Совместное использование. Перегрузка операций

```
double abs(double x);
int    abs(int x);

abs(1);
abs(1.0);
```

```
class Buffer
 { private:
    char *p;
    int size;
   protected:
    Buffer(int s, char *np) {size = s; p = np; }
   public:
    Buffer(int s) {p = new char[size = s]; }
   ...
  };
```

Процесс поиска подходящей функции из множества перегруженных заключается в нахождении наилучшего соответствия типов формальных и фактических аргументов. Это осуществляется путем проверки набора критериев в следующем порядке:

- точное соответствие типов, т.е. полное соответствие или соответствие, достигаемое тривиальными преобразованиями типов (например, имя массива и указатель, имя функции и указатель на функцию, типы *T* и *const T*);
- соответствие, достигаемое «продвижением» интегральных типов (например, *bool* в *int*, *char* в *int*, *short* в *int*) и *float* в *double*;
- соответствие, достигаемое путем стандартных преобразований (например, *int* в *double*, *double* в *int*, *double* в *long double*, указателей на производные типы в указатели на базовые, указателей на произвольные типы в *void**, *int* в *unsigned int*);
- соответствие, достигаемое при помощи преобразований, определяемых пользователем;
- соответствие за счет многоточий в объявлении функции.

```
void f(int);

void g()
 { void f(double);
   f(1);  // Вызов f(double)
  }
```

x + y * z

умножить y на z
и прибавить результат к x

```
operator <операция> (<операнды>)
```

.

.*

::

? :

