Методы класса Z не имеют права обращаться к защищенным членам класса X.

Friend- метод другого класса:

Можно дать права не всем, а выборочно некоторым методам другого класса обращаться к защищенным переменным данного класса, например:

```
class Rect{

      friend Circle::Circle(const Rect&); //только данный метод
                                          класса Circle имеет право
                                          обращаться к защищенным членам
                                          класса Rect => в любом другом
                                          методе при аналогичной попытке
                                          компилятор выдаст ошибку

};
```


Тема IV. Перегрузка операторов.

Перегрузка операторов для базовых и пользовательских типов

Компилятор генерирует **разные** низкоуровневые инструкции, встречая одно и то же действие, но над разными типами.

Для базовых типов также существует перегрузка операторов, которую осуществляет компилятор самостоятельно (встроенные операторы):

```
int x,y,z;
z=x+y;      //add


double x,y,z;
z=x+y;      //fadd
```

Язык C++ допускает перегрузку операторов (overloading), то есть способность переопределения привычных операторов типа **+, -, *, >, <, =, >>, <<, [], ()**, ... но только для Ваших пользовательских типов данных (полный список операторов, которые можно перегружать, приведен в **MSDN Library – "Redefinable Operators"**. Там же перечислены операторы, которые перегружать нельзя).

Но, в отличие от базовых типов, компилятор «не знает» как интерпретировать следующую запись и выдаст ошибку: «в классе не определен оператор +»

```
Point pt1, pt2, pt3;
pt3 = pt1 +pt2;   //ошибка
```

Только Вы можете определить, каким образом следует поступать компилятору, когда он встречает в тексте программы оператор, который должен воздействовать на Ваш тип. Целью перегрузки операторов является простота и интуитивная интерпретация операторов.

Специфика перегружаемых операторов.

Перегрузка операторов – один из видов перегрузки имен функций. Просто у такой функции несколько необычное имя. Если Вы перегружаете оператор ♥, то должны создать в Вашем классе функцию с именем **operator♥**, которую компилятор будет вызывать всякий раз, когда по отношению к объекту данного класса применяется указанный оператор ♥.

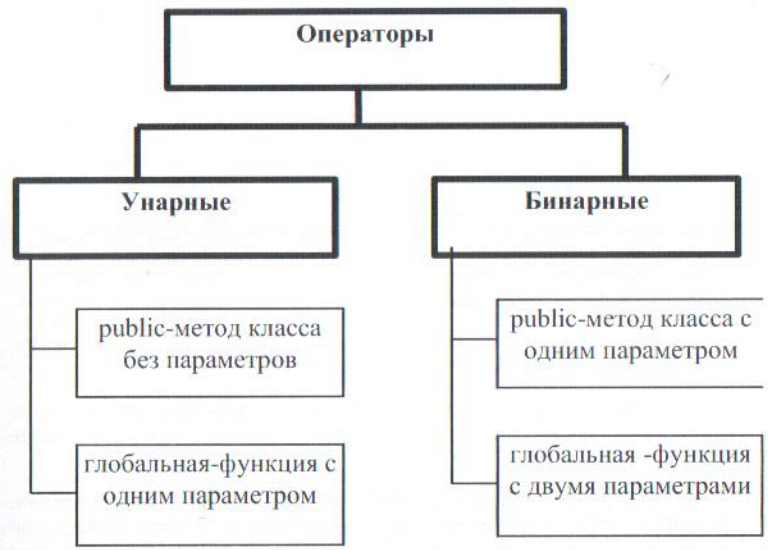
Но так как функция эта все же специфичная, вводятся некоторые дополнительные правила:

- нельзя создавать собственные операторы, а можно только перегружать существующие (и то не все);
- перегруженный оператор действует только по отношению к объектам того класса, для которого он переопределен (например, operator+ класса A не имеет никакого отношения к operator+ класса B);
- нельзя менять число операндов оператора (например, нельзя перегрузить оператор * (умножения) таким образом, чтобы в нем использовался один операнд);
- перегруженные операторы наследуют приоритеты и ассоциативность от встроенных операторов;
- оператор перегружается только относительно пользовательского типа данных (обычно класса) => **нельзя перегружать встроенные операторы** (например, оператор целочисленного сложения). Чтобы обеспечить выполнение этого условия, компилятор требует, чтобы хотя бы один аргумент каждого перегруженного оператора относился к пользовательскому типу;
- нельзя перегружать операторы **. :: .* ? :**
- нет правил, которые предписывали бы сохранение смысла оператора, но по возможности рекомендуется это делать (никто не запрещает вложить в реализацию оператор сложения смысл оператора вычитания, но вряд ли это улучшит «читабельность» Вашего кода);
- как и любая другая функция, перегруженный оператор может иметь сколько угодно реализаций, «различимых» по типу параметров компилятором;
- оператор, перегруженный с помощью метода класса, может быть виртуальным и даже чисто виртуальным;
- оператор, перегруженный с помощью метода класса, не может быть статическим;
- не существует ограничений на тип возвращаемого значения, но следует учитывать «преемственность» использования перегружаемого оператора применительно к базовым типам и эффективность реализации.

Способы перегрузки операторов

Для того, чтобы перегрузить оператор ♥, необходимо объявить и определить функцию с именем **operator♥.** Эта функция может быть глобальной или методом класса (но не обоими вариантами сразу).

Все переопределяемые операторы делятся на унарные (действие производится над одним объектом) и бинарные (действие производится над двумя объектами). Тернарный оператор перегружать запрещено.



