};                    // конец определения внешнего класса Extern
```

Задание №2.

Определить класс из задания №1 в теле произвольной функции, которая последовательно выводит данные о каждом члене семьи.

Расширьте определение класса из первого задания, добавив в определение внешнего класса локальный класс, содержащий элементы, описывающие доходы членов семьи.

Константные методы класса.

Среди методов класса необходимо выделить отдельный вид – константные методы. Эти методы объявляются с ключевым словом const, следующим сразу за списком формальных параметров. Константные методы (их чаще называют безопасными)

предназначены для работы с константными объектами определяемого класса, изменять значение полей которых не допускается. Они имеют специфические особенности:

- объявляется с ключевым словом const;
- не может менять значение полей класса;
- может вызываться для любых объектов, в том числе и не константных.

Для класса Small, рассмотренного в первом примере, безопасным методом может быть только showdata(), так как он только выводит значения полей, не меняя их значений. Для полноты использования константного метода дополним класс Small конструктором (специального вида функцией), который даст возможность инициализации константного объекта.

```
class Small
{
        private:
           int small;
        public:
          Small(int sm)
            { small = sm;}
           void setdata(int s)
            { small = s; }
           void showdata() const                // константный метод
             { cout << "  Значение поля: " << small << endl; }
};
```

Теперь мы вправе объявить константный объект класса Small:

const Small const_small(3864);

и вызвать метод const_small.showdata();.

Применять не константный метод по отношению к const_small нельзя, так вызов const_small.setdata(745); приведет к ошибке.

В отдельных изданиях константные методы называют привилегированными составляющими класса.

Задание №3.

Дальнейшая модификация класса семейных отношений состоит в переопределении (добавлении) методов, не меняющих значения полей класса в константные. Объявите константные объекты данного класса и испытайте действие константных и не константных методов. Оцените и объясните реакцию компилятора на подобные действия.

Конструкторы.

Инициализация полей класса при объявлении объектов данного типа производится с помощью составной функции класса, называемой конструктором. Имя этой функции идентично имени класса, а ее идентификатор может быть перегружен произвольное число раз. Это дает возможность для объектов класса применять инициализаторы с разными типами аргументов и в разных количествах. Если пользователь при определении класса не объявил ни одного конструктора, компилятор автоматически создает конструктор без параметров с заголовком Name(), где Name – имя создаваемого класса. Такой конструктор называется *конструктором по умолчанию*.

Вызов конструктора осуществляется автоматически при объявлении объекта. Если такой вызов является инициализатором объекта, то заключенные в круглые скобки аргументы могут быть размещены сразу же за идентификатором объявляемого объекта. Предположим, что пользователь определил класс для работы с комплексными числами Complex. Следующие примеры показывают способы инициализации объектов при их объявлении:

Complex complex_number_1 = Complex(23.4, -4.5);, что эквивалентно объявлению
Complex complex_number_1(23.4, -4.5);.
Complex complex_number_2 = Complex(); или
Complex complex_number_2; .
Определим основные свойства конструкторов:
- конструктор не возвращает значение, даже типа void;
- класс может содержать несколько конструкторов с разным количеством и типом параметров для разных видов инициализации объектов. Можно сказать, что конструкторы класса, перегружаемые функции;
- конструктор, не имеющий параметров, называется конструктором по умолчанию. Компилятор автоматически создает таковой, если пользователь не создал ни одного конструктора;
- конструкторы могут иметь параметры любого типа кроме типа создаваемого класса. Конструктор, имеющий единственный параметр – константную ссылку на определяемый класс называется конструктором *копирования (методом «поле за полем»)*. Компилятор автоматически создает такой конструктор, если пользователь не определил его самостоятельно;
- конструкторы не наследуются;
- конструкторы нельзя описывать с модификаторами const, virtual, static;
- конструкторы глобальных объектов вызываются до вызова функции main. Локальные объекты создаются сразу после того, как становится активна их область действия, например, при передаче объекта в качестве фактического параметра некоторой функции.

Примеры объявления конструкторов.

