

# Объектно-ориентированное программирование

## Лекции 3–4. Абстрактные типы данных и инкапсуляция

П. А. Макаров



СЫКТЫВКАРСКИЙ  
ЛЕСНОЙ  
ИНСТИТУТ

7 и 14 октября 2022 г.

1. Объекты, методы и инкапсуляция
2. Статический полиморфизм
3. Конструкторы
4. Ссылки
5. Модификатор `const` и константные методы
6. Динамическая память и копирование объектов
7. Значения параметров по умолчанию
8. Заголовок класса и его реализация. Области видимости
9. Инициализация членов класса в конструкторе
10. Перегрузка операций функциями вне класса
11. Дружественные функции и классы
12. Статические поля и методы
13. А как с этим в Python?
14. Контрольные вопросы и задания

# Объекты, методы и инкапсуляция

Пример объекта — комплексные числа

## Определение и основные формы представления

$$\forall x, y \in \mathbb{R} \quad \exists z \in \mathbb{C} : z = (x, y). \quad (1)$$

$$z = x + iy, \quad i = \sqrt{-1}. \quad (2)$$

$$z = r(\cos \varphi + i \sin \varphi), \quad \forall r, \varphi \in \mathbb{R} : r \geq 0. \quad (3)$$

$$z = r e^{i\varphi}. \quad (4)$$

## Связанные понятия

$$x = \operatorname{Re} z, \quad y = \operatorname{Im} z. \quad (5)$$

$$r = \operatorname{Mod} z = |z| = \sqrt{x^2 + y^2}. \quad (6)$$

$$\varphi = \operatorname{Arg} z = \arctan \frac{y}{x}. \quad (7)$$

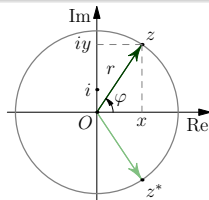


Рис. 1: Геометрическая интерпретация

# Объекты, методы и инкапсуляция

## Подход к комплексным числам “ленивого” программиста C++

```
1 #include <iostream>
2 #include <complex>
3
4 using namespace std;
5
6 int main() {
7     complex<double> z1;
8     z1 = 2;
9     z1 += complex<double>(0, 3);
10    cout << "z1 = " << z1 << endl;
11    cout << "|z1| = " << abs(z1) << endl;
12    cout << "Arg(z1) = " << arg(z1) << endl << endl;
13
14    complex<double> z2(z1.real(), -z1.imag());
15    cout << "z2 = " << z2 << endl;
16    cout << "|z2| = " << abs(z2) << endl;
17    cout << "Arg(z2) = " << arg(z2) << endl;
18
19    return 0;
20 }
```

Листинг 1: Пример работы с комплексными числами на языке C++

---

```
1 import cmath
2
3 z1 = 2
4 z1 += 3j
5 print('z1 =', z1)
6 print('|z1| =', abs(z1))
7 print('Arg(z1) =', cmath.phase(z1), '\n')
8
9 z2 = z1.conjugate()
10 print('z2 =', z2)
11 print('|z2| =', abs(z2))
12 print('Arg(z2) =', cmath.phase(z2))
```

---

Листинг 2: Пример работы с комплексными числами на языке Python

# Объекты, методы и инкапсуляция

“Самостоятельный” подход в C++

---

```
1 struct str_complex {
2     double re, im;
3     double modulo()
4         return sqrt(re*re + im*im);
5 };
```

---

Листинг 3: Пример структуры комплексного числа

---

```
1 double modulo(struct str_complex *c) {
2     return sqrt(c->re*c->re + c->im*c->im);
3 }
```

---

Листинг 4: Функция вычисления модуля в C

---

```
1 str_complex z;
2 z.re = 2.7;
3 z.im = 3.8;
4 double mod = z.modulo();
```

---

Листинг 5: Использование методов

# Объекты, методы и инкапсуляция

Адрес объекта и ключевое слово `this`

---

```
1 struct str_complex {
2     double re, im;
3     double modulo()
4         return sqrt(this->re*this->re + this->im*this->im);
5 }
```