Пример:

```
class A{...};

A x,y,z;

z = x+y;     //нормальная     синтаксическая     форма     записи.
                              Компилятор сгенерирует или вызов
                              public метода operator+ класса A,
                              или вызов глобальной функции (в
                              зависимости от реализации)

z = operator+(x,y);      //функциональная форма вызова глобальной
                                 функции
```

```
z = x.operator+(y);     //функциональная форма вызова метода класса
```

Рекомендация: если оператор может быть перегружен как глобальной функцией, так и методом класса – предпочитайте перегрузку в форме метода класса!

Исключения (когда перегрузка методом класса невозможна):

- первый операнд относится к базовому типу, например `z = 1+x;`
- тип первого операнда библиотечный

В таких ситуациях перегрузка возможна только посредством глобальных функций.

Порядок поиска компилятором перегруженного оператора

Если компилятор встречает оператор, он должен решить, как его компилировать. Например, для бинарного оператора:

- если оба аргумента относятся к базовым типам, используется встроенный оператор;
- если слева – операнд пользовательского типа, компилятор ищет оператор в форме метода класса. Если находит, генерирует вызов метода класса;
- если перегрузки в форме метода не найдено или слева операнд базового типа, компилятор ищет перегрузку в форме глобальной функции. Если такая форма существует, использует ее при компиляции;
- если все перечисленные варианты испробованы, а перегрузки не найдено, выдает ошибку.

Замечание: если оператор перегружен и методом, и глобальной функцией, то при нормальной форме вызова компилятор не может различить какую именно форму перегрузки требуется вызвать. В такой ситуации некоторые компиляторы:

а) выдают ошибку двойственность => ошибка компилятора

б) другие компиляторы могут предпочесть метод => глобальная функция будет проигнорирована.

Перегрузка операторов методом класса

Замечание: операторы =, (), [], -> могут быть перегружены только с помощью метода класса.

operator= (memberwise assignment).

В качестве примера рассмотрим перегрузку оператора «=» (присваивания). Оператор присваивания – наиболее часто встречаемый оператор класса. Он настолько тесно связан с понятием класса (так же, как конструктор копирования), что если Вы явно не реализуете такой оператор в классе, то компилятор сгенерирует автоматический оператор присваивания сам. Такой автоматический оператор присваивания «умеет»:

- поэлементно копировать данные базового типа из одного объекта в другой;
- вызывать оператор присваивания базового класса (явно определенный программистом или автоматический)
- вызывать оператор присваивания для встроенных объектов.

Компилятор не может сгенерировать автоматический оператор присваивания в следующих случаях:

- в классе объявлен константный объект;
- в классе объявлена ссылка;
- в базовом классе оператор присваивания объявлен private;
- во встроенном объекте оператор присваивания объявлен private.

Следствие: для сложных классов пишите свои операторы присваивания.

Замечание: оператор присваивания не наследуется! Это означает: если в производном классе оператор присваивания явно программистом не определен, то компилятор генерирует автоматический оператор присваивания, а не использует метод базового класса.

Замечание: оператор присваивания (также называемый оператором копирующего присваивания – memberwise assignment) очень похож на конструктор копирования, но существует принципиальная разница:

```
Point pt1, pt2;
```

```
Point pt3 = pt1;  //объявление с инициализацией - создается новый объект - вызывается конструктор копирования

pt2 = pt1;  //одному существующему объекту присваивается значение другого существующего объекта - вызывается оператор присваивания
```

В качестве примера перегрузим оператор присваивания для класса Point, хотя для такого простого класса компилятор прекрасно бы сгенерировал автоматический оператор присваивания (и, возможно, эффективнее, чем сделаем это мы):

```
int main()
{
    Point pt1(1,1), pt2(2,2);
    pt2 = pt1;  //нормальная форма вызова. Если в классе Point не перегружен оператор присваивания, то компилятор вызовет автоматический (при этом содержимое pt1 копируется в pt2, затирая прежние значения pt2), а если Вы перегрузили оператор присваивания, то компилятор вызовет Ваш метод класса и будут выполнены действия, предусмотренные программистом
    pt2.operator=(pt1);     //функциональная форма вызова
}
```

Замечание: для компилятора обе формы эквивалентны. Программисту привычнее писать выражения в нормальной форме, но в большинстве случаев для того, чтобы понять, какого типа параметры должна принимать перегруженная функция и каков тип возвращаемого значения, стоит написать функциональную форму вызова.

```
class Point{
    int m_x, m_y;
public:
```



```
    Point (int x, int y){m_x = x; m_y = y;}
    Point& operator = (const Point& refPt);  //объявление оператора присваивания. Ключевое слово const необязательно, но говорит о хорошем стиле программирования. В этом случае компилятор не позволит модифицировать параметр (то есть значение справа от знака равенства)
    ...
};
//Реализация оператора присваивания
Point& Point::operator = (const Point& refPt)
{
    m_nX = refPt.m_nX;
    m_nY = refPt.m_nY;
    return *this;
}
```

1.1.19. Тип возвращаемого оператором присваивания значения.

Нет ограничений на тип возвращаемого любым перегруженным оператором значения, поэтому, пожалуй, в большинстве случаев следует учитывать следующие соображения:

- использование перегруженного оператора ничем не должно отличаться от использования этого оператора для базовых типов;
- предпочитать эффективный вариант.

