Канонические формы арифметических операторов и операторов присваивания

- Если можно записать a+b, то необходимо, чтобы можно было записать и a+=b. В общем случае для некоторого бинарного оператора @ (+, -, * и т.д.) должна также быть определена его присваивающая версия, так чтобы a@=b и a=a@b имели один и тот же смысл (причем первая версия может быть более эффективна).

Канонический способ:

```
T& T::operator@=( const T& ) {
  // ... реализация ...
  return *this;
}
T operator@( const T& lhs, const T& rhs ) {
   T temp( lhs );
   return temp @= rhs;
}
```

- *Пример. Реализация += для строк*.

  При конкатенации строк полезно заранее знать длину, чтобы выделять память только один раз:

```
String& String::operator+=( const String& rhs ) {
  // ... Реализация ...
  return *this;
}
String operator+( const String& lhs, const String& rhs ) {
  String temp;    // изначально пуста
  temp.Reserve(lhs.size() + rhs.size());
      // выделение достаточного количества памяти
  return (temp += lhs) += rhs;    // Конкатенация строк и возврат
}
```

- *Исключения*

  В некоторых случаях (например, оператор operator*= для комплексных чисел), оператор может изменять левый аргумент настолько существенно, что более выгодным может оказаться реализация оператора operator*= посредством оператора operator*, а не наоборот.

Канонический вид ++ и --

```
// --- Префиксные операторы

T& T::operator++()   // Префиксный вид:
{
// Выполнение  инкремента
return *this;
}

 T& T::operator--()    // Префиксный
     вид:
{
// Выполнение   декремента
return *this;
}
```

```
// --- Постфиксные операторы

T T::operator++(int)   // Постфиксный вид:
{
T old( *this ); // Запомним старое
     значение
++*this;   // Вызов префиксной версии
return old;   // Возврат старого значения
}
T T::operator--(int)   // Постфиксный вид:
{
T old( *this ); // Запомним старое
     значение
--*this;   // Вызов префиксной версии
return old;   // Возврат старого значения
}
```

Или **const** T T::operator++(int) , чтобы избежать конструкций вида t++++;

Канонический вид присваивания

- Предпочтительно объявлять копирующее присваивание для типа T с одной из следующих сигнатур

T& operator=(const T&); // Классический вид

T& operator=(T) // Потенциально оптимизированный вид (если в любом случае требуется копия аргумента в теле оператора)

- Избегайте делать любой оператор присваивания виртуальным. При необходимости виртуального поведения лучше использовать виртуальную именованную функцию, например,

virtual void Assign(const T&);

- Не возвращайте const T&. Хотя этот тип возвращаемого значения имеет то преимущество, что защищает от странных присваиваний наподобие (a=b)=c, главным его недостатком является то, что вы не сможете поместить объекты типа T в контейнеры стандартной библиотеки; эти контейнеры требуют, чтобы оператор присваивания возвращал тип T&.

```
class Base {
  public:
     Base(int initialValue = 0) : x(initialValue) {}
  private:
     int x;
} ;
class Derived: public Base {
public:
     Derived(int initialValue) : Base(initialValue),
       y(initialValue) {}
     Derived& operator=(const Derived^ rhs) ;
private:
     int y;
} ;
Derived& Derived::operator=(const Derived& d){
     if (this == &d) return *this;
     Base::operator=(d);
       // Вызов this->Base::operator=.
     y = d.y;
     return *this;
     }
```

- Явно вызывайте все операторы присваивания базовых классов и всех данных-членов
- Возвращайте из оператора присваивания значение *this
- Убедитесь, что ваш оператор присваивания безопасен в смысле присваивания самому себе (простейший способ – проверка на равенство указателей)

Перегрузка << и >> для классов

```
ostream& opeartor << (ostream& os, const Matrix& M){
    os<<m.size<<endl;
    for (int i=0; i<=m.size; ++i)
    {
      for (int j=0; j<=m.size; ++j)
        os <<m.data[i,j]<<“-”;
    } os<<endl;
    return os;  // чтобы можно было записать cout<<m1<<m2;
}
```

- Ввод-вывод не может быть методом класса, так как первым аргументом должен быть поток

```
Matrix M;
cout<<M; // нормально
M.cout<<M;  или M<<cout; // необычный синтаксис
```

Используйте перегрузку, чтобы избежать неявного преобразования типов

- Пример: сравнение строк

```
class String {
  // ...
  String( const char* text );  // Обеспечивает неявное преобразование типов
};
bool operator==( const String&, const String& );
// ... Где-то в коде ...
if( someString == "Hello" ) { ... }
```

приведенное сравнение таким образом, как если бы оно было записано в виде someString == String("Hello")

- Решение (дублирование сигнатур):

```
bool operator==( const String& lhs, const String& rhs ); // #1
bool operator==( const String& lhs, const char* rhs );    // #2
bool operator==( const char* lhs, const String& rhs );    // #3
```

Избегайте возможностей неявного преобразования типов

Две основные проблемы:

- Они могут проявиться в самых неожиданных местах.
- Они не всегда хорошо согласуются с остальными частями языка программирования.