---

Листинг 6: Применение ключевого слова `this`

# Объекты, методы и инкапсуляция

Механизм защиты. Ключевые слова `public` и `private`

---

```
1 struct str_complex {
2 private:
3     double re, im;
4 public:
5     double modulo()
6         return sqrt(re*re + im*im);
7 };
```

---

Листинг 7: Применение ключевых слов `public` и `private`



# Объекты, методы и инкапсуляция

Механизм защиты. Ключевые слова `public` и `private`

---

```
1 struct str_complex {
2 private:
3     double re, im;
4 public:
5     double modulo()
6         return sqrt(re*re + im*im);
7 };
```

---

Листинг 7: Применение ключевых слов `public` и `private`

В чем проблема этого фрагмента кода?

# Объекты, методы и инкапсуляция

## Простейшая инициализация объекта

---

```
1 struct str_complex {
2     private:
3         double re, im;
4     public:
5         void set(double a_re, double a_im) {
6             re = a_re;
7             im = a_im;
8         }
9         double modulo()
10            return sqrt(re*re + im*im);
11 };
```

---

**Листинг 8:** Пример метода, инициализирующего объект

---

```
1 str_complex z;
2 z.set(2.7, 3.8);
3 double mod = z.modulo();
```

---

**Листинг 9:** Применение метода, инициализирующего объект

# Объекты, методы и инкапсуляция

## Простейшая инициализация объекта

```
1 struct str_complex {
2 private:
3     double re, im;
4 public:
5     void set(double a_re, double a_im) {
6         re = a_re;
7         im = a_im;
8     }
9     double modulo()
10         return sqrt(re*re + im*im);
11 };
```

Листинг 8: Пример метода, инициализирующего объект

```
1 str_complex z;
2 z.set(2.7, 3.8);
3 double mod = z.modulo();
```

Листинг 9: Применение метода, инициализирующего объект

В чем проблема этого фрагмента кода?

### Определение 1

**Конструктор** — это метод, описывающий необходимый порядок действий при создании объекта данного типа. Имя конструктора совпадает с именем описываемого типа, а тип возвращаемого значения указывать нельзя.

```
1 struct str_complex {
2     private:
3         double re, im;
4     public:
5         str_complex(double a_re, double a_im) {
6             re = a_re;
7             im = a_im;
8         }
9         double modulo()
10             return sqrt(re*re + im*im);
11 };
```

Листинг 10: Описание конструктора объекта

# Объекты, методы и инкапсуляция

## Использование конструкторов

---

```
1 str_complex z(2.7, 3.8);  
2 double mod = z.modulo();
```

---

Листинг 11: Использование конструктора

---

```
1 double mod = str_complex(2.7, 3.8).modulo();
```

---

Листинг 12: Анонимные объекты

### Замечание 1

*В языке C++ любая переменная создаётся с помощью конструкторов.*

---

```
1 int a(3); // int a = 3;
```

---

Листинг 13: Инициализация переменных и конструкторы

```
1 class Complex {
2     double re, im;
3 public:
4     Complex(double a_re, double a_im) {
5         re = a_re;
6         im = a_im;
7     }
8     double get_re()
9         return re;
10    double get_im()
11        return im;
12    double modulo()
13        return sqrt(re*re + im*im);
14    double argument()
15        return atan2(im, re);
16 };
```

---

Листинг 14: Реализация комплексного числа в виде класса

### Определение 2

*Деструктор — это метод, вызов которого автоматически помещается компилятором в код программы в любой ситуации, когда объект прекращает своё существование.*

```
1 class File {
2     int fd;    // file descriptor
3 public:
4     File() { fd = -1; }
5     bool OpenRO (const char * name) {
6         fd = open(name, O_RDONLY);
7         return (fd != -1);
8     }
9     // ...
10    ~File() {
11        if (fd != -1) close(fd);
12    }
13 };
```