```
class Child
{
    private:
        char *Name;
        int Age;
   public:
        Child(char *name, int age)              // конструктор с двумя параметрами
        {
          Name = name; Age = age;
        }
        void Out()
        {
            cout << " Name = " << Name << " Age =  " << Age << endl;
        }
};

// объявление объектов типа Child
Child child_1(" Ваня ", 6), child_2(" Ира ", 15);
```

Объявление объектов без инициализации, например, Child child_2; для данного типа не возможно, так как уже описан конструктор с параметрами и компилятор не создает автоматически конструктор по умолчанию. Исправить эту «ошибку» можно, самостоятельно, добавив в тело класса, конструктор по умолчанию: Child(){};.

Задание №4.

Как уже говорилось, число конструкторов может быть произвольным. Расширьте класс семейных отношений объявлением 3-4 конструкторов, имеющих различное

количество параметров и различные их типы, позволяющих инициализировать объекты различными способами.

Список инициализации конструктора.

Инициализация полей объекта может проводиться не только с помощью операторов, содержащихся в теле конструктора, но также с помощью списка инициализации, который находится в заголовке конструктора. Список инициализации отделяется от заголовка двоеточием (:) и состоит из записей вида field(list), где field – идентификатор поля, а list – список выражений.

Объектные, константные и ссылочные поля класса инициализируются только с помощью списка инициализации!

Статические поля класса инициализируются вне класса!

Выполнение действий, определенных конструктором, состоит из обработки списка инициализации, а затем – тела конструктора.

Список инициализации обязателен также при определении производного класса от некоторого базового, если в базовом классе определен хотя бы один конструктор, требующий инициализации.

В качестве примера рассмотрим определение класса Child, в котором конструктор имеет список инициализации.

```
class Child
{
    private:
        char *Name;
        int Age;
   public:
        // конструктор с двумя параметрами и списком инициализации
        Child(char *name, int age): Name(name), Age(age){};
        // тело конструктора пусто
        void Out()
        {
            cout << " Name = " << Name << " Age =  " << Age << endl;
        }
};
```

Наличие в данном классе конструктора со списком инициализации не обязательно, так как нет ни константных, ни объектных и ни ссылочных полей. Следующий пример содержит константные и ссылочные поля.

```
class Pair
{
    private:
        const int Fix;                  // константное поле
        const int &Ref;                 // константное и ссылочное поле
    public:
        Pair(int fix, int ref): Fix(fix), Ref(ref){};
        void  Out()
          {
                cout << Fix + Ref;
          }
};
int Num = 64;
int main()
{
        Pair Twin(330, Num);
```

```
        Twin.Out();
        return 0;
}
```

Конструктор Pair::Pair(int,int) не может быть представлен в следующем виде

```
Pair(int fix, int ref)
{
        Fix = fix;
        Ref = ref;
}
```

Задание №5.

Определить собственный класс, моделирующий целочисленные (вещественные) величины. Предусмотреть наличие константных и ссылочных полей. Проанализировать сообщения компилятора при попытке инициализации их операторами в теле конструкторов.

Инициализация объектных полей.

Поле считается объектным, если оно имеет тип класса (структуры), определенного или объявленного ранее. Следующий пример показывает использование объектного поля.

```
class  Integral
{
    private:
        int Fix;
    public:
        Integral(int val)
        {
           Fix = val;
        }
        Integral()
        {
            Fix  = 10;
        }
        int  Get()
        {
                Return Fix;
        }
};

class Rational
{
    private:
        Integral Num, Den;              // объектные поля типа Integral
    public:
        Rational(int num, int) : Num(num){};
        void  Out()
        {
                cout << Num.Get() << " / " << Den.Get() << endl;
        }
};
// Объявление объектов типа Rational  с объектными полями
Rational Num(13,64);
```

Список инициализации конструктора класса Rational трактуется так, как если бы он был записан в виде Num(num), Den().

Задание №6.

Определить класс комплексные числа Complex, объектными полями которого являются поля Re и Im типа Float (вещественный тип), объявленный ранее. Определить конструктор со списком инициализации, обеспечивающим инициализацию объявленных полей. Проанализируйте реакцию компилятора при попытке инициализации полей другим способом.

Копирующие конструкторы.

Среди конструкторов данного класса особую роль выполняют копирующие конструкторы. Предположим, что имя класса – Name, тогда конструктор вида Name::Name(const Name &), есть копирующий конструктор.

Если программист не определяет подобный конструктор явно, компилятор формирует его автоматически. Использование предполагаемого копирующего конструктора приводит к инициализации методом «поле – за - полем» всех полей инициализируемого объекта класса.

Пример использования предполагаемого конструктора.