Рассмотрим три варианта:

1.

```
class Point{
    ...
```

```
    void operator = (const Point& refPt){m_x = refPt.m_x; m_y
                    = refPt.m_y;}
};
int main()
{
    Point pt1(1,1), pt2(2,2), pt3(3,3);
    pt2 = pt1;  //в данном примере возвращаемое значение не
                требуется, так как в результате
                вызова метода будет требуемым
                образом модифицирован объект слева
                от знака равенства
    pt3= pt2 = pt1;   //а в этом случае компилятор выдаст
                ошибку, которая станет очевидна,
                если написать такое цепочечное
                присваивание в функциональной
                форме:
    pt3.operator=(pt2.operator=(pt1)); //скобки не
                обязательны и расставлены только
                для того, чтобы подчеркнуть
                правоассоциативность оператора
                присваивания. Проблема заключается
                в том, что аргументом для вызова
                pt3.operator= является значение,
                возвращаемое pt2.operator=, а в
                данном варианте реализации наш
                перегруженный оператор ничего не
                возвращает!
}
```

2. Для обеспечения цепочечного присваивания operator= должен возвращать уже модифицированное значение левого операнда, например:

```
class Point{
    …
    Point operator = (const Point& refPt){m_x = refPt.m_x;
                m_y = refPt.m_y; return *this;}
                //при возвращении по значению для
                формирования возвращаемого
                значения компилятор создает копию
                того объекта, для которого был
                вызван метод посредством
```



```
                конструктора копирования ->
                дополнительная память для
                возвращаемого значения и
                дополнительное время на выполнение
                конструктора копирования
};
int main()
{
    Point pt1(1,1), pt2(2,2), pt3(3,3);
    pt3= pt2 = pt1;   //теперь цепочечное присваивание
                выполняется корректно, но не
                эффективно
}
```

3. Эффективнее не заставлять компилятор делать копию того объекта, для которого вызывается operator=, а просто вернуть адрес этого объекта. Если возвращать адрес посредством указателя, цепочечное присваивание имело бы довольно непривычную форму, поэтому адрес логичнее возвращать по ссылке.

Замечание: мы имеем право возвращать адрес, так как сам объект гарантированно существует в вызывающей функции.

```
class Point{
    …
    Point& operator = (const Point& refPt)
    {m_x = refPt.m_x; m_y = refPt.m_y; return *this;} //в
                качестве возвращаемого значения
                компилятор формирует адрес объекта
                слева от знака равенства (адрес
                того объекта, для которого
                вызывается метод)
};
int main()
{
    Point pt1(1,1), pt2(2,2), pt3(3,3);
    pt3= pt2 = pt1;   //теперь цепочечное присваивание
                выполняется корректно и эффективно
}
```

1.1.20. Оператор присваивания и нетривиальные классы

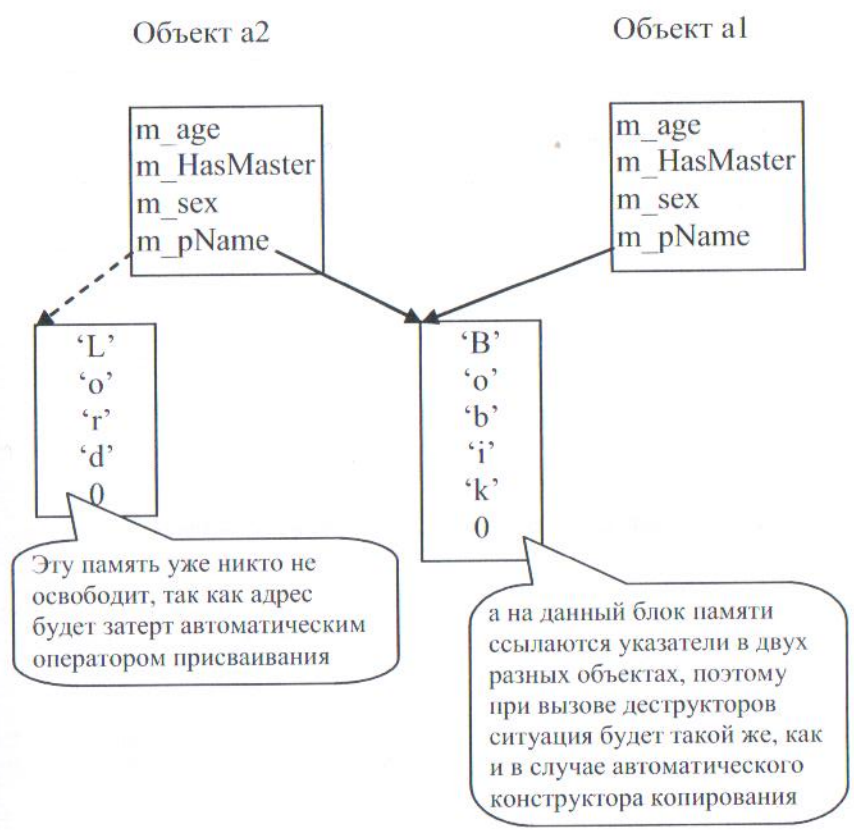
Для простых классов (таких как класс Point) не стоит явно реализовывать оператор присваивания, так как для них прекрасно подходит оператор присваивания, который автоматически умеет генерировать компилятор. Но для чуть более сложных классов (таких как Animal) программист обязан реализовать корректный оператор присваивания, так как использование автоматического приведет к ошибкам времени исполнения:

```
{
      Animal            an1(1,MALE,false,"Bobik"),          an2
                              (5,MALE,true,"Lord");    //создание
                              локальных объектов

      an2 = an1; //вызов автоматического оператора присваивания

}//вызов деструкторов локальных объектов
```

Ситуация напоминает проблему, которая возникала при использовании автоматического конструктора копирования, но усугубляется тем, что оба объекта на момент вызова оператора присваивания существуют, поэтому для каждого объекта была динамически выделена память.



