ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Перегрузка операторов

Никитин Михаил Евгеньевич, 2015

Использование const

- Что такое const и зачем его использовать?
- Есть две точки зрения на использование const:
  - Первая: const - это плохо.
  - Вторая: const - это хорошо.
- Чем отличаются?

```
int *const p1;
int const* p2;
const int* p3;
```

Константы и методы

- Для методов допустимо использование const, применительно к this.

```
class A {
private:
        int x;
    public:
        void f(int a) const{
            x = a; // <-- не работает
        }
};
```

Перегрузка операторов

- Рассмотрим бинарный оператор @, если ***x*** имеет тип ***X***, а ***y*** имеет тип ***Y***, правила разрешения выражения ***x@y*** применяются следующим образом:
  - если ***X*** является классом, выяснить, определяется ли ***operator@*** в качестве члена класса X, либо базового класса ***X***;
  - если ***X*** определен в пространстве имен ***N***, поискать объявления ***operator@*** в ***N;***
  - если ***У*** определен в пространстве имен ***M***, поискать объявления ***operator@*** в ***M.***

Операторы-члены и не-члены

- Рекомендуется сводить к минимуму количество функций, непосредственно манипулирующих представлением объекта.
- Старайтесь определять в теле самого класса только те операторы, которые должны модифицировать значение первого аргумента, например оператор +=.
- Операторы, которые просто выдают новое значение на основе своих аргументов, такие как +, рекомендуется определять вне класса

Операторы-члены и не-члены


```
class complex {
        double re, im;
    public:
        complex(double r, double i);
        complex& operator+=(complex a); // требует доступа к представлению
};
complex operator+(complex a, complex b)
{
    complex r = a;
    return r += b; // доступ к представлению при помощи +=
}
inline complex& complex::operator+=(complex a)
{
    re += a.re;
    im += a.im;
    return *this;
}
```

Операторы-члены и не-члены


```
void f(complex x, complex y, com
{
    complex r1 = x + y + z;
    complex r2 = x;
    r2 += y,
    r2 += z;
    complex d = 2 + b;
}
```

```
// r1 = operator+(operator+(x,y), z)
// r2 = x
// r2.operator+= (y)
// r2.operator+=(z)
// Ошибка!!! Нет operator+(2,b)
```

Скомпилируется программа?

Все ли хорошо работает?

Операторы-члены и не-члены


```
complex& complex::operator+=(double a){
     re += a;
     return *this;
}
complex operator+(complex a, complex b){
     complex r = a;
     return r+= b; // вызывает complex: :operator+= (complex)
}
complex operator+(complex a, double b){
     complex r = a;
     return r += b; // вызывает complex::operator+=(double)
}
complex operator+ (double a, complex b){
     complex r = b;
     return r += a; // вызывает complex::operator+= (double)
}
```

Операторы-члены и не-члены


```
void f(complex x, complex y)
{
    complex r1 = x + y;
    complex r2 = x + 2;
    complex r3 = 2 + x;
}
```

Конструктор инициализатор


```
class complex {
    double re, im;
    public:
    complex(double r, double i);
};

class test_class {
    const int n;
    complex e;
    complex& t;
    public:
    test_class();
};
```

У класса complex есть конструктор по умолчанию?

В чем проблема(ы) класса test_class?

Как инициализировать значения полей?

Конструктор инициализатор


```
class test_class {
    const int n;
    complex e;
    complex& t;
public:
    test_class(int _n=0, complex _e=complex(0,0), complex&
_t=complex(0,0));
};

test_class::test_class(int _n,complex _e,complex& _t):n(_n),e(_e),t(_t){}
```

Инициализация

- Для инициализации и присваивания комплексным переменным скалярных значений, нам нужно преобразование скалярной величины

```
complex b = 3;
```

```
class complex {
    double re, im;
    public:
    complex(double r) : re(r), im(0) {}
    // ...
};
```

Конструкторы и преобразования


```
complex(double r) : re(r), im(0) {}
```

- Конструктор является предписанием для создания значения данного типа из типа double.
- Конструктор используется, когда ожидается значение данного типа, и когда такое значение может быть создано конструктором из значения, заданного как инициализатор, или присваиваемого значения. Поэтому, конструктор, имеющий один аргумент, не нужно вызывать явно.
- Например,

```
complex b = 3;
complex b = complex(3);
```

Конструкторы и преобразования

- Мы определили три варианта каждой из четырех стандартных арифметических операций:

```
complex operator+ (complex, complex);
complex operator+ (complex, double);
complex operator+ (double, complex);
```

Конструкторы и преобразования

- Альтернативным способом реализации различных версий функции для каждой комбинации аргументов является использование преобразований типов.

```
class complex {
     double re, im;
     public:
     complex(double r) : re(r), im(0) {}
     // ...
};
complex operator+(complex a, complex b);
int main()
{
     complex a = complex(1, 1);
     complex b = 2+a;
     return 0;
}
```

Друзья класса

- Обычное объявление функции-члена гарантирует три логически разные вещи:
  1. функция имеет право доступа к закрытой части объявления класса;
  2. функция находится в области видимости класса;
  3. функцию должна вызываться для объекта класса (имеется указатель this).