```
class Test
{
	int test;
public:
	Test(int t)
	{
		test = t;
	}
	void Out()
	{
		cout << " Test: " << test << endl;
	}
};

int main()
{
	Test tst_1(100), tst_2 = tst_1;
	tst_1.Out();
	tst_2.Out();
	return 0;
}
```

Как видно, в классе Test не определен копирующий конструктор в явном виде, однако, есть возможность инициализировать новый объект tst_2 инициализируется полями уже существующего объекта tst_1. Это возможно потому, что компилятор создал свою версию конструктора по методу «поле – за – полем»:

```
Test(const Test &t)
{
	Test = t.test;
}
```

, или с использованием списка инициализации:

```
Test(const Test &t) : test(t.test){};
```

Наберите этот пример, проанализируйте результаты работы программы.

Задание №7.

Для класса, оперделенного в задании №6, определить копирующий конструктор, инициализирующий все поля объявляемого объекта.

Если в классе определено несколько полей, их нициализация производится последовательно. Рассмотрим пример класса Student, имеющий три поля различного типа.

```
class Student
{
        char *Name;
        int Age;
        double Ball;
        // остальные поля
public:
        //
        Student(const Student &);
        // остальные методы
};
Student :: Student(const Student &st)
{
        Name = st.Name;
        Age = st.Age;
        Ball = st.Ball;
}
```

Попытайтесь самостоятельно определить аналогичный конструктор, но со списком инициализации.

Копирующие конструкторы вызываются автоматически в случае если объявляемый объект инициализируется уже существующим, а так же в случае передачи параметра типа класс в какую-либо функцию или же функция возвращает объект типа класс. Для класса Student определим внешнюю функцию, в качестве ее парамеира передадим ссылку на объект типа Student.

```
class Student
{
 public:
        char *Name;
        int Age;
        double Ball;
        // остальные поля
        Student(const Student &);
        // остальные методы
};
Student :: Student(const Student &st)
{
        Name = st.Name;
        Age = st.Age;
        Ball = st.Ball;
}
void ExtFunc(Student st)
{
        cout << " Name: " << st.Name << ' '
             << " Age: " << st.Age << ' '
             << " Ball: " << st.Ball << endl;
}
```

Заметьте, что все поля класса объявлены как открытые. Это связано с тем, что обращение вида st.Name недопустимо если поле объявлено под спецификатором private. Прблема решается открытием полей или же объявлением методов, возвращающих значения закрытых полей класса.

Задание №8.

Для класса, определенного в задании №6, определит внешнюю функцию, возвращающую в качестве результата объект типа Complex. В копирующем конструкторе предусмотреть вывод «Работает копирующий конструктор», который позволит убедится в в каких случаях вызывается данный вид конструктора.

Преобразования, задаваемые конструктором.

Поскольку каждый конструктор задает преобразование в своем классе, следовательно, если в некоторм месте программы используется выражение Exp, представляющее данное, которое должно быть преобразовано к типу класса, то сразу же после обработки этого выражения неявно активизируется конструктор с инициализатором (Exp), что выполняет необходимые преобразования. Конструктором преобразования считается любой конструктор, имеющий один параметр, тип которого отличен от типа определяемого класса. Следующий пример показывает использование конструктора преобразования.