Так как в данном случае нас не устраивает автоматический оператор присваивания, реализуем собственный.

```
Animal& Animal::operator=(const Animal& r)
{
      //Простые данные просто копируем (то же самое сделал бы
                              автоматический          оператор
                              присваивания)

      m_nAge = r.m_nAge;
      m_Sex = r.m_Sex;
```

```
    m_bMaster = r. m_bMaster;


    //А для указателя создаем свою динамическую копию
    delete[] m_pName;          //сначала освобождаем предыдущий
                               блок памяти
    m_pName = new char[strlen(r.m_pName)+1];  //а потом
                               выделяем новый для копии строки
    strcpy(m_pName, r.m_pName);   //и копируем содержимое


    return *this;     //для обеспечения цепочечного
                      присваивания возвращаем по ссылке
                      адрес данного объекта
}
```

Теперь в нижеприведенном фрагменте кода все будет корректно:

```
{
    Animal                          an1(1,MALE,false,"Bobik"),
                      an2(5,MALE,true,"Lord");
    an2 = an1; //вызов перегруженного оператора присваивания
}//вызов деструкторов локальных объектов (для каждого объекта
                      деструктор освобождает свой
                      динамически выделенный блок
                      памяти)
```

А в следующем примере опять возникнет ошибка времени исполнения, так как мы не предусмотрели защиту от ситуации, когда слева и справа от знака равенства находится один и тот же объект:

```
{
    Animal an1(1,MALE,false,"Bobik");
    an1 = an1; //при освобождении блока памяти «приемника» мы
                      одновременно сделаем
                      недействительным блок памяти
                      «источника», так как это один и
                      тот же объект
```



```
}
```

Модифицируем оператор присваивания: если адрес объекта, для которого вызывается метод, совпадает с адресом объекта, полученного в качестве параметра, происходит «присваивание самому себе», поэтому ничего копировать не нужно, а только следует для обеспечения цепочечного присваивания вернуть *this.

```
Animal& Animal::operator=(const Animal& r)
{
    if(this!=&r)       //а) более общий случай, чем (m_pName!=
                               r.m_pName)
                       //б) проверка (*this!=r) требует перегрузки
                               оператора ==
    {
        m_nAge = r.m_nAge;
        m_Sex = r.m_Sex;
        m_bMaster = r. m_bMaster;


        delete[] m_pName;
        m_pName = new char[strlen(r.m_pName)+1];
        strcpy(m_pName, r.m_pName);


    }
    return *this;
}
```

Правило:

- убедитесь, что не происходит присваивания вида x=x
- удалите предыдущие данные
- скопируйте новые (все!) данные
- возвратите *this

1.1.21. Оператор присваивания и наследование

Ситуация с нетривиальными классами усугубляется при наследовании. Если в производном классе оператор присваивания не реализован, то в автоматическом операторе присваивания производного класса компилятор сам вызовет оператор присваивания базового класса. Но, если Вы взяли реализацию оператора присваивания на себя, то компилятор ничего автоматически делать за Вас не будет, то есть:

- (как и в случае конструктора копирования) перегруженный оператор присваивания производного класса работает только со своей (производной) частью объекта;
- если нет явного вызова оператора присваивания базового класса, базовая часть остается прежней!

Поэтому программист должен предусмотреть явный вызов оператора присваивания базового класса:

```
Dog& Dog::operator=(const Dog& r)
{
      if(this!=&r)
      {
            //Способы вызова оператора присваивания базового
                        класса
            Animal::operator=(r);   //функциональный вызов
                        метода operator= базового класса
                        Animal
            *static_cast< Animal* >(this)=r;    //или нормальная
                        форма вызова (посредством явного
                        приведения типа заставляем
                        компилятор интерпретировать
                        указатель производного типа как
                        указатель базового типа)
            static_cast< Animal& >(*this)=r;    //аналогично

            //копирование производной части
            …
      }
      return *this;
```



```
}
```

1.1.22. Оператор присваивания и перегрузка

Если конструктор копирования может быть только один и тип его параметра предопределен, то оператор присваивания может быть перегружен сколько угодно раз.

Замечание: автоматически компилятор генерирует только оператор присваивания вида: A& operator=(const A&).

```
{
      Point pt1(1,1), pt2(2,2);
      pt1 = pt2;  //для такого случая (и если класс простой)
                        компилятор сгенерирует
                        автоматический оператор
                        присваивания сам

      int z = 55;
      pt1 = z;    //или pt1.operator=(z); - а здесь выдаст
                        ошибку, если Вы не перегрузите
                        оператор присваивания таким
                        образом, чтобы он принимал
                        параметр требуемого типа
}
```

Для того, чтобы компилятор «знал», что ему следует делать, встречая такое выражение, введем в класс Point оператор присваивания, который принимает параметр нужного типа:

```
class Point{
      int m_x, m_y;
public:
      …
      Point& operator=(int n){m_x = n; m_y = n;return *this;}
};
```