**Листинг 15:** Класс File, инкапсулирующий дескриптор файла

# Статический полиморфизм

## Определение и использование перегруженных функций

---

```
1 void print(int n) {
2     printf("%d\n", n);
3 }
4 void print(const char *s) {
5     printf("%s\n", s);
6 }
7 void print() {
8     printf("Hello, World!\n");
9 }
```

---

Листинг 16: Перегрузка имён функций

---

```
1 print(50);
2 print("Good day");
3 print();
```

---

Листинг 17: Использование перегруженных функций



### Замечание 2

*Перегрузку имён функций необходимо использовать очень аккуратно, так как это может привести к неопределённому поведению (UB — Undefined Behavior).*

```
1 void f(const char *str)
2     printf("This is string: %s\n", str);
3 void f(float num)
4     printf("This is float-point number: %f\n", num);
```

Листинг 18: Пример определения перегруженных функций, приводящий к UB

В чем подвох?

### Замечание 2

*Перегрузку имён функций необходимо использовать очень аккуратно, так как это может привести к неопределённому поведению (UB — Undefined Behavior).*

```
1 void f(const char *str)
2     printf("This is string: %s\n", str);
3 void f(float num)
4     printf("This is float-point number: %f\n", num);
```

**Листинг 18:** Пример определения перегруженных функций, приводящий к UB

### В чем подвох?

```
1 f("string");
2 f(2.5);
3 f(1);
4 f(0);
```

**Листинг 19:** Вызовы функций, приводящие к UB

# Статический полиморфизм

## Переопределение символов стандартных операций

```
1 class Complex {
2     double re, im;
3 public:
4     Complex(double a_re, double a_im) {re = a_re; im = a_im;}
5     double get_re() return re;
6     double get_im() return im;
7     double modulo() return sqrt(re*re + im*im);
8     double argument() return atan2(im, re);
9     Complex operator+(Complex op2)
10        return Complex(re+op2.re, im+op2.im);
11    Complex operator-(Complex op2)
12        return Complex(re-op2.re, im-op2.im);
13    Complex operator*(Complex op2)
14        return Complex(re*op2.re-im*op2.im, re*op2.im+im*op2.re);
15    Complex operator/(Complex op2) {
16        double dvs = op2.re*op2.re + op2.im*op2.im;
17        Complex res((re*op2.re + im*op2.im)/dvs, (im*op2.re - re
18            *op2.im)/dvs);
19        return res; }
20 };
```

Листинг 20: Пример: operator.cpp

# Статический полиморфизм

## Работа с перегруженными символами стандартных операций

---

```
1 a = b.operator+(c);
```

---

Листинг 21: Перегруженные операции (вариант 1)

---

```
1 a = b + c;
```

---

Листинг 22: Перегруженные операции (вариант 2)

# Статический полиморфизм

## Работа с перегруженными символами стандартных операций

---

```
1 a = b.operator+(c);
```

---

Листинг 21: Перегруженные операции (вариант 1)

---

```
1 a = b + c;
```

---

Листинг 22: Перегруженные операции (вариант 2)

---

```
1 Complex x(1, 0);  
2 Complex y(0, 1);  
3 double mod = (x + y).modulo();
```

---

Листинг 23: Перегруженные операции (реалистичный вариант)

### Исключения

1. тернарная условная операция  $a ? b : c$
2. операция прямого выбора поля структуры или класса .

# Конструкторы

## Проблема определения объектов

### Вернёмся к Листингу 14.

---

```
1 class Complex {
2     double re, im;
3 public:
4     Complex(double a_re, double a_im) {
5         re = a_re;
6         im = a_im;
7     }
8     double get_re()
9         return re;
10    double get_im()
11        return im;
12    double modulo()
13        return sqrt(re*re + im*im);
14    double argument()
15        return atan2(im, re);
16 };
```

---

# Конструкторы

## Проблема определения объектов

Вернёмся к Листингу 14.