sizeof

```
class Complex
 { private:
    double r, m;
   public:
    Complex(double nr = 0, double nm = 0) :
        r(nr), m(nm) { }
    Complex operator ++ ();
    Complex operator ++ (int);
    Complex operator +  (const Complex& c) const;
    Complex operator += (const Complex& c);
    bool    operator == (const Complex& c) const;
  };
```

```
Complex Complex::operator ++ ()
 { ++r; return *this; }

Complex Complex::operator ++ (int)
 { Complex x = *this;
   r++;
   return x;
  }

Complex Complex::operator + (const Complex& c) const
 { return Complex(r + c.r, m + c.m); }

Complex Complex::operator +=(const Complex& c)
 { r += c.r; m += c.m; return *this; }

bool    Complex::operator == (const Complex& c) const
 { return r == c.r && m == c.m; }
```

```
void main()
 { Complex a(0, 0), b(2, -2), c;

   ++a;          // a.operator++()
   a++;          // a.operator++(0)
   c = a + b;      // a.operator+(b)
   a += b;
   c = a + 2.5;
   c = a + ++b;
  }
```

```
class Complex
 { private:
    double r, m;
   public:
    Complex(double nr = 0, double nm = 0);
    Complex operator += (const Complex& c);
  };

Complex Complex::operator += (const Complex& c)
 { r += c.r;
   m += c.m;
   return *this;
  }
```

```
Complex operator + (const Complex& c1, const Complex& c2)
 { Complex x = c1;
   return x += c2;
  }

void main()
 { Complex a(0, 0), b(2, 2), c(7, -5);

   Complex r1 = a + b + c;  // r1 = operator+(a, operator+(b, c))
   Complex r2 = a;          // r2 = a
   r2 += b;                 // r2.operator+=(b)
   r2 += c;                 // r2.operator+=(c)
  }
```

```
class Vector
 { private:
    int    size;
    double *v;
   public:
    explicit Vector(int n = 0);
    Vector(const Vector& vector);
    ~Vector();   // Деструктор
    int GetSize() const { return size; }
    int SetSize(int n);
    Vector  operator =  (const Vector& vector);
    double& operator [] (int n);
    Vector  operator -  () const;
```

```
  int     operator == (const Vector& vector) const;
  int     operator != (const Vector& vector) const;
  Vector  operator +  (const Vector& vector) const;
  Vector  operator += (const Vector& vector);
  Vector  operator -  (const Vector& vector) const;
  Vector  operator -= (const Vector& vector);
  Vector  operator +  (double value) const;
  Vector  operator += (double value);
  Vector  operator -  (double value) const;
  Vector  operator -= (double value);
  double  operator *  (const Vector& vector) const;
};
```

```
Vector::Vector(int n)
 { if (n < 0) n = 0;
   size = n;
   v    = NULL;
   if (size)
    if ((v = (double *)malloc(size * sizeof(double))) == NULL)
     size = 0;
  }

Vector::Vector(const Vector& vector)
 { size = vector.size;
   v    = NULL;
   if (size)
    if ((v = (double *)malloc(size * sizeof(double))) == NULL)
     size = 0;
    else
     for (int i = 0; i < size; i++)
      *(v + i) = vector[i];
  }
```

```
Vector::~Vector()
 { if (v) free(v); }

int  Vector::SetSize(int n)
 { if (n < 0) n = 0;
   size = n;
   if (size)
    if ((v = (double *)realloc(v, size * sizeof(double))) == 0)
     size = 0;
   return size;
  }
```

```
Vector  Vector::operator =  (const Vector& vector)
 { if (this == &vector) return *this;
   size = vector.size;
   if (size)
    if ((v = (double *)realloc(v, size * sizeof(double))) == 0)
     size = 0;
    else
     for (int i = 0; i < size; i++)
      *(v + i) = vector[i];
   return *this;
  }
```

```
double& Vector::operator [] (int n)
 { if (n < 0)     n = 0;
   if (n >= size) n = size - 1;
   return *(this->v + n);
  }

Vector  Vector::operator -  () const
 { Vector res(size);

   for (int i = 0; i < size; i++)
    *(res.v + i) = -*(this->v + i);
   return res;
  }
```

```
int     Vector::operator == (const Vector& vector) const
 { if (size != vector.size) return 0;
   for (int i = 0; i < size; i++)
    if (*(this->v + i) != *(vector.v + i))
     return 0;
   return 1;
  }

int     Vector::operator != (const Vector& vector) const
 { if (size != vector.size) return 1;
   for (int i = 0; i < size; i++)
    if (*(this->v + i) != *(vector.v + i))
     return 1;
   return 0;
  }
```

```
Vector  Vector::operator +  (const Vector& vector) const
 { Vector res(size);

   if (size != vector.size) return res;
   res = *this;
   return res += vector;
  }

Vector  Vector::operator += (const Vector& vector)
 { if (size != vector.size) return * this;
   for (int i = 0; i < size; i++)
    *(this->v + i) += *(vector.v + i);
   return *this;
  }
```

```
Vector  Vector::operator -  (const Vector& vector) const
 { Vector res(size);

   if (size != vector.size) return res;
   res = *this;
   return res -= vector;
  }

Vector  Vector::operator -= (const Vector& vector)
 { if (size != vector.size) return * this;
   for (int i = 0; i < size; i++)
    *(this->v + i) -= *(vector.v + i);
   return *this;
  }
```

```
Vector  Vector::operator +  (double value) const
 { Vector res(size) = *this;
   return res += value;
  }

Vector  Vector::operator += (double value)
 { for (int i = 0; i < size; i++)
    *(this->v + i) += value;
   return *this;
  }
```

```
Vector  Vector::operator -  (double value) const
 { Vector res(size) = *this;
   return res -= value;
  }

Vector  Vector::operator -= (double value)
 { for (int i = 0; i < size; i++)
    *(this->v + i) -= value;
   return *this;
  }
```

```
double  Vector::operator *  (const Vector& vector) const
 { double res = 0;
   int i;

   if (size != vector.size) return res;
   for (i = 0; i < size; i++)
    res += *(this->v + i) * *(vector.v + i);
   return res;
  }
```

```
void main()
 { Vector v1(3), v2, v3(2), v4;
   double r;

   v2.SetSize(3);
   for (i = 0; i < v1.GetSize(); i++)
    { v1[i] = i + 1; v2[i] = i + 5; }

   v3 = v1 + v2;
   v4 = v1 - v2 + v3;
   v4 = -v4;
   v3 -= v2;
   v4 += v2;
   r  = v1 * v2;
   v4 = v1 * v2 + v3;
  }
```