Подготовительная программа по программированию на C/C++

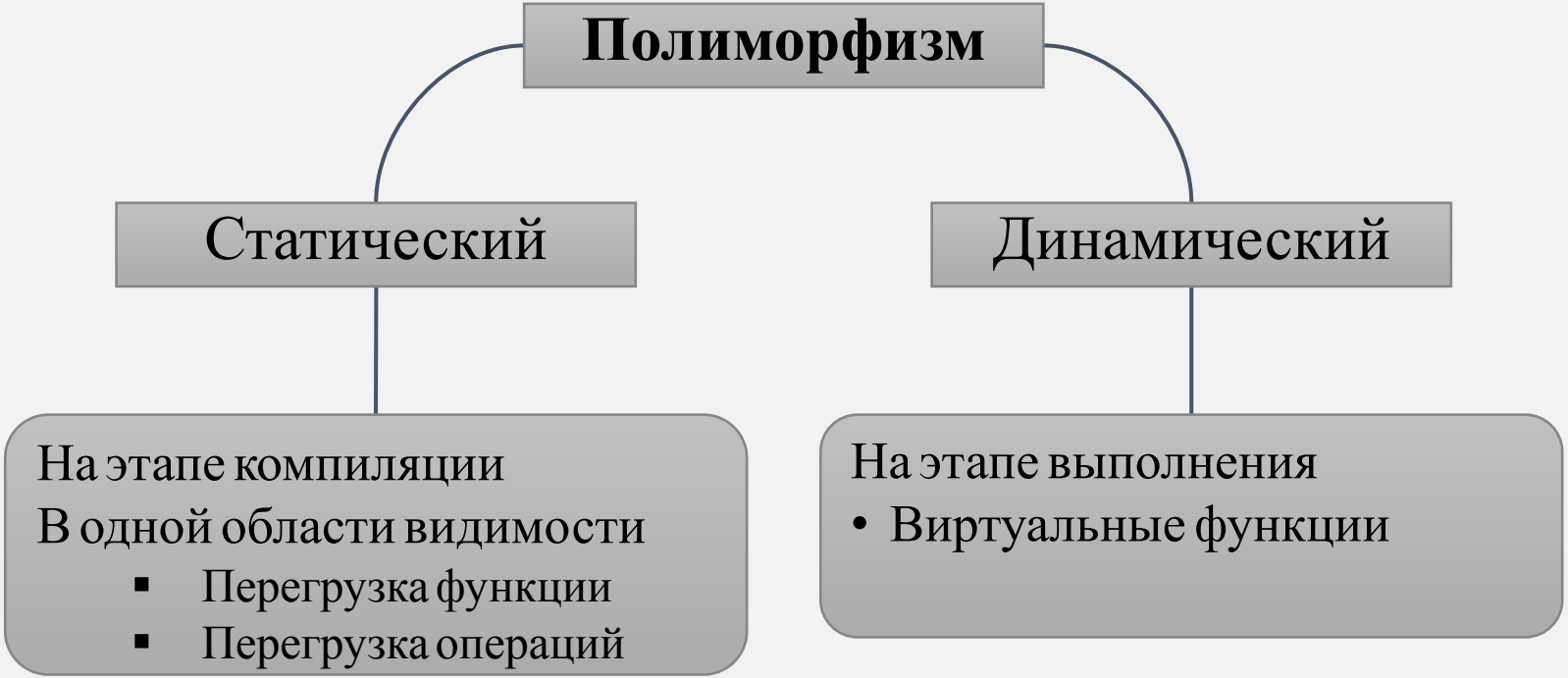
Занятие №5

Валентина Глазкова

Специальные вопросы полиморфизма. Перегрузка функций и операций

- Раннее и позднее связывание
- Динамический полиморфизм и виртуальные функции.
- Перегрузка функций. Алгоритм выбора перегруженной функции с одним и несколькими аргументами
- Перегрузка операций.
- Перегрузка с точностью до константности

Полиморфизм



Виртуальные функции: пример 1

```
class A {
public: virtual void f (int x) { h (x); cout << "A::f," << x << endl; }
        void g () { h (0); cout << "A::g" << endl; }
        virtual void h (int k) {cout << "A::h," << k << endl; }
};
class B: virtual public A {
public: void f (int y) { h (y); cout << "B::f," << y << endl; }
        void g () { h (1); cout << "B::g" << endl; }
        void h (int k) {cout << "B::h," << k << endl; }
};
int main( ){
                A a; B b; A * p = & b;
                p -> f (2);     //B::h,2          B::f,2
                p -> g ();      //B::h,0          A::g
                p -> h ();// о ш и б к а
                p -> h (3);//B::h,3
}
```