```
class Test
{
	int test;
public:
	Test(){};
	Test(int t)
	{
		cout << " Конструктор преобразования" << endl;
		test = t;
	}
	Test(const Test &t):test(t.test)
	{
		cout << " Копирующий конструктор " << endl;
	}

	void Out()
	{

		cout << " Test: " << test << endl;
	}
};


int main()
{
	setlocale(0,"RUS");
	Test tst;
	tst = 300;	// или tst = Test(300);
	tst.Out();
```

```
	return 0;
}
```
Преобразующий конструктор класса Test:
```
	Test(int t)
	{
		cout << " Конструктор преобразования" << endl;
		test = t;
	}
```
осуществляет преобразование параметра типа int к типу Test и оператор tst = 300; в теле основной функции трактуется как оператор tst = Test(300);. Такое преобразование называется типичным.

В некотрых случаях преобразования, задаваемые конструкторами можно запретить, так как они могут привести к нежелательным результатам. Для этого перед именем конструктора записывается стандартное слово explicit, которе предупреждает компилятор о том, что при объявлении некоторго объекта и инициализации его значением не относящимся к типу определяемого класса, неявных преоборазований производить не следует.

Рассмотрим еще раз класс Test

```
class Test
{
public:
	int test;
public:
	Test(){};
	explicit Test(int t)
	{
		cout << " Конструктор преобразования, запрещающий
			неявные преобразования типа int  в тип Test" << endl;
		test = t;
	}
	Test(const Test &t):test(t.test)
	{
		cout << " Копирующий конструктор " << endl;
	}

	void Out()
	{

		cout << " Test: " << test << endl;
	}
};


int main()
{
	setlocale(0,"RUS");
	Test tst_1(38);
	Test tst_2 = tst_1;
	Test tst_3 = 64;		// Здесь ошибка!
	return 0;
```

```
}
```

Откомпилируйте этот фрагмент и посмотрите сообщения компилятора. В операторе Test tst_3 = 64; компилятор не сможет преобразовать тип int к типу Test. Ошибку можно «исправить», если использовать явное преобразование типа Test tst_3 = (Test)64; , если это необходимо. Этот пример не единственный, более того, он очень прямолинеен. Больших неприятностей можно ожидать в случае передачи некоторой внешней функции параметра типа определяемого класса. Например,

```
void FunOut(Test t)
{
        cout << " Test: ";
        t.Out();
}
```

Если вызвать эту функцию с объектом типа Test, например, FunOut(tst_1),никаких проблем не возникнет. Еси же вызвать с переменной типа int, например, FunOut(300);, то неявно вызывается конструктор преобразования, котрый приводит число 300 к типу Test. Если перед конструктором стоит спецификатор explicit, неприятностей преобразования не будет, так как компилятор выдаст соответствующее сообщение.

Задание №9.

Для класса комплексных чисел добавить конструктор с одним аргументом, инициализирующий действительную или мнимую часть комплексного числа, запрещающий преобразования.

Переменная this.

Возможность неквалфицированного обращения в теле составной функции (метода) класса к компонентам класса обуславливается следующими утверждениями:

- в теле каждой нестатической составной функции класса с именем Name доступна переменная this типа Name с присвоенным ей указателем того объекта класса Name, применительно к которому вызвана данная составная функция;
- каждое неквалифицированное обращение к нестатическому компоненту класса Nane, например Field, трактуется как сокращенная запись выражения this->Field.

Рассмотрим модифицированный класс Student, используя переменную this.

```
class Student
{
 private:
        char *Name;
        int Age;
        double Ball;
public:
        Student(char *, int, double);
        void Out();
};

Student :: Student(char *Name, int Age, double Ball)
{
        this->Student::Name = Name;
        this->Student::Age = Age;
        this->Student::Ball = Ball;
}
void Student::Out()
{
        cout << " Name:" << this->Student::Name
```

```
                << " Age:" << this->Student::Age
               << "Ball:" << this->Student::Ball << endl;
}
```

Объявления вида this->Student::Name = Name; интерпретируются легко: this- указатель на объект, применительно которого вызывается данный конструктор, Student::Name – опреатор разрешения области видимости компоненты Name класса, последнее слово Name – это имя аргумента конструктора.

Этот пример искусственно усложнен, его более читаемый вид:

```
Student :: Student(char *name, int age, double ball)
{
        Name = name;
        Age = age;
        Ball = ball;
}
```

Использование переменной this здесь так же не обязательно. Необходимость в ней возникает в случае если на момент написания кода программы фактический объект - получатель данного метода не известен, например, при перегрузке операций.

Задание №10.

Для класса комплексные числа определить методы, использующие переменную this.

Дружественные функции и классы.

Функции, объявленные в классе, делятся на составные и дружественные. Каждая нестатическая функция, объявленная со спецификатором friend, является дружественной классу. Такая функция имеет доступ ко всем компонентам класса, к которым имеют доступ составные функции класса. Рассмотрим пример.

