Программирование и алгоритмизация

Санкт-Петербургский государственный политехнический университет

17 марта 2015

friend

Ключевое слово предоставляющее возможность обращения к защищенным понятиям класса извне. Может быть полезно при перезгрузке операторов и реализации вспомогательных классов.

Без friend

```
class Rect
{
  int m_left, m_top, m_right, m_bottom;
};

Rect BoundingRect(const Rect &r1, const Rect &r2)
{
  int l = (r1.m_left < r2.m_left) ? r1.m_left : r2.
      m_left; //???
  ...
}
```

C friend

Секция расположения ключевого слова friend не имеет значения, хотя часто для ясности располагают прототип функции в секции public.

```
class Rect
{
  int m_left, m_top, m_right, m_bottom;
  friend Rect BoundingRect(const Rect &r1, const
      Rect &r2);
};


Rect BoundingRect(const Rect &r1, const Rect &r2);
{
  int l = (r1.m_left < r2.m_left) ? r1.m_left : r2.
      m_left; //???
  ...
}
```

C friend

Для всех остальных функций правила доступа к полям остаются прежними.
Дружба не передается по наследству.

```
int main()
{
   Rect r;
   r.m_left = 1; //???
}
```

Дружба при наследовании

Друг производного класса имеет доступ к protected членам базового класса.

```
class A { int m_a1;
protected: int m_a2;
};
class B { int m_b;
  friend void F(B &b);
};
void F(B& b)
{
  std::cout << b.m_a1; //??
  std::cout << b.m_a2; //??
  std::cout << b.m_b; //??
}
```

Дружба между классами

```
class Circle { int r;
public:
    Circle(const Rect&r) {
int w = r.m_right - r.m_left; //???
    }
};
```

friend class

```
class Rect {
      friend class Circle;
      ...
};
Circle::Circle(const Rect &r)
{
     int w = r.m_right - r.m_left;
     int h = r.m_bottom - r.m_top;
}
```

Дружба классов не наследуется.

Дружба классов не является транзитивной.

```
class X { friend class Y; };
class Y { friend class Z; };
```

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

friend метод другого класса

```
class Rect {
      friend Circle::Circle(const Rect&);
      ...
};
Circle::Circle(const Rect &r)
{
     int w = r.m_right - r.m_left; //???
     int h = r.m_bottom - r.m_top; //???
}
```

Перегрузка операторов

Предоставление возможности обращаться с объектами пользовательского типа также как с переменными базового типа.

```
int x, y;
x + y;
double x1, y1;
x1 + y1;
```

//выполняются разные низкоуровневые операторы для разных типов: add, fadd.

Пользовательские типы

Для класса A программист должен реализовать функцию с именем operator+.

```
A a1(1), a2(2), a3;
a3 = a1 + a2; //???
```

Специфика перегружаемых операторов

- Нельзя создавать собственные операторы, а можно перегружать только существующие;
- перегруженный оператор действует только по отношению к объектам того класса, для которого он переопределен;
- нельзя менять число операндов оператора;
- перегруженные операторы наследуют приоритеты и ассоциативность от встроенных операторов;
- оператор перегружается только относительно пользовательского типа данных;
- нельзя перегружать операторы: . :: .* ? :
- как любая другая функция оператор может быть перегружен несколько раз, быть вирульаным или чисто виртуальным;
- не существует ограничений на тип возвращаемого значения.

Способы перегрузки операторов

Унарные операторы:

- метод класса без параметров;
- глобальная функция с одним параметром.

Бинарные операторы:

- метод класса с одним параметром;
- глобальная функция с двумя параметрами.

Пример вызова operator+

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

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

Рекомендации при перегрузке

Всегда стоит предпочитать перегрузку методом класса, за исключением:

- первый операнд относится к базовому типу, например, z = 1 + z;
- тип первого операнда библиотечный;
- операторы =, (), [], -> могут быть перегружены только методом класса.