---

```
1 class Complex {
2     double re, im;
3 public:
4     Complex(double a_re, double a_im) {
5         re = a_re;
6         im = a_im;
7     }
8     double get_re()
9         return re;
10    double get_im()
11        return im;
12    double modulo()
13        return sqrt(re*re + im*im);
14    double argument()
15        return atan2(im, re);
16 };
```

---

---

```
1 Complex z;
```

---

Листинг 24: Ошибка определения объекта

# Конструкторы

## Решение проблемы с помощью конструктора по умолчанию

```
1 class Complex {
2     double re, im;
3 public:
4     Complex() {
5         re = 0;
6         im = 0;
7     }
8     Complex(double a_re, double a_im) {
9         re = a_re;
10        im = a_im;
11    }
12    // ...
13};
```

### Листинг 25: Перегрузка конструкторов

```
1 Complex z;
2 Complex a[50];
```

### Листинг 26: Примеры использования



---

```
1 int a = 5;  
2 float b = 2.5;  
3 double c = a + b;
```

---

Листинг 27: Простейший пример преобразования типов

(int) → (float) → (double)

---

```
1 int a = 5;  
2 float b = 2.5;  
3 double c = a + b;
```

---

Листинг 27: Простейший пример преобразования типов

(int) → (float) → (double)

---

```
1 void f(float) {  
2     // ...  
3 }  
4 f(2.8);  
5 f(25);
```

---

Листинг 28: Второй пример преобразования типов

$(\text{type A}) \rightarrow (\text{type B})$

Ключевое слово `explicit` запрещает компилятору использовать указанное преобразование для выполнения неявных преобразований.

# Конструкторы

## Обобщение процедуры преобразования типов и конструктор преобразования

(type A)  $\rightarrow$  (type B)

Ключевое слово `explicit` запрещает компилятору использовать указанное преобразование для выполнения неявных преобразований.

---

```
1 Complex(double a) {  
2     re = a;  
3     im = 0;  
4 }
```

---

Листинг 29: Пример приведения  $\mathbb{R} \rightarrow \mathbb{C}$

---

```
1 Complex u(2.5);  
2 Complex v = 3.8;  
3 void f(Complex a) {  
4     // ...  
5 }  
6 f(2.7);
```

---

Листинг 30: Примеры использования конструктора преобразования

### Определение 3

*Ссылка в C++ — это особый вид объектов данных, реализуемый путём хранения адреса объекта, но семантически эквивалентный самому объекту на который ссылается.*

```
1 int i;           // i - variable
2 int* p = &i;    // p - pointer to i
3 int& r = i;     // r - reference to i
4
5 i++;
6 (*p)++;
7 r++;
```

**Листинг 31:** Простейший пример на указатели и ссылки

# Ссылки

Пример: Поиск элементов в массиве с помощью указателей

```
1 void max_min(float *arr, int len, float* min, float* max) {
2     int i;
3     *min = arr[0];
4     *max = arr[0];
5     for(i = 1; i < len; i++) {
6         if(*min > arr[i])
7             *min = arr[i];
8         if(*max < arr[i])
9             *max = arr[i];
10    }
11 }
```

Листинг 32: Реализация с помощью указателей

```
1 float a[500];
2 float min, max;
3 // ...
4 max_min(a, 500, &min, &max);
```

Листинг 33: Вызов функции

# Ссылки

Пример: Поиск элементов в массиве с помощью ссылок

```
1 void max_min(float *arr, int len, float& min, float& max) {
2     int i;
3     min = arr[0];
4     max = arr[0];
5     for(i = 1; i < len; i++) {
6         if(min > arr[i])
7             min = arr[i];
8         if(max < arr[i])
9             max = arr[i];
10    }
11 }
```

Листинг 34: Реализация с помощью ссылок

```
1 float a[500];
2 float min, max;
3 // ...
4 max_min(a, 500, min, max);
```

Листинг 35: Вызов функции

# Ссылки

## Ссылка как тип возвращаемого значения

---

```
1 int& find_var(/* params */);
2 // ...
3 int x = find_var(/* params */) + 5;
4 find_var(/* params */) = 3;
5 find_var(/* params */) *= 10;
6 find_var(/* params */)++;
7 int y = ++find_var(/* params */);
```