Друзья класса

- Объявив функцию-член как ***static*** какие свойства мы ей придаем?
- Объявив функцию как ***friend,*** мы наделяем ее только первым свойством.
- **Дружественная функция** — это функция, которая не является членом класса, но имеет доступ к членам класса, объявленным в полях private или protected.

Друзья класса

- **Дружественные функции**
  - Дружественная функция объявляется внутри класса с модификатором **friend**
  - Дружественные функции не являются членами класса, поэтому им не передается указатель **this**
  - Объявление функций-друзей friend можно поместить и в закрытой и в открытой частях объявления класса — не имеет значения, где именно.
- Правило использования:
  - Если нет важных доводов использовать дружественные функции – используйте вместо них члены класса
  - Если важные доводы есть – подумайте, а действительно ли они так важны

Друзья класса


```
class Matrix;
class Vector {
   float v[4];
   friend Vector operator*(const Matrix&, const Vector&);
};
class Matrix {
   float v[4][4];
   friend Vector operator*(const Matrix&, const Vector&);
};
Vector operator*(const Matrix& matr, const Vector& vec){
   Vector r;
   for (int i = 0; i < 4; i++) {
      r.v[i] = 0;
      for (int j = 0; j < 4; j++)
     r.v[i] += matr.v[i][j] * vec.v[j];
   }
   return r;
   }
```

Дополнительные функции-члены


```
class complex {
    double re, im;
  public:
    complex(double r = 0, double i = 0) :re(r), im(i) {};
    complex& operator+= (complex);
    complex& operator+= (double);
    complex operator* (complex);
    double real() const { return re;}
    double imag() const { return im;}
};

inline bool operator== (complex a, complex b){
  return (a.real() == b.real()) && (a.imag() == b.imag());
}
```

Индексация


```
int main()
{
    string buf;
    Assoc_array vec;
    while (cin >> buf) vec[buf]++;
    vec.print_all();
    return 0;
}
```

Индексация

- Чтобы придать смысл индексам для объектов класса можно воспользоваться функцией ***operator***[].
- Второй аргумент (индекс) функции ***operator")***] может быть любого типа.
- Это делает возможным определение векторов, ассоциативных массивов и т. д.

Индексация


```
#include <iostream>
#include <string>
using namespace std;
struct Node{
   string key;
   double val;
   Node* next;
};
typedef Node* PNode;
class Assoc_array{
    PNode Head=NULL;
    PNode add(string key);
public:
    Assoc_array() {}
    double& operator[](const string&);
    void print_all() const;
};
```

Индексация


```
PNode Assoc_array::add(string key) {
    PNode q = Head;
    while (q && (q->key != key)) q = q->next;
    if (!q)
    {
       PNode tmp = new Node;
       tmp->key = key;
       tmp->val = 0;
       tmp->next = Head;
       Head = tmp;
       return tmp;
    }
    else return q;
}
```

Индексация


```
double& Assoc_array::operator[](const string& key)
{
     PNode q = Head;
     while (q && (q->key != key)) q = q->next;
     if (q) return q->val;
     else return add(key)->val;
}

void Assoc_array::print_all() const
{
     PNode q = Head;
     while (q)
     {
        cout << "(" << q->key << "," << q->val << ")" << endl;
        q = q->next;
     }
}
```

Индексация


```
int main()
{
    string buf;
    Assoc_array vec;
    while (cin >> buf) vec[buf]++;
    vec.print_all();
    return 0;
}
```

Пост- и пре- фиксные операторы?

- По сравнению с другими операторами у этих есть особенность - вы можете использовать их как в префиксной (++i, --i), так и в постфиксной (i++, i--) нотации.
- Для того чтобы отличить постфиксную реализацию перегрузки от префиксной использует не используемый аргумент типа int.

Пост- и пре- фиксные операторы?


```
class Point{
    int _x, _y;
public:
    Point& operator++();       // Prefix increment operator.
    Point operator++(int);     // Postfix increment operator.
    Point& operator--();       // Prefix decrement operator.
    Point operator--(int);     // Postfix decrement operator.
    Point() { _x = _y = 0; }
    int x() { return _x; } // Define accessor functions.
    int y() { return _y; } // Define accessor functions.
};
```

Пост- и пре- фиксные операторы?


```
Point& Point::operator++() // Define prefix increment operator.
{
    _x++;
    _y++;
    return *this;
}
Point Point::operator++(int) // Define postfix increment operator.
{
    Point temp = *this;
    ++*this;
    return temp;
}
```

```
Point& Point::operator--() // Define prefix decrement operator.
{
    _x--;
    _y--;
    return *this;
}

Point Point::operator--(int) // Define postfix decrement operator.
{
    Point temp = *this;
    --*this;
    return temp;
}
```