Виртуальные функции: пример 2

```
class C {
public: C (int x = 0) {}
        virtual int f (int x) { cout << "C::f," << x << endl; return h(x); }
        virtual int g () { cout << "C::g"<< endl; return 1; }
        virtual int h (int x) { cout << "C::h," << x << endl; return x; }
        virtual operator int () { return  99; }
 };
class D: public C {
public: int f (int x) { cout << "D::f," << x << endl; return h (x); }
        int g (int x) { cout << "D::g" << endl; return 1; }
        int h (int x) { cout << "D::h," << x << endl; return x; }
        D (int x = 0) {}
        operator int () { return 100; }
};
int main( ) {D d; C *t = & d; t -> f (3); t -> f (d);
                t -> g (); t -> h (5);  return 0; }
```

D::f,3 D::h,3 D::f,100 D::h,100 C::g D::h,5

Виртуальные функции: пример 3

```
struct B {
                    virtual void f (int n) { cout << “f (int) from B” ; }
                    static int i;
};
struct D:  B {
                    virtual void f (char n) { cout << “f (char) from D” ; }
};
int B::i = 1;
int main ( ) {
                    D  d;   B b1, b2;   B *pb = &d;
                    pb -> f( ‘a’ );
                    b1.i += 2; b2.i += 3; d.i += 4;
                    cout << b1.i <<  ‘  ‘ << b2.i << ’   ‘ << d.i <<  ‘  ‘ <<  B::i << endl;
                    return 0;
}
```

```
f (int) from B
10   10   10   10
```

Перегрузка функций: выбор перегруженной функции

1. Точное соответствие типов аргументов
2. Соответствие при приведении типов (type promotion)
3. Соответствие при преобразовании типов (type conversion)
4. Соответствие после применения пользовательских преобразований
5. Совпадения для функций с переменным числом аргументов (`f(...)`)

Точное соответствие

- Точное совпадение типов
- Совпадение с точностью до typedef
- Тривиальные преобразования:
  - T[] <-> T*
  - T <-> T&
  - T -> const T

Приведение типов

- Любой целочисленный тип (char, short, bool, enum, bit-field) к типу int или unsigned int
- Тип float к типу double
- При приведении типов значение гарантированно сохраняется

Преобразование типов

- Любой числовой тип к любому числовому типу
- Константа `0` к любому числовому типу или к любому указателю
- Любой указатель к указателю `void*`

Выбор перегруженной функции

- Если существует несколько разновидностей функции с одинаковыми приоритетами, то возникает ситуация неоднозначности (ambiguity)
- Неоднозначность может быть порождена как самим объявлением функций, так и конкретным вызовом
- Для успешной компиляции код не должен содержать неоднозначностей

Пример 1

```
void f(int*) {}

void f(char*) {}

void main()

{

    f(0); // Ambiguous call

}
```

```
void f(int) {}
void f(double) {}

void main()
{
  f('a'); // Type promotion char -> int
}
```

Пример 2

```
void f(int) {}

void f(char) {}

void main()

{

  f('a'); // Exact match

}
```

```
void f(int) {}
void f(float) {}

void main()
{
  f(1.0); // Ambiguous call
}
```

Суффиксы числовых литералов

```
int i = 7;
long l = 7L;
unsigned int ui = 7U;
unsigned long ul = 7UL;
float f = 3.14F;
double d = 3.14;
long double ld = 3.14L;
```

При объявлении числовых литералов можно в явном виде указывать их тип с помощью суффиксов

Пример 3

```
void f(int) {}
void f(short) {}
void f(unsigned) {}
void main()
{
  f('a'); // Type promotion
}
```

```
void f(bool) {}
void f(short) {}
void f(unsigned) {}
void main()
{
  f('a'); // Ambiguous call
}
```

Пример 4

```
void f(char) {}

void f(int*) {}

void main()

{

  f(5); // Type conversion

}
```

```
void f(char) {}
void f(long) {}
void f(int*) {}
void main()
{
  f(5); // Ambiguous call
}
```