Порядок поиска компилятором перегруженного бинарного оператора

1. если оба аргумента относятся к базовым типам, то используется встроенный оператор;
2. если слева стоит операнд пользовательского типа, то компилятор ищет оператор в форме метода класса;
3. если перегрузки в форме метода не найдено или слева стоит операнд базового типа, то компилятор ищет перегрузку в форме глобальной функции;
4. ошибка при компиляции.

operator=

Автоматический оператор присваивания генерируется компилятором автоматически.

- поэлементно копирует данные из одного объекта в другой;
- вызывает оператор присваивания базового класса (определенный программистом явно или автоматически);
- вызывает оператор присваивания для встроенных объектов.

Компилятор не генерирует автоматически оператор присваивания в следующих случаях:

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

operator=

Тип возвращаемого оператором присваивания значения.

```
class A {
        int m_a;
public: ...
void operator = (const A & other) { m_a = other.m_a
    ; }
int main()
{
    A a1(1), a2(2), a3;
    a2 = a1; //???
    a3 = a2 = a1; //??
    a3.operator=(a2.operator=(a1));
}
```

Т.к. нет ограничений, то следует учитывать преемственность от базовых типов и предпочитать эффективный вариант.

operator= возвращаемое значение

```
class A {
        int m_a;
public: ...
A operator = (const A & other) { m_a = other.m_a;
    return *this; //??? }
int main()
{
    A a1(1), a2(2), a3;
    a2 = a1; //???
    a3 = a2 = a1; //??
    a3.operator=(a2.operator=(a1));
}
```

operator= возвращаемое значение

```
class A {
        int m_a;
public: ...
A& operator = (const A & other) { m_a = other.m_a;
    return *this; //??? }
int main()
{
    A a1(1), a2(2), a3;
    a2 = a1; //???
    a3 = a2 = a1; //??
    a3.operator=(a2.operator=(a1));
}
```

Присваивание и классы с указателями

```
class Animal {
    int m_age;
    char *m_name;
    public: Animal(int, const char*) { ... };
};
int main()
{
    Animal a1(10, "Bobik");
    Animal a2(15, "Tuzik");
    a1 = a2;
}
```

"Правильный"оператор присваивания

```
Animal &Animal::operator=(const Animal &a)
{
    m_age = a.m_age;

    delete[] m_name;
    m_name = new char[strlen(a.m_name)+1];
    strcpy(m_name, a.m_name);
    return *this;
}
```

Нетривиальный класс и оператор присваивания

```
int main
{
    Animal a1(10, "Bobik");
    Animal a2(15, "Tuzik");
    a1 = a2; //???

    a1 = a1; //???
}
```

Защита от самоприсваивания

```
Animal &Animal::operator=(const Animal &a)
{
    if(&a == this)
        return *this;

    m_age = a.m_age;
    delete[] m_name;
    m_name = new char[strlen(a.m_name)+1];
    strcpy(m_name, a.m_name);
    return *this;
}
```

Оператор присваивания и наследование

- оператор присваивания не наследуется;
- если нет явного вызова оператора присваивания базовой части класса, то базовая часть останется прежней;
- может быть virtual;

Оператор присваивания и наследование

```
Dog & Dog::operator=(const Dog &r)
{
        if(this != r)
        {
            //копирование базовой части
            Animal::operator=(r);
            *static_cast<Animal*>(this)=r;
            static_cast<Animal&>(*this)=r;

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

Конструктор копирования через присваивание

```
MyString::MyString(const MyString &r)
{
        m_pStr = 0;
        *this = r;
}
```

move operator=

```
int main()
{
        MyString s1("My␣My");
        s1 = MyString("StrStr"); //???
}
```

move operator=

```
MyString &MyString::operator=(MyString &&other)
{
delete[] m_pStr;
m_pStr = other.m_pStr;
other.m_pStr = 0;
return *this;
}
int main()
{
        MyString s1("My My");
        s1 = std::move(MyString("StrStr"));
}
```