Пример 1. Перегрузка. Пусть у нас есть, например, Widget::Widget(unsigned int), который может быть вызван неявно, и функция Display, перегруженная для Widget и double. Рассмотрим следующий сюрприз при разрешении перегрузки:

```
void Display(double); // Вывод double
void Display(const Widget&); // Вывод Widget
Display(5); // Гм! Создание и вывод Widget
```

Пример 2. Работающие ошибки.

```
class String {
  // ...
  public:    operator const char*();  // Сомнительное решение...
};
```

Пусть s1 и s2 — объекты типа String. Все приведенные ниже строки компилируются:

```
int x = s1 - s2;              // Неопределенное поведение
const char* p = s1 - 5; // Неопределенное поведение
p = s1 + '0';                 // Делает не то, что вы ожидаете
if( s1 == "0" ) { ... }     // Делает не то, что вы ожидаете
```

- Решение:

1. *По умолчанию используйте explicit в конструкторах с одним аргументом*

```
class Widget {
  // ...
  explicit Widget(unsigned int widgetizationFactor);
  explicit Widget(const char* name, const Widget* other = 0);
};
```

2. *Используйте для преобразований типов именованные функции, а не соответствующие операторы:*

```
class String {
  // ...
  const char* as_char_pointer() const;  // В традициях c_str
};
```

Не перегружайте без крайней необходимости &&, || и , (запятую)

```
Employee* e = TryToGetEmployee();
if( e && e->Manager() )
// ...
```

Корректность этого кода обусловлена тем, что e->Manager() не будет вычисляться, если e имеет нулевое значение. Если же используется перегруженный оператор operator&&, то код потенциально может вызвать e->Manager() при нулевом значении e

```
int i = 0;
f( i++ ), g( i );
//…
```

Если используется пользовательский оператор-запятая, то неизвестно, получит ли функция g аргумент 0 или 1

Сохраняйте естественную семантику перегруженных операторов!

- Исключение — высокоспециализированные библиотеки (например, генераторы синтаксических анализаторов, шаблоны для научных вычислений)

Отношения в C++ (по силе взаимосвязи)

1. Дружба
2. Наследование
3. “Владение” (композиция, агрегация)

Должна ли функция быть реализована как метод или друг класса?

- *Если нет выбора - делайте функцию методом класса.* В частности, если функция представляет собой один из операторов =, ->, [] или ().
- *Если* 1. функция требует левый аргумент иного типа (как, например, в случае операторов >> или <<) или 2. требует преобразования типов для левого аргумента, *то сделайте её внешней (и при необходимости другом)*.
- *Если функция может быть реализована с использованием только открытого интерфейса класса, то делайте внешнюю функцию* (не-друг и не метод).
- *Если функция требует виртуального поведения*, то добавьте виртуальный метод для обеспечения виртуального поведения, и реализуйте внешнюю функцию с использованием этого метода.

Виртуализация функций, не являющихся методами класса

```
class NLComponent{
  public:
    virtual ostream& print(ostream&s) const = 0;
   …
};
class TextBlock: public NLComponent{
  public:
    virtual ostream& print(ostream&s) const;
   …
};
class Graphic: public NLComponent{
  public:
    virtual ostream& print(ostream&s) const;
   …
};
inline // чтобы избежать расходов на вызов функции
   ostream& operator<< (ostream& s, const NLComponent c) { return c.print(s); }
```

- Мульти-методы (функции, являющиеся виртуальными к более чем одному объекту) языком C++ в явном виде не поддерживаются.

Открытое наследование как моделирования отношения «является» (“is a”)

- Каждый объект типа D(derived) является также объектом типа B(base), но *не наоборот.*
- B представляет собой более общую концепцию, чем D, а D – более конкретную концепцию, чем B.
- Везде, где может быть использован объект B, можно использовать также объект D, потому что D является объектом типа B.

```
class Person {...};
class Student: public Person {...};

void eat(const Person& p); // все люди могут есть
void study(const Student& s); // только студент учится
Person p; // p – человек
Student s; // s – студент
eat(p); // правильно, p есть человек
eat(s); // правильно, s – это студент, и студент также является человеком
study(s); // правильно
study(p); // ошибка! p – не студент
```

```
class Bird {
public:
   virtual void fly(); // птицы умеют летать
...
};
class Penguin: public Bird { // пингвины – птицы
... };
```
Затруднение: утверждается, что пингвины могут летать

```
class Bird {
... // функция fly не объявлена
};
class FlyingBird: public Bird {
public:
  virtual void fly();
...};
class Penguin: public Bird {
... // функция fly не объявлена
};
```
Данная иерархия гораздо точнее отражает реальность, чем первоначальная.