1.1.23. Оптимизация оператора присваивания

Если речь идет о динамическом выделении памяти, то по возможности следует избегать фрагментации heap-а. В частности такую возможность можно (и нужно) предусмотреть при реализации оператора присваивания. В качестве примера рассмотрим класс, инкапсулирующий одномерный динамический массив:

```
class Array{
      int m_size;        //количество элементов в массиве
      int m_capacity;    //количество элементов, для которых
                         динамически выделена память
                         (емкость)>= m_size
      double* m_p;       //указатель на динамический массив
public:
      Array(int n){  //конструктор, в котором динамически
                     выделяется память
            m_size = 0;
            m_capacity = n;
            m_p = new double[n];
      }
      ~Array(){delete[] m_p;}   //деструктор, в котором
                     динамически выделенная память
                     освобождается
      Array(const Array& other); //конструктор копирования
                     обязательно должен быть явно ( и
                     корректно) реализован для такого
                     класса программистом
      Array& operator=(const Array& other);  //оператор
                     присваивания
      …
};
```

Реализация оператора присваивания.

```
Array& Array::operator=(const Array& other)
{
```



```
      if(this!=&other)
      {
            if(m_capacity<other.m_size)//если выделенной памяти
                        не хватает
            {
                  delete[] m_p;//старый блок освобождаем
                  m_p = new double[other.m_size];//выделяем
                        новый
                  m_capacity = other.m_size;
            }
            m_size=other.m_size;//формируем новый размер
            memcpy(m_p,other.m_p,m_size*sizeof(double));
                        //копируем
      }
      return *this;
}
```

1.1.24. Интересные приемы программирования

Если посмотреть на реализации конструктора копирования и оператора присваивания для нетривиальных классов, то можно увидеть много общего, поэтому некоторые программисты предпочитают реализовывать конструктор копирования посредством оператора присваивания:

```
Array::Array(const Array& other)
{
      m_capacity = 0;   //для того, чтобы в операторе
                        присваивания была выделена новая
                        память
      m_p = 0;    //для того, чтобы оператор delete, который
                  будет вызван в операторе
                  присваивания сработал корректно (с
                  нулевым указателем оператор delete
                  ничего делать не будет, а если
                  оставить случайное значение, то
```

```
                        скорее всего произойдет ошибка
                        времени выполнения)
    *this = other;    //нормальная форма вызова оператора
                        присваивания
}
```

Следующий прием позволяет:

- явно не освобождать предыдущий блок захваченной динамически памяти
- не писать проверки на присваивание вида x=x;

Вводим в класс Array вспомогательный метод:

```
void Array::Swap(Array & r)
{
    //обмениваем значения простых данных посредством
                        вспомогательного временного
                        объекта
    int size = m_size;
    m_size = r.m_size;
    r.m_size = size;

    int capacity = m_capacity;
    m_capacity = r.m_capacity;
    r.m_capacity = capacity;

    //обмениваем значения адресов динамических массивов
    double* p = m_p;
    m_p = r.m_p;
    r.m_p = p;
}
Array & Array::operator=(const Array & r)
{
```



```
    Array tmp-r;//копия параметра
    Swap(tmp);  //обмен данными текущего объекта с копией.
    return *this;
}//для копии будет вызван деструктор, но к этому моменту в
                        копии будут "старые" данные,
                        которые как раз и нужно уничтожить
```

Это красивый прием, но он не позволяет сделать никаких оптимизаций, поэтому лично я им не пользуюсь.

Перегрузка оператора []

Рассмотренный в предыдущем разделе класс Array является оберткой для одномерного динамического массива, поэтому логично предоставить пользователю такого класса возможность обращаться к элементам массива посредством привычного для программиста оператора индексирования:

```
{
    Array a(10);
    for(int i=0; i<10; i++)
    {
        a[i] = i; //нормальная форма вызова оператора
                    индексирования
        a.operator[](i) = i; //функциональная форма вызова
                    оператора индексирования
    }
}
```

Для этого в классе Array должен быть перегружен operator[]:

```
double& Array::operator[](int n) //возвращать следует по ссылке
                    - для того, чтобы можно было
                    использовать возвращаемое значение
                    как справа, так и слева от знака=
                    (запись/чтение)
{
    if(n>-0 && n<m_size) return m_p[n];
    else
    {
```

```
            //генерация исключения
        }
}
```

Замечания:

- перегруженный оператор индексирования должен возвращать не значение требуемого элемента массива, а его адрес, для того чтобы вызов этого метода можно было использовать слева от знака равенства (то есть по возвращенному адресу присвоить новое значение). Указатель в качестве возвращаемого значения использовать неудобно, поэтому принято возвращать ссылку;
- для обычных массивов для повышения эффективности вычислений компилятор не проверяет «выход» значения индекса за пределы массива, но в своем классе в перегруженном операторе индексирования мы можем обезопасить себя от такой ситуации (тема «Обработка исключений»);
- для обычных массивов индекс может быть только целым. Тип параметра перегруженного оператора индексирования может быть любым (пример – в разделе «Встроенные объекты»);
- так как для константных объектов компилятор данный метод вызвать не позволит, обычно в класс вводят еще один перегруженный константный оператор индексирования, а компилятор вызывает тот или иной метод, исходя из константности объекта:

```
double Array::operator[](int i) const    //метод предназначен
                                         только для чтения, поэтому не
                                         имеет смысл возвращать адрес
{
      if(i>=0 && i<m_size) return m_p[i];
      else …
}

int main()
{
            Array a(10);
            int tmp = a[5];   //вызов неконстантного метода
            a[1] = 2;   //вызов неконстантного метода
```



```
            const Array a1(20);
            int tmp1 = a1[5]; // константная версия
//          a1[1] = 2;  //ошибка
}
```