---

**Листинг 36:** Применение ссылки в качестве возвращаемого значения



# Модификатор `const` и константные методы

## Определение и назначение модификатора `const`

### Определение 4

*Ключевое слово `const` позволяет ввести именованные константы. Используемая при этом память не подлежит изменению.*

---

```
1 const int max_line_length = 100;      // C++ style
2
3 #define MAX_LINE_LENGTH 100          // C style
```

---

### Листинг 37: Пример

Впервые ключевое слово `const` появилось в C++, но затем вошло и в стандарт языка C.

# Модификатор const и константные методы

## Указатели на константу и константные указатели

---

```
1 const char *p;           // p - pointer to constant
2 p = "A string";        // alright
3 *p = 'a';              // error!
4 p[5] = 'b';           // error!
```

---

Листинг 38: Указатели на константу

---

```
1 char buf[20];
2 char * const p = buf + 5; // p - constant pointer
3 p++;                     // error!
4 *p = 'a';                // alright
5 p[5] = 'b';              // alright
```

---

Листинг 39: Константные указатели

# Модификатор const и константные методы

## Константные ссылки

---

```
1 int i;
2 const int &r = i;    // r - reference to constant
3 int x = r + 5;      // alright
4 i = 7;              // alright
5 r = 12;             // error!
6
7 const int j = 5;
8 int &jr = j;        // error!
9 const int &jcr = j; // alright
```

---

### Листинг 40: Константные ссылки

---

```
1 Complex operator+(const Complex & op2) {
2     return Complex(re + op2.re, im + op2.im);
3 }
```

---

Листинг 41: Применение константных ссылок (нужно сравнить с Листингом 20)

# Модификатор const и константные методы

## Временные, анонимные объекты и константные ссылки

---

```
1 Complex x, y;  
2 // ...  
3 Complex Imag_1(0,1);  
4 y = x * Image_1;
```

---

Листинг 42: Исходный пример

---

```
1 y = x * Complex(0, 1);
```

---

Листинг 43: Пример анонимного объекта

---

```
1 Complex rez = x + y + z;    // operator+()
```

---

Листинг 44: Пример временного объекта

Свойства временных и анонимных объектов:

1. ограниченное время жизни;
2. на временный или анонимный объект нельзя ссылаться неконстантной ссылкой.

# Модификатор const и константные методы

## Константные методы

Для работы с константными объектами в C++ предусмотрены константные методы.

---

```
1 class C1 {
2     // ...
3     void method(int a, int b) const {
4         // ...
5     }
6 };
```

---

Листинг 45: Синтаксис описания константного метода

---

```
1 void f(const MyClass * p) {
2     p->my_method();
3 }
```

---

Листинг 46: Пример вызова метода константного объекта

### Замечание 3

*Рекомендуется все методы, которые по своему смыслу не должны изменять состояние объекта, обязательно помечать как константные, разрешая этим вызывать их для константных объектов.*

### Замечание 4

*В теле константного метода недопускаются вызовы неконстантных методов для того же объекта.*

### Замечание 5

*Внутри константного метода произвольного класса или структуры `C` указатель `this` имеет тип `const C *`.*

# Динамическая память и копирование объектов

## Операции работы с динамической памятью

- Средства C: `malloc()`, `calloc()`, `realloc()`, `free()`;
- Средства C++: операции `new` и `delete`.