Пример 5

```
void f(unsigned) {} // 1
void f(char) {}     // 2
void f(int) {}      // 3
void f(long) {}     // 4
void main()
{
  unsigned char c = 'a';
  f(c);     // 3
  f(false); // 3
  f(0U);    // 1
  f(1.0);   // Ambiguous call
}
```

Пример 6

```
void f(double*) {} // 1
void f(void*) {}   // 2
void f(char) {}    // 3
void f(char*) {}   // 4
```

(*) Строковые литералы в стандарте имеют тип const char*, но фактически в MSVS и в g++ трактуются как char*. При этом g++ фиксирует предупреждение компилятора в отличие от MSVS. Т.о. по стандарту будет ошибка, а фактически вариант 4.

```
void main()
{
  int i = 0;
  double d = 0;
  const char* s = "nice";
  f(i);      // 3
  f(&i);     // 2
  f(d);      // 3
  f(&d);     // 1
  f("omg"); // (*)
  f(s);      // No overload
  f('a');    // 3
}
```

Перегрузка функций с несколькими аргументами

- Выбираются функции с соответствующим числом параметров
- Из них убираются неотождествляемые хотя бы по одному из фактических параметров
- Для каждого фактического параметра строится множество функций, наиболее подходящих по этому параметру
- Находится пересечение этих множеств
- Если пересечение состоит из единственной функции, она и вызывается
- Иначе фиксируется ошибка

Пример 7

```
void f(long, int) {} // Conversion / promotion
void f(int, int) {}  // Exact match / promotion
void main()
{
  f(2, 'a');
}
```

Пример 8

```
void f(int, char) {}    // Promotion / conversion
void f(long, double) {} // Conversion / exact match
void main()
{
  f('a', 2.7); // Ambiguous call
}
```

```
void f(char, int) {}    // Conversion / conversion
void f(float, short) {} // Conversion / conversion

void main()
{
  f(1.0, 1.0); // Ambiguous call
}
```

Пример 9

```
void f(char) {}            // 1
void f(double, int = 0) {} // 2
void f(double) {}          // 3

void main()
{
  f('a');  // 1
  f(5, 7); // 2
  f(7);    // Ambiguous call
  f(5.7);  // Ambiguous call
}
```

Пример 10

```
void g(int = 0, int = 0) {}
void g(const char *) {}
```

```
void main()
{
  g(); //1
  g(«abc"); //2
  g(2); //1
  g('+', 3); //1
}
```

Перегрузка операторов

Для любого класса можно перегрузить любой оператор, кроме указанных:

Operator	Name
.	Member selection
.*	Pointer-to-member selection
::	Scope resolution
?:	Conditional (ternary)
sizeof	Not actually an operator
typeid	RTTI operator

Перегрузка операторов

Таким образом, для любого класса доступны для перегрузки следующие операторы:

Operator	Name	Type
,	Comma	Binary
!	Logical NOT	Unary
!=	Inequality	Binary
%	Modulus	Binary
%=	Modulus assignment	Binary
&	Bitwise AND	Binary
&	Address-of	Unary
&&	Logical AND	Binary
&=	Bitwise AND assignment	Binary
()	Function call	—
()	Cast Operator	Unary
*	Multiplication	Binary
*	Pointer dereference	Unary
*=	Multiplication assignment	Binary
+	Addition	Binary
+	Unary Plus	Unary

Operator	Name	Type
++	Preincrement / postincrement	Unary
+=	Addition assignment	Binary
–	Subtraction	Binary
–	Unary negation	Unary
––	Predecrement / postdecrement	Unary
–=	Subtraction assignment	Binary
–>	Member selection	Binary
–>*	Pointer-to-member selection	Binary
/	Division	Binary
/=	Division assignment	Binary
<	Less than	Binary
<<	Left shift	Binary
<<=	Left shift assignment	Binary
<=	Less than or equal to	Binary
=	Assignment	Binary
==	Equality	Binary

Operator	Name	Type
>	Greater than	Binary
>=	Greater than or equal to	Binary
>>	Right shift	Binary
>>=	Right shift assignment	Binary
[ ]	Array subscript	—
^	Exclusive OR	Binary
^=	Exclusive OR assignment	Binary
|	Bitwise inclusive OR	Binary
|=	Bitwise inclusive OR assignment	Binary
||	Logical OR	Binary
~	One's complement	Unary
delete	Delete	—
new	New	—
conversion operators	conversion operators	Unary