```
class Fixed
{
        Int Fix;
public:
        Fixed(int val):Fix(val){};
        friend void Show(Fixed);
};

void Show(Fixed Arg)
{
        cout << Arg.Fix;
}

 int main()
{
        Fixed Num(100);
        Show(Num);
        return 0;
}
```

Поскольку функция Show дружественная для класса, в ней можно использовать идентификатор приватного поля этого класса.

Использовать дружественные функции следует осторожно, так как они нарушают инкапсуляцию полей класса.

Составная функция некоторого класса может быть дружественной по отношению к другому классу. Это достигается путем передачи составной функции параметра типа класс, по отношению к которому она должна быть дружественной.

Задание №11.

Определить два простых класса, объявив составную функцию одного из них, дружественной для другого. Например,

```
class Test2;                    // прототип класса Test2
class Test1
{//
public:
        void fun(Test2);
};
class Test2
{//
public:
        friend void Test1::fun(Test2);
}
```

Все составные функции одного класса можно сделать дружественными по отношению ко второму, если в описании второго объявить дружественный класс. Например, для задания №11 класс Test2 определить дружественным классом по отношению к Test1 по следующей схеме:

```
class Test2;
class Test1
{//
public:
        friend class Test2;
        //
};
class Test2
{
        Test1 tst1;
public:
        fun_1();
        fun_2();
};
```

Дружественные функции используются для придания объектам класса свойств не характерных их типу, например, ввод из стандартного потока и вывод в стандартный поток. Типовой пример вывода показан ниже.

```
#include<ostream>
class Name
{
        int name;
public:
        Name(int val)
{
        name = val;
}
//
friend ostream &operator <<(ostream &, const Name &);
};
ostream &operator <<(ostream &out, const Name &n)
{
```

```
        out << " Message: " << n.name << endl;
}
```

Задание №12.

Для задания №11 класс Test1 дополнить дружественными функциями – операциями ввода и вывода.

Перегрузка операций.

Функцией-операцией является каждая функция с именем operator #, где # представляет одну из следующих операций: new, delete, +, - , *, /, %, &, !, = , <, >, !=, (), [], -> и так далее.

Функции-операции могут быть одноместные (унарные) и двуместные (бинарные), при этом число операндов, приоритет и ассоциативность определенной функции, такие же, как в базовом языке.

Не разрешается перегружать следующие операции: *, sizeof, #, ##, ::, ?:.

Необходимо, что специальные операции, а именно =, [], () были нестатическими составными функциями класса. Среди них особого внимания заслуживает операция присваивания, так как она не подлежит наследованию. Если пользователь не переопределяет самостоятельно операцию присваивания, то неявно создается предполагаемое присваивание

Class &Class::operator =(const Class &),

реализующее присваивание методом «поле – за - полем».

Функции-операции можно определять тремя способами: как составную функцию класса, как внешнюю функцию или как дружественную. В двух последних случаях функции передается хотя бы один параметр типа класс, указатель или ссылка на класс.

Перегрузка унарных операций.

Унарная функция-операция, определяемая внутри класса, должна быть представлена с помощью нестатического метода без параметров. Рассмотрим пример простого класса.

```
class Test
{
        int test;
public:
        Test(){};
        Test(int tst)
        {
                test = tst;
        }
        Test &operator ++()
        {
                ++test;
                return *this;
        }
        void Out()
        {
                cout << " Test: " << test << endl;
        }
};

int main()
{
        Test tst(200);
```

```
        ++tst;
        tst.Out();
        return 0;
}
```

Задание №13.

Переопределите самостоятельно для данного класса унарную префиксную операцию как дружественную и как внешнюю.

Постфиксную унарную операцию отличают от префиксной путем передачи пустого параметра типа int, причем, если операция определяется как внешняя, этот параметр должен быть вторым. Для класса Test:

```
Test operator ++(Test &tst, int)
{
        tst.test++;
        return tst;
}
```

Обратите внимание на то, что для работы такой внешней функции, необходимо открыть доступ к полю test.

В продолжение задания №13, определите унарную постфиксную операцию как дружественную классу Test.

Перегрузка бинарных операций.

Если бинарная операция определяется как нестатическая составная функция класса, то она должна иметь один параметр, как правило, константная ссылка на объект определяемого класса. Этот параметр представляет собой правый операнд, а вызвавший объект считается левым операндом операции.

Для класса Test перегрузка операции сложения может быть представлена следующим образом:

```
Test Test:: operator+(const Test &tst)
{
        return this->test+tst.test;
}
```

Определить самостоятельно перегрузку бинарных арифметических операций как внешнюю и дружественную функцию, с учетом того, что передавть им необходимо два параметра типа класс.

Бинарные арифметические операции всегда должны возвращать в качестве результата объект типа класс. Операции сравнения возвращают в качестве результата объект типа bool. Дополните класс Test операциями сравнения.

Важное место в определении класса является перегрузка операции присваивания. При определении данной операции следует руководствоваться следущими рассуждениями:

-эту операцию имеет смысл переопределять в том случае, если класс содержит поля, память под которые выделяется динамически;

-если программист не оперделял операцию самостоятельно, компилятор автоматически сгенерирует ее код как поэлементное копирование полей одного объекта в другой;

-операция не наследуется в производных классах;

-операцию можно определить только как составную функцию класса.

Разберите следующий пример, обратив особое внимание на операцию присваивания.

```
class Text
{
        int Len;
        char *Ref;
public:
```

```
        Text()
        {
                Len = 0;
                Ref = new char[Len];
        };
        Text(char *Str)
        {
                Ref = new char[Len=strlen(Str)+1];
                strcpy_s(Ref,Len,Str);
        }
        ~Text()
        {
                delete Ref;
        }
        Text &operator =(const Text &);
        void Out()
        {
                cout << " Out text = "  << Ref << endl;
        }
};

Text &Text::operator =(const Text &Par)
{
        if(this == &Par) return *this;
        else
        {
                delete Ref;
                Ref = new char[strlen(Par.Ref)+1];
                strcpy_s(Ref,Len-1,Par.Ref);
        }
        return *this;
}

int main()
{
        Text head("Text Text"), tail;
        tail = head;
        tail.Out();
        return 0;
}
```

Операция strcpy_s, спецефичная для стандарта языка С, в последних версиях VC может работать не корректно.

Задание №14.

Определите собственный класс, содержащий поля-указатели на стандартные типы, требующие выделения места в динамической области памяти. Переопределить операцию присваивания для данного класса.

Перегрузка операции индексирования.

Операция индексирования [] так же специфична и ее перегружают в тех случаях, когда тип класса представляет собой некое множество, для которого индексирование имеет определенный смысл. Например, в классе, моделирующем работу массива. Необходимо учитывать то, что эта операция возвращает в качестве результата объект того типа, который содержится во множестве.

Рассмотрим класс Vector, описывающий модель стандартного массива.

```
#include<iostream>
#include<stdlib.h>
using namespace std;

class Vector
{
protected:
	int size;
	int *p;
public:
	explicit Vector(int n=5);
	Vector(const int arr[], int n):size(n)
	{
		p = new int[size];
		for(int i=0; i<size; i++)
			p[i] = arr[i];
	}
	~Vector()
	{
		delete []p;
	}
	int &operator [](int);
	void Out()
	{
		for(int i=0; i<size; i++)
			cout << p[i] << ' ';
		cout << endl;
	}
};
int &Vector::operator [](int i)
{
	if(i<0 || i>size)
	{
		cout << " Выход за границы массива! " << endl;
		exit(0);
	}
	return p[i];
}


int main()
{
	setlocale(0,"RUS");
	int array_int[10]={23,-45,6,0,435,-6};
	Vector vector(array_int,10);
	vector.Out();
```

```
        cout << vector[1] << endl;
        cout << vector[11] << endl;

        return 0;
}
```

В этом примере показана модель стандартного массива, котрая не в полной мере воспроизводит реальный массив. Например, объявление объекта класса Vector, по аналогии с обычными массивами Vector vector[10]={1,2,3,4,5}; , в данном случае приведет к ошибке.

Задание №15.

Определите класс однонаправленный список, к элементам которого можно обращаться по индексу.

Перегрузка операции доступа к компонентам класса ->.

Эта операция используется для доступа к полям и методам класса. Обращение x -> m трактуется как (x.operator())->m, причем x – это объект, а не указатель на него. Операция возврвщает указатель на объект , либо ссылку на объект класса, поскольку оригинальное значение операции -> не теряется, а только задерживается. Перегрузка операции -> допускается только через нестатическую составную функцию класса.

Рассмотрим подробнее определение этого оператора.