Перегрузка оператора ++ (--)

Специфика операторов инкремента и декремента заключается в том, что этот оператор может быть как постфиксным, так и префиксным, а имя у функции одно и то же - operator++. В ранних версиях C++ не было возможности различить две эти формы, а в современных спецификациях языка разработчики ввели несколько искусственный способ, по которому компилятор может различить префиксную и постфиксную формы, поэтому для каждой формы определяют свою версию перегруженного оператора:

```
class Point{
      int m_x, m_y;
public:
      Point (int x, int y);
      Point& operator++();    //префиксный инкремент
                              (подчиняется правилам: унарный
                              оператор, перегруженный методом
                              класса, не принимает параметров)
      Point operator++(int unused);    //постфиксный -
                              принимает фиктивный параметр
                              (компилятор в качестве
                              передаваемого значения формирует
                              0)
};
```

Реализация префиксного инкремента:

```
Point& Point::operator++()    //возвращается адрес
                              модифицированного объекта
{
      m_ix++;
      m_iy++;
      return *this;
}
```

Реализация постфиксного инкремента:

```
Point Point::operator++(int)  //не имеем права возвращать адрес
                              локального объекта, поэтому
                              возвращать нужно только по
                              значению
{
      return Point(m_ix++, m_iy++);
}
int main()
{
      Point pt1(1,1), pt2(3,3), pt3;
      pt3 = pt1++;        //встречая такую запись, компилятор
                          генерирует вызов функции с
                          фиктивным параметром -
                          эквивалентно pt1.operator++(0);
      pt1.operator++(0);  //функциональная форма вызова
                          постфиксного инкремента
      pt3 = ++pt2;        //pt2.operator++();
}
```

Перегрузка оператора приведения типа

Специфика: нельзя задать тип возвращаемого значения, потому что данный тип представляет собой имя функции

```
      class A{
            char ar[10];
            char c;
            int n;
            public:
            A(char* p, char cc, int nn);
            operator char*(){return ar;}
            operator char(){return c;}
            operator int(){return n;}
```



```
      };

int main()
{
      A a("QWERTYqq",'W',33);
      cout << static_cast<char*>(a) << endl;//operator char*()
      cout << static_cast<char>(a) << endl;//operator char()
      cout << static_cast<int>(a) << endl;//operator int()

//Если по контексту использования компилятор может "догадаться"
                              -
      //какую из форм перегрузки он должен использовать:
      char x1=a;//operator char
      char* x2 = a;//operator char*
        x2 = a.operator char* ();//функциональная форма
      int x3=a;//operator int
//double res = 3*a + a;//ошибка - ambigious operator* -
                       компилятор не знает к какому типу
                       преобразовать a
}
```

Замечание: главная проблема таких функций заключается в том, что они могут неявно вызываться компилятором, когда Вы этого не ожидаете. В таких случаях поведение программы не подчиняется интуитивному анализу, поэтому ее очень трудно отлаживать.

Перегрузка оператора ->. Умные указатели.

Если Вы динамически выделили память, то всегда есть вероятность того, что Вы забудете ее освободить:

```
{
      A* p = new A; //динамический объект
      A a; //локальный стековый
      //работа с обоими
```

```
    //забыли освободить память
}//для локального объекта компилятор вызовет деструктор
```

А динамический объект программист забыл уничтожить, поэтому:

- память «потекла», так как видимость и существование локальной переменной p закончилась, поэтому память будет освобождена ОС только при завершении приложения
- деструктор тоже вызван не будет

Чтобы обезопасить себя от такой ситуации, неплохо было бы «завернуть» такой потенциально опасный указатель в оболочку, которая с гарантией память освободит. Джэфф Элджер называет такие классы-обертки «умными указателями».

Например, есть некоторый класс A, а мне для решения моей задачи нужны объекты такого типа:

```
class A{
    int m_a;
public:
    A(int a) {m_a = a;}
    void func() { … }
};
```

Вводим вспомогательный класс ptr и «заворачиваем» потенциально опасный указатель A* в этот класс-обертку. Специфика:

```
class ptr{
    A* m_p;
public:
    ptr(int a=0){m_p=new A(a);}  // конструктор класса-обертки принимает параметр для целевого класса
    ~ptr(){delete m_p;}          //в деструкторе память освобождаем
    A* operator->(){return m_p;} //объектом класса-обертки позволяем пользоваться таким же образом, как целевым указателем A* посредством перегрузки operator->
```



```
    operator A*(){return m_p;}   //перегрузка оператора приведения типа таким образом, чтобы везде, где нужен указатель A*, можно было использовать объект типа ptr
};
```

Модифицируем целевой класс A таким образом, чтобы пользователю было запрещено самому создавать объекты типа A, а пользоваться этим классом разрешаем только посредством обертки:

```
class A{
    int m_a;
protected://делаем конструктор защищенным, теперь создавать объекты класса A может только метод класса ptr!!!
    A(int a) {m_a = a;}
public:
    void func() { … }
    friend class ptr;        //а классу-обертке предоставляются все права для обращения к защищенным членам класса A
};
void f(A*); //для примера введем еще глобальную функцию, которая принимает указатель A*
int main()
{
    …
    {
        //A a(1);  //ошибка доступа (конструктор защищен)
        ptr my(1);  //в конструкторе ptr динамически создается объект целевого типа
        f(my);//оператор преобразования типа -> на самом деле - f(my->m_p)
        my->func();//operator-> => на самом деле - my.operator->()->func();
```

```
    }//деструктор ~my, в котором память будет освобождена
    …
}
```