Перегрузка операторов

- При перегрузке определенного оператора необходимо лишь указать соответствующее количество аргументов операторной функции
- Приоритет операций при перегрузке менять нельзя
- Тип аргументов, как и тип возвращаемого значения может быть произвольным
- Операторная функция представляет собой обычную функцию с возможностью сокращенной формы вызова

Пользовательские операторы

- Для всех встроенных типов данных существуют встроенные операторы
- Переопределять можно только поведение для пользовательских классов

```
// Error: 'operator +' must have at least
// one formal parameter of class type
int operator+(int a, int b) {
  return a * b;
}
```

Оператор ==

```
class Complex
{
  double m_Real;
  double m_Imag;
public:
  Complex(double in_Real = 0.0, double in_Imag = 0.0)
:
    m_Real(in_Real), m_Imag(in_Imag) {}
  bool operator==(const Complex& in_Target) const {
    return m_Real == in_Target.m_Real &&
           m_Imag == in_Target.m_Imag;
  }
};
```

```
void main()
{
  Complex a(3.0, 0.0), b(3.0), c;
  cout << (a == b ? "a == b" : "a != b") << endl; // a == b
  cout << (b == c ? "b == c" : "b != c") << endl; // b != c
}
```

Оператор +

```
class Complex
{
  double m_Real;
  double m_Imag;
public:
  /* ... */
  Complex operator+(const Complex& in_Target) const {
    return Complex(m_Real + in_Target.m_Real,
                   m_Imag + in_Target.m_Imag);
  }
};
void main()
{
  Complex a(3.0, 0.0), b(3.0, 4.0), c;
  a = a + 2; // a = (5.0, 0.0)
  c = a + b; // c = (6.0, 4.0)
}
```

Константная ссылка, чтобы можно было передавать объекты другого типа. Можно передавать по значению, но это вызовет накладное копирование.

Не изменяет состояние объекта. Обязательно указать модификатор const.

Автоматически преобразуется к типу Complex, т.к. есть соответствующий конструктор

Глобальные операторные функции

```
void main()
{
  Complex a(3.0, 0.0), b(3.0, 4.0);
  a = a + 2;
  a = a.operator+(2); // OK
  b = 3 + b;
  b = 3.operator+(b); // ???
}
```

- Если нет доступа к классу левого операнда, необходимо перегружать операторную функцию не как член класса, а как глобальную
- В этом случае ее часто бывает удобно сделать дружественной классу

Глобальный оператор +

```
class Complex
{
  /* ... */
  friend Complex operator+(const Complex&, const Complex&);
};
Complex operator+(const Complex& in_Left,
                  const Complex& in_Right) {
  return Complex(in_Left.m_Real + in_Right.m_Real,
                 in_Left.m_Imag + in_Right.m_Imag);
}
void main()
{
  Complex a(3.0, 0.0), b(3.0, 4.0);
  a = a + 2; // OK: operator+(a, 2);
  b = 3 + b; // OK: operator+(3, b);
}
```

Объявляем оператор как дружественную функцию, чтобы получить доступ к закрытым полям

В прототипе достаточно указать только типы аргументов, без их имен

Глобальный оператор <<

```
class Complex
{
  /* ... */
  friend ostream& operator<<(ostream&, const Complex&);
};
ostream& operator<<(ostream& io_Stream,
                    const Complex& in_Target) {
  io_Stream << "(" << in_Target.m_Real <<
               ", " << in_Target.m_Imag << ")";
  return io_Stream;
}
void main()
{
  Complex a(1.0, 2.0), b(3.0, 4.0);
  cout << a + b << endl; // (4, 6)
  operator<<(cout, a + b).operator<<(endl);
}
```

Возвращаем ссылку на поток, чтобы можно было писать цепочки вывода

Ссылка неконстантная, т.к. вывод в поток будет изменять его состояние

Глобальный оператор >>

```
class Complex
{
/* ... */
  friend istream& operator>>(istream&, Complex&);
};
istream& operator>>(istream& io_Stream,
                    Complex& in_Target) {
  io_Stream >> in_Target.m_Real >> in_Target.m_Imag;
  return io_Stream;
}
void main()
{
  Complex a, b;
  cin >> a >> b;
}
```

В отличие от оператора вывода, ссылка на объект неконстантная, т.к. его значение должно измениться