```
struct Test
{
        int test;
        Test(){};
        Test(int t)
        { test = t;}
        Test *operator ->()
        {
                cout << " Собственный оператор -> " << endl;
                return this;
        }
        void Out()
        {  cout << " Test: " << test << endl;  }

};

int main()
{
        int int_obj = 100;
        Test tst(int_obj);
        cout << tst->test << endl;      // или cout << (tst.operator->())->test << endl;
        tst->Out();                     // или (tst.operator->())->Out();
        return 0;
}
```

Ипользование ключевого слова struct вместо class связано с тем, что обращение к скрытым полям (public:)в классе не допускается и доступ ptr_tst->test приведет к ошибке.

Следует отметить, что этот оператор может быть перегружен только как нестатический метод класса.

Обратите внимание на следующий момент, если объявить указатель на объект класса и вызвать операцию доступа, работать будет стандартная операция. Оцените это на следующем примере:

```
Test *ptr_Test = new Test(int_obj);
 cout << ptr_Test->test << endl;
```

Задание №16.

Для класса комплекных чисел переопределить оператор доступа, возвращающий значения действительной и мнимой части комплексного числа.

Для любопытных: перегрузку операции ->* - выделение указателя на компоненту можно посмотреть в Microsoft Developer Network (MSDN).

Перегрузка операций new и delete.

Перегрузку операций new и delete выполняют для обеспечения альтернативности управления динамической памятью. Эти функции должны соответствовать требованиям описаным в международном стандарте ISO/IEC 14882. К особенностям операций следует отнести:

1. -им не требуется передавть параметр типа класс;
   - первым параметром функции new должен передаваться размер объекта типа size_t (тип определен в заголовочном файле stddef.h), который при вызове передается неявно;
   - операция new возвращает *void;
   - операция delete возвращает результат типа void;
   - опрерация delete имеет первый параметр типа *void;
   - операции new и delete, определенные как составные функции класса, по умолчанию считаются статическими функциями.

   Перегружать можно как глобальные операции new и delete, так и для определяемых типов данных. Рассмотрим типовой пример перегрузки глобальных операций.

```
#include<iostream>
#include<stddef.h>
#include<malloc.h>

using namespace std;

void* operator new(size_t sz)
{
        cout << "Оператор new " << " Байт: " << sz << endl;
        void* m = malloc(sz);
        if(!m) cout << " Нехватка памяти " << endl;
        return m;
}
void operator delete(void* m)
{
          cout << "Оператор delete" << endl;
          free(m);
}

int main()
{
        setlocale(0,"RUS");

        char *ptr_char =  new char;
        int *ptr_int = new int;
        double *ptr_double = new double;
```

```
        delete ptr_char;
        delete ptr_int;
        delete ptr_double;

        return 0;
}
```

Протестируйте данный пример, оцените результаты. Следующий пример показывает перегрузку операций внутри класса.

```
#include<iostream>
#include<stddef.h>
#include<malloc.h>

using namespace std;

class Test
{
        int test;
public:
        Test(){};
        Test(int tst):test(tst)    {};
        void *operator new(size_t);
        void operator delete(void*);
        void Out()
        {
                cout << " Test: " << test << endl;
        }