Проблема: пока нет никаких препятствий при использовании класса ptr сделать:

```
{
    ptr my(1);
    A* pA = my; //будет вызван оператор приведения типа
    delete pA;
}//а в деструкторе my будет тоже вызван оператор delete, что
                скорее всего приведет к ошибке
                времени выполнения
```

Для предотвращения таких ситуаций можно запретить пользователю вызывать деструктор класса A, сделав его защищенным:

```
class A{
…
protected
    ~A() { } // пусть деструктор ничего не делает, но таким
                образом мы запретим даже
                посредством оператора delete
                вызывать его извне

};
{
    ptr my(1);
    A* pA = my; //будет вызван оператор приведения типа
    //delete pA; //ошибка - нет доступа
}
```

Проблема: так как в классе ptr в качестве члена данных содержится указатель на динамически создаваемый объект, то при использовании автоматического конструктора копирования и оператора присваивания класса ptr в следующем фрагменте ожидаются большие неприятности времени выполнения:

```
    {
```



```
        ptr my1(1);
        ptr my2 = my1;    //конструктор копирования
        my1 = my2;  //оператор присваивания
    }//~my1, ~my2 - ошибка времени выполнения!!!
```

Решения:

а) просто запретить и то, и другое, объявив их защищенными (private или protected) в классе ptr

б) реализовать их корректно!

```
ptr(const ptr& r)
{
    m_p = new A(*(r.m_p));//присваивает указателю адрес
                          своей копии, созданной с помощью
                          конструктора копирования
}

ptr& operator=(const ptr& r)
{
    *m_p = *(r.m_p);
    return *this;
}
```

в) альтернативой является прием, который используется при реализации класса auto_ptr стандартной библиотеки: у объекта всегда один владелец, а при копировании или присваивании объект просто меняет владельца (то есть объект передается от одного объекта к другому как эстафетная палочка)!!!

```
ptr(ptr& r)
{
    m_p = r.m_p;
    r.m_p = 0;
}
```

```
ptr& operator=(ptr& r)
{
      if(this !=&r)
      {
            delete m_p;
            m_p = r.m_p;
            r.m_p = 0;
      }
      return *this;
}
```

Перегрузка оператора (). Функциональные объекты.

Функциональный объект – это класс, в котором перегружен оператор вызов функции - (). Объекты-функции работают почти так же как указатели на функции, но обладают большими возможностями (могут содержать дополнительные данные в своих переменных класса).

Замечание: преимущества использования функциональных объектов начинают проявляться при задании предикатов в обобщенных алгоритмах стандартной библиотеки (везде, где обобщенный алгоритм требует в качестве параметра указатель на функцию, можно использовать функциональный объект).

```
class Point{
      int   m_nX, m_nY;
public:
      …
      void operator()( int dx, int dy )   { m_nX += dx; m_nY +=
                              dy; }
      void operator()(const Point& r)     {         m_nX+=r.m_nX;
                              m_nY+=r.m_nY;}
};

int main()
```



```
{
      Point ar[]={Point(1,2), Point(3,3), Point(4,4) };
      for(int i=0; i<3; i++)
      {
            ar[i](5,6);
            ar[i](Point(2,3));//создается временный объект
                  и     для     ar[i]     вызывается
                  operator()(const Point& r))
      }
}
```

Специфика при использовании перегруженных операторов в классах с конструктором, принимающим один параметр

```
class A{
      int m_a;
public:
      A(int a=0){m_a=a;}
      A operator+(const A& r){return A(m_a+r.m_a);}
};

{
      A a1(1),a2;
      a2 = a1+3;  //в данном выражении компилятор для второго
                  операнда с помощью конструктора с
                  одним параметром неявно приводит
                  операнд типа int к типу A,
                  создавая временный объект, и
                  вызывает перегруженный operator+,
                  как если бы Вы написали a2 =
                  a1+A(3);
```

```
}
```

Замечание: неявное преобразование можно запретить, объявив конструктор с ключевым словом explicit.

Перегрузка оператора с помощью глобальной функции

Если Вы хотите, чтобы компилятор вызывал перегруженный Вами оператор для класса A, глобальная функция должна иметь по крайней мере один параметр типа A. Для удобства (чтобы можно было в такой глобальной функции обращаться к защищенным переменным класса A) такую функцию можно объявить friend-функцией класса A. Но это необязательно!

Перегрузка оператора «==» (проверка на равенство).

Согласно рекомендациям такой оператор предпочтительнее перегружать методом класса, но для примера реализуем перегрузку глобальной функцией.

```
class Animal{
      ...
      friend bool operator == (const Animal&, const Animal&);
                              //объявление глобальной friend-
                              функции перегруженного оператора
                              ==
};
//Реализация глобальной friend-функции перегрузки оператора ==
bool  operator == (const Animal& ref1, const Animal& ref2)
{
      return (    ref1.m_age == ref2.m_age  &&
                  ref1.m_sex == ref2.m_sex &&
                  (strcmp(ref1.m_pName, ref2.m_pName)==0)    );
}
int main()
{
      Animal an1(...), an2(...);
      if( an2 == an1 )  //нормальная форма вызова. Исходя из
                              типа объектов an1 и an2 компилятор
```



```
                              генерирует вызов глобальной
                              функции operator == с параметрами
                              - ссылками на an1 и an2
      cout<<"an1 == an2"<<endl;
      else cout<<"an1 != an2"<<endl;
      if( operator==(an1,an2) == true ) ...;   //то же самое -
                              функциональная форма вызова
      …
}
```