Унарный оператор -

```
class Complex
{
  /* ... */
  Complex operator-() const {
    return Complex(-m_Real, -m_Imag);
  }
};
void main()
{
  Complex a(3.0, 4.0), b;
  b = -a; // (-3.0, -4.0)
}
```

Является перегруженным по отношению к бинарному оператору

Префиксная и постфиксная формы унарного оператора ++

```
class Complex
{
  /* ... */
  Complex& operator++() {
    m_Real++;
    return *this;
  }
  Complex operator++(int) {
    Complex t(*this);
    m_Real++;
    return t;
  }
};
```

Возвращается ссылка на текущий объект

Операторы не являются константными

Фиктивный параметр типа int для задания постфиксной формы

Возвращается временный объект

```
void main()
{
  Complex a, b;
  ++++a; // (2.0, 0.0)
  b++++; // (1.0, 0.0)
}
```

Перегрузка операторов

- Некоторые операторы можно перегружать только как члены класса:

Operator	Name
=	Assignment
[ ]	Array subscript
( )	Function call
conversion operators	Type cast operators

Оператор присваивания

```
class Complex
{
  /* ... */
  Complex& operator=(const Complex& in_Src) {
    m_Real = in_Src.m_Real;
    m_Imag = in_Src.m_Imag;
    return *this;
  }
};
void main()
{
  Complex a(1.0, 2.0), b(3.0, 4.0), c;
  c = a + b;
  a = b = c;
}
```

В данном случае было бы достаточно создаваемого по умолчанию оператора присваивания, выполняющего поверхностное копирование

Возвращаем ссылку на текущий объект, чтобы можно было формировать цепочки присваивания.

Оператор преобразования типа

```
class Complex
{
  /* ... */
  operator double() const {
    return sqrt(m_Real * m_Real + m_Imag * m_Imag);
  }
};
void main()
{
  Complex a(3.0, 4.0);
  double x = a;  // x == 5
  x = a.operator double();
}
```

Для оператора преобразования типа не указывается тип возвращаемого значения: он неявно полагается равным типу преобразования

Не должен изменять состояние объекта, поэтому объявляется как константный метод

Пользовательские и стандартные операторы преобразования типа (1/2)

```
struct A {
  operator double();
};
struct B {
  operator A();
};
void main()
{
  A a;
  double x = a;
  float y = a;
  B b;
  x = b;
  y = static_cast<A>(b);
}
```

Возможно выполнение стандартного преобразования типа после пользовательского

Ошибка: автоматически не выполняется несколько пользовательских преобразований подряд

С помощью конструкции `static_cast<>()` можно в явном виде вызвать оператор преобразования типа. Как пользовательский, так и стандартный.

Пользовательские и стандартные операторы преобразования типа (2/2)

```
struct A
{
  A(int i);
  operator int() const;
};
int operator+(
  const A& a1,
  const A& a2);
void main()
{
  A a = 1;
  int i = 2;
  i = i + a;
}
```

Неоднозначность между пользовательским оператором сложения для класса A и стандартным оператором сложения для int

```
'operator +' : 2 overloads have similar conversions

could be 'int operator +(const A &,const A &)'  or
'built-in C++ operator+(int, int)'
```

Оператор ()

- **Классы, у которых перегружен оператор (), называются функторами, т.к. их объектами можно оперировать как функциями**

```
class CloserTo
{
  double m_Value;
public:
  CloserTo(double in_Value) : m_Value(in_Value) {}
  bool operator()(double in_A, double in_B) {
    return abs(in_A - m_Value) < abs(in_B - m_Value);
  }
};
```

Список формальных параметров функции. Может содержать любое количество.

Тип возвращаемого значения функции

Оператор ()

```
void main()
{
  double x = 2, y = 7;
  CloserTo cx(x), cy(y);
  double a[] = { 9, 1, 7.2, 2.3, 5, 1.5, 6.9 };
  int count = sizeof a / sizeof(double);
  double ax = 0, ay = 0;
  for (int i = 0; i < count; i++)
  {
    if (cx(a[i], ax)) ax = a[i];
    if (cy(a[i], ay)) ay = a[i];
  }
  cout << "closest to " << x << " in a: " << ax << endl;
  cout << "closest to " << y << " in a: " << ay << endl;
}
```