};

void* Test::operator new(size_t size)
{
        cout << " New операция для класса Test, необходимое число байт: " << size
<< endl;
   void *storage = malloc(size);
   if(NULL == storage)
        {
                cout << " Нехватка памяти " << endl;
        }
        return storage;
}


void Test::operator delete(void *p)
{
        cout << " Delete операция для класса Test " << endl;
        free(p);
}

int main()
{
        setlocale(0,"RUS");
```

```
        // Вызов собственной операции new
        Test *ptr_Test_1 = new Test(100);
        ptr_Test_1->Out();
        // Вызов глобальной операции new
        Test *ptr_Test_2 = ::new Test(200);
        ptr_Test_2->Out();
        // Вызов собственной операции delete
        delete ptr_Test_1;
        // Вызов глобальной операции delete
        ::delete ptr_Test_2;

        return 0;
}
```

Операции new и delete могут иметь другие форматы списка параметров, описанные в стандарте ISO/IEC 14882.

Задание №17.

Для класса комплексных чисел переопределить свои собственные операции new и delete.

Указатели на компоненты класса.

На любую нестатическую компонента класса можно содать указатель. Этот указатель отличаетстся от обычного наличием спецификатором типа класс. В частности, указатель на тип int объявляется следующим образом: int *, а указатель на поле целого типа класса Class – int Class::*.

Примеры:

int Fix;	int Class:: *
float *Num;	float Class::*
long (*Ref)[2];	long (*Class::*)[2]
int Sub(int);	int (Class::*)(int)

Благодаря связи компонента класса с идентификатором класса возможно осуществление контроля правильности обращения к компонентам класса через указатели.

С точки зрения синтаксиса обращений применяется простая нотация: если Ptr – выражение, указывающее на компоненту класса, то *Ptr – имя этого компонента. Это приводи к тому, что если Obj имя объекта, а Ref – указатель на объект класса, справедливы следующие записи:

Obj.*Ptr; и
Ref->*Ptr;

Рассмотрим пример указателей на поля класса.

```
#include<iostream>
using namespace std;

class Test
{
public:
        int Tst;

};
// Указатель на поле целого типа класса Test
int Test::*ptr_Test = &Test::Tst;
```

```
// Похоже на объявление обычного указателя int *Ptr_Test = &Tst; , только
// присутствует имя класса Test::
int main()
{
	setlocale(0,"RUS");

	// Объект класс Test
	Test Num;
	Num.Tst = 63;
	cout << " Обычным способом через объект класса: " << Num.Tst << endl;
	cout << " Через указатель на компоненту: " << Num.*ptr_Test << endl;

	// Указатель на объект типа Test
	Test *Ptr_Test = &Num;
	Ptr_Test->Tst = 38;
	cout << " Обычным способои через указатель на объект класса: " << Ptr_Test-
>Tst << endl;
	cout << " Через указатель на компоненту: " << Ptr_Test->*ptr_Test << endl;

	return 0;
}
```

Внимательно изучите этот пример, возможно потребуется вспомнить указатели на стандартные типы данных. Создать указатель на закрытое (защищенное) поле класса можно, но обратиться аналогичным способом не разарешит компилятор.

А теперь разберемся с указателями на методы класса.

```
#include<iostream>
using namespace std;

class Test
{
	int Tst;
public:
	Test(){};
	Test(int tst):Tst(tst){};
	int *TestPtr()
	{
		return &Tst;
	}
	int TestSumm(int arg)
	{
		Tst += arg;
		return Tst;
	}
};

// Указатель на метод TestSumma класса Test
int (Test::*Ptr_TestSumm)(int) = &Test::TestSumm;
// Указатель на метод *TestPtr() класса Test
int *(Test::*Ptr_TestPtr)() = &Test::TestPtr;
```

```
int main()
{
	setlocale(0,"RUS");
	// Для метода TestSumm(int);
	Test Num(100);
	// Через объект класса
	cout << (Num.*Ptr_TestSumm)(30) << endl;
	// Через указатель
	Test *ptr_Test = &Num;
	cout << (ptr_Test->*Ptr_TestSumm)(50) << endl << endl;

	// Для метода *TestPtr()
	// Через объект класса
	cout << *(Num.*Ptr_TestPtr)() << endl;
	// Через указатель
	cout << *(ptr_Test->*Ptr_TestPtr)() << endl;
	return 0;
}
```

Нельзя создать указатель на конструктор класса, следущее выражение компилятор посчитает ошибочным: (Test::*PtrTest)(int) = &Test::Test(int);. Нельзя так же создать указатели на статические компоненты класса.

Задние №18.

Для класса компексные числа определит указатели на поля и методы класса. Показать их использование.