Другой вариант – лучше ли?

```
void error(const std::string& msg); // определено в другом месте
class Penguin: public Bird {
   public:
      virtual void fly() {error("Попытка заставить пингвина летать!");}
   ...
};
```

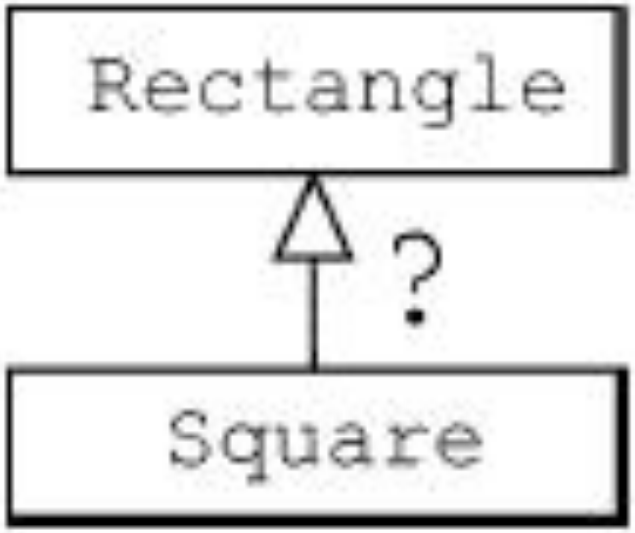
Не говорим: «Пингвины не могут летать», а лишь сообщаем: «Пингвины могут летать, но с их стороны было бы ошибкой это делать».

В чем разница? Утверждение «пингвины не могут летать» может быть поддержано на уровне компилятора, а соответствие утверждения «попытка полета ошибочна для пингвинов» реальному положению дел может быть обнаружено во время выполнения программы.

Чтобы обозначить ограничение «пингвины не могут летать – и точка», следует убедиться, что для объектов Penguin функция fly() не определена. Если теперь попробовать заставить пингвина взлететь, компилятор сделает выговор за нарушение правил:

```
Penguin p;
p.fly(); // ошибка!
```

Должен ли класс Square (квадрат) открыто наследовать классу Rectangle (прямоугольник)?



```
class Rectangle {
public:
    virtual void setHeight(int newHeight);
    virtual void setWidth(int newWidth);
    virtual int height() const; // возвращают текущие значения
    virtual int width() const;
    ...
};
void makeBigger(Rectangle& r) // увеличивает площадь r
{
    int oldHeight = r.height();
    r.setWidth(r.width() + 10); // увеличить ширину r на 10
    assert(r.height() == oldHeight);
      // убедиться, что высота r  не изменилась
}
```

Функция make-Bigger изменяет только ширину r. Высота остается постоянной.

```
class Square: public Rectangle {…};

Square s;
 ...
assert(s.width() == s.height()); // должно быть справедливо для всех квадратов
makeBigger(s); // из-за наследования s является Rectangle, поэтому мы можем
                // увеличить его площадь
assert(s.width() == s.height());
    // По-прежнему должно быть справедливо для всех квадратов
```

Проблема:

- Перед вызовом makeBigger высота s равна ширине.
- Внутри makeBigger ширина s изменяется, а высота – нет.
- После возврата из makeBigger высота s снова равна ширине (отметим, что s передается по ссылке, поэтому makeBigger модифицирует именно s, а не его копию).

Принцип подстановки Барбары Лисков (*Liskov Substitution Principle, LSP*)

- Пусть q(x) является свойством, верным относительно объектов x некоторого типа T. Тогда q(y) также должно быть верным для объектов y типа S , где S является подтипом типа T.

Другая формулировка:

- Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Отношение "является" ("работает как")

- **Открытое наследование означает заменимость. Наследовать надо не для повторного использования, а чтобы быть повторно использованным**
- Открытое наследование позволяет указателю или ссылке на базовый класс в действительности обращаться к объекту некоторого производного класса без изменения существующего кода и нарушения его корректности
- До ООП было легко решить вопрос вызова старого кода новым. Открытое наследование упростило прозрачный и безопасный вызов нового кода старым
- Исключения (классы стратегий)