Инициализация функторов

А таким образом удобно определять размер статических массивов

Использование функторов в функциональной форме

Оператор []

```
class Complex
{
  /* ... */
  double& operator[](int in_Index) {
    return in_Index == 0 ? m_Real : m_Imag;
  }
};
void main()
{
  Complex a(3.0, 4.0);
  a[1] = 5.0;
  double r = a[0]; // r == 3
  double i = a[1]; // i == 5
}
```

Возможен только один аргумент

Возвращается ссылка на значение, чтобы можно было использовать как l-value

Перегрузка с точностью до «константности» (1/3)

- Если метод не отмечен модификатором `const`, то его нельзя вызвать для константного объекта, даже если по факту не происходит изменения объекта
- Можно создавать методы, перегруженные с точностью до модификатора `const`

```
bool is_pure_imaginary(const Complex& c)
{
  return c[0] == 0; // Cannot modify const object
}
```

Перегрузка с точностью до «константности» (2/3)

Константная версия оператора возвращает константную ссылку, чтобы гарантировать неизменность объекта

```
class Complex
{
  /* ... */
  double& operator[](int in_Index) {
    return in_Index == 0 ? m_Real : m_Imag;
  }
  const double& operator[](int in_Index) const {
    return in_Index == 0 ? m_Real : m_Imag;
  }
};
```

```
bool is_pure_imaginary(const Complex& c)
{
  return c[0] == 0; // OK: const-flavor used
}
```

Перегрузка с точностью до «константности» (3/3)

- Константность вызываемого метода соответствует константности объекта
- Чтобы вызвать константную версию метода для неконстантного объекта, можно обратиться к нему через константную ссылку

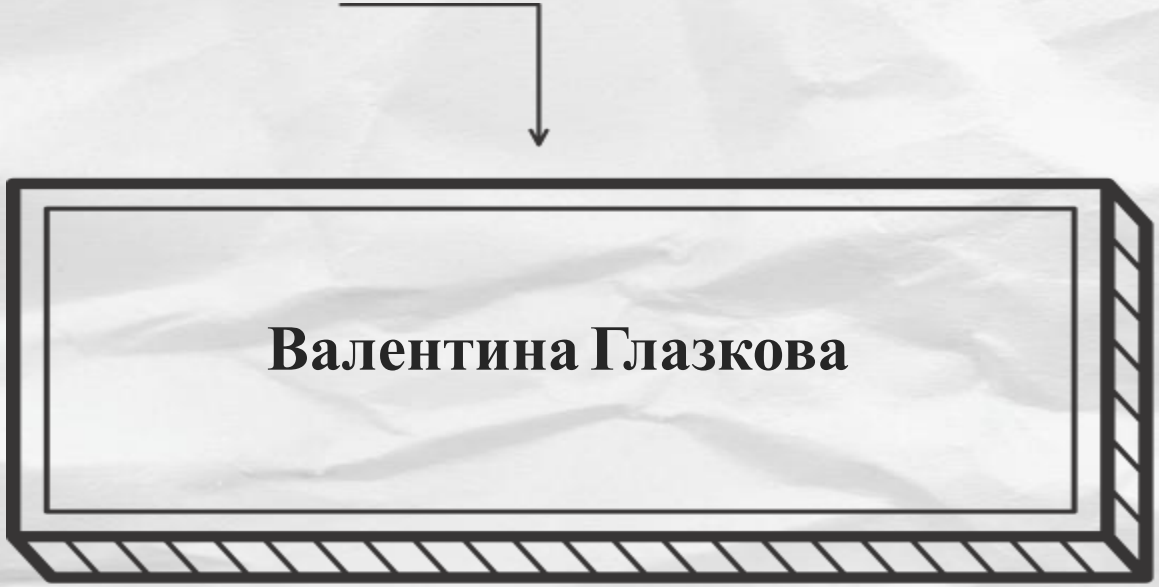
```
void main()
{
  Complex a(3.0, 4.0);
  a[0];  // non-const operator[]
  const Complex& const_a = a;
  const_a[0]; // const operator[]
}
```

Перегрузка операторов для типов-перечислений

- Перечисление не является классом, но является пользовательским типом

```
enum Day { mo, tu, we, th, fr, sa, su };
Day& operator++(Day& d) {
  d = Day(d + 1);
  return d;
}
void main()
{
  Day d = fr;
  ++++d; // d == su
}
```

Возможна перегрузка только с помощью глобальных операторных функций



Спасибо за внимание!