Для производных классов:

```
bool  operator == (const Dog& ref1, const Dog& ref2)
{
      //Сравниваем базовые части посредством уже реализованного
                              оператора== для класса Animal
      if(operator==(static_cast<const          Animal&>(ref1),
                              static_cast<const
                              Animal&>(ref2))==false)
                              //приводить явно второй параметр в
                              VC необязательно, но может
                              зависеть от реализации
      return false;
      //Если базовые части совпадают, сравниваем производные
      return (    ref1.m_bHasMaster == ref2.m_bHasMaster &&
                  (strcmp(ref1.m_pMasterName,
                              ref2.m_pMasterName)==0));
}
int main()
{
      Dog dog1(…), dog2(…);
      if(dog1== dog2)…
}
```

Перегрузка оператора << (вывод в библиотечный ostream)

Хотелось бы выводить содержимое объектов пользовательского типа на консоль (или в файл) также просто и элегантно, как мы до сих пор выводили значения переменных базового типа:

```
{
      Animal a(5,MALE,"Bobik");
      std::cout<<a<<std::endl;      //при этом хотелось бы
                                    увидеть на экране что-нибудь типа:
                                    animal: age=5 sex=male name=Bobik
}
```

Если в предыдущем примере оператор == можно было перегрузить как методом класса (предпочтительнее), так и глобальной функцией, то перегрузка оператора << для вывода в поток это исключение, когда перегрузка возможна только глобальной функцией, так как операнд слева (std::cout) библиотечного типа ostream.

Замечание: в классе ostream стандартной библиотеки перегружен методами класса operator<<, который «умеет» выводить значения базовых типов, а про Ваш пользовательский тип компилятор ничего не знает, и вряд ли стоит модифицировать код стандартной библиотеки, поэтому перегрузка возможна только глобальной функцией.

```
class Animal{
      …
friend  ostream&  operator<<(ostream&  os,  const  Animal&  an);
                              //для  того,  чтобы  иметь
                              возможность  в  этой  глобальной
                              функции  обращаться  к  защищенным
                              переменным  класса  Animal,
                              объявляем ее friend
};

ostream& operator<<(ostream& os, const Animal& an)
{
      os<<"animal: age="<<an.m_age<<
            " sex="<<(an.m_sex==MALE ? "male":"female")<<
```



```
            " name="<<an.m_pName;
      return os;
}
```

Замечание: если хочется выводить в файл информацию в таком же виде, как на экран, то можно использовать один и тот же перегруженный оператор, так как классы ostream и ofstream (для файлового вывода) связаны наследованием:

```
{
      Animal a(5,MALE,"Bobik");
      std::ofstream file ("dat.txt");
      file<<a<<std::endl;
}
```

Попробуйте перегрузить оператор для вывода объекта типа Dog.

Перегрузка перегруженных глобальными функциями операторов

Как и обычную глобальную функцию, перегруженный оператор можно в свою очередь перегрузить сколько угодно раз, Главное, чтобы компилятор при вызове функции смог различить по типу параметров, какую из перегруженных версий вызывать. Замечание: так как количество параметров в данном случае предопределено, то перегрузка возможна только по типу параметров.

```
int main()
{
      A a1(1),a2(2),a3;
//в  двух  следующих  выражениях  предпочтительнее  было  бы
                              перегрузить  оператор  методом
                              класса
      a3 = a1+a2;       //или a3 = operator+(a1,a2);
      a3 = a1+1;        //или a3 = operator+(a1,1);
//а  здесь  перегрузка  возможна  только  глобальной  функцией,  так
                              как  левый  операнд  базового  типа
      a3 = 2+a2;        // или a3 = operator+(2,a2);
}
```

```
class A{
      int m_a;
public:
      A(int n);
//перегрузка оператора + глобальной функцией
      friend A operator+(const A& r1, const A& r2){return
                        A(r1.m_a+r2.m_a);}
      friend A operator+(const A& r1, int n){return
                        A(r1.m_a+n);}
      friend A operator+(int n, const A& r2){return
                        A(n+r2.m_a);}
};
```

Правила выбора формы перегрузки операторов

- виртуальные функции должны быть членами класса
- при использовании библиотечных классов операторы всегда перегружаются глобальными функциями. Если такая функция должна иметь доступ к защищенным данным, объявляйте ее в классе friend
- если аргумент слева базового типа, объявляйте перегруженный оператор глобальной функцией. Если такая функция должна иметь доступ к защищенным данным, объявляйте ее в классе friend
- в остальных случаях рекомендуется перегружать оператор методом класса

Тема V. Сложные указатели. Указатели на члены класса.

В некоторых случаях нужно уметь объявлять и использовать указатели на члены класса (реже на переменные, чаще на методы).

Для демонстрации создадим простенькую иерархию классов:

```
class A{
      int m_Private_a;
public:
      int m_Public_a;   //для примера в нарушение правил ООП
                        объявим общедоступную переменную

      A(int a1=0, int a2=0){m_Private_a=a1; m_Public_a =a2;}
      void f();
      virtual void vf();
};

class B:public A{
      …
public:
      virtual void vf();
};

int main()
{
      // указатели на переменные класса
      A a(1,2);
      //a)
      //int* p = &A::m_Private_a;//ошибка доступа - cannot
                        access private member
```