---

```
1 int *p1;  
2 p1 = new int;  
3  
4 Complex *p2 = new Complex(2.7, 3.2);  
5  
6 delete p1, p2;
```

---

Листинг 47: Пример использования операций `new` и `delete`

---

```
1 int *p = new int [100];  
2 delete [] p;
```

---

Листинг 48: Векторная форма операций `new` и `delete`

# Динамическая память и копирование объектов

## Динамические поля классов

---

```
1 class Class1 {
2     int *p;
3 public:
4     Class1() {
5         p = new int[20];
6     }
7     ~Class1() {
8         delete [] p;
9     }
10 };
```

---

Листинг 49: Пример класса, использующего динамический массив



# Динамическая память и копирование объектов

Пример ситуации, в которой происходит копирование объекта

```
1 void f(Class1 x) {  
2     // ...  
3 }  
4  
5 int main() {  
6     // ...  
7     Class1 c;  
8     f(c);  
9     // ...  
10 }
```

**Листинг 50:** Передача объекта как параметра функции

# Динамическая память и копирование объектов

Пример ситуации, в которой происходит копирование объекта

```
1 void f(Class1 x) {  
2     // ...  
3 }  
4  
5 int main() {  
6     // ...  
7     Class1 c;  
8     f(c);  
9     // ...  
10 }
```

Листинг 50: Передача объекта как параметра функции

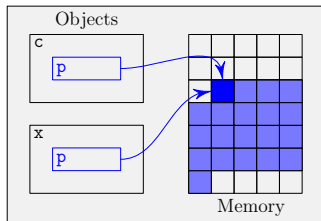


Рис. 2: Состояние памяти после копирования объекта c

```
1 class Class1 {
2     int *p;
3 public:
4     Class1() {
5         p = new int[20];
6     }
7     Class1(const Class1 & a) {
8         p = new int[20];
9         for(int i = 0; i < 20; i++)
10            p[i] = a.p[i];
11    }
12    ~Class1() {
13        delete [] p;
14    }
15 };
```

Листинг 51: Описание конструктора копирования в классе Class1

# Динамическая память и копирование объектов

Состояние памяти после вызова конструктора копирования

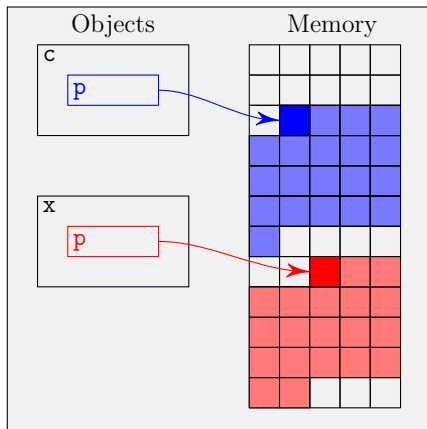


Рис. 3: Состояние памяти после корректного копирования объекта *c*

Компилятор C++ неявно генерирует два вида конструкторов:

- конструктор копирования;
- конструктор по умолчанию.

### Замечание 6

*Отсутствие явного описания конструктора копирования не означает невозможности создания копии объекта! Для запрета возможности копирования объекта необходимо описать конструктор копирования явно в приватной части класса.*

### Замечание 7

*Конструктор по умолчанию генерируется только в том случае, если программист не описал вообще ни одного конструктора.*

# Значения параметров по умолчанию

## Значения параметров функций по умолчанию

```
1 void f(int a = 3, const char * b = "string", int c = 5);
```

### Листинг 52: Прототип функции

```
1 f(7, "name", 10);  
2 f(7, "name");           // the same as f(7, "name", 5)  
3 f(7);                   // the same as f(7, "string", 5)  
4 f();                     // the same as f(3, "string", 5)
```

### Листинг 53: Примеры вызова

#### Замечание 8

*Все параметры функции, следующие в списке параметров за первым, имеющим значение по умолчанию, также должны иметь значение по умолчанию.*

# Значения параметров по умолчанию

## Примеры прототипов функций со значениями параметров по умолчанию

---

```
1 void f(int a, int b, int c);
2 void f(int a = 0, int b = 10, int c = 20);
3 void f(int a = 0, int b, int c = 20);
4 void f(int a = 0, int b = 10, int c);
5 void f(int a, int b = 10, int c = 20);
6 void f(int a = 0, int b = 10, int c);
7 void f(int a, int b, int c = 20);
8 void f(int a = 0, int b, int c);
9 void f(int a, int b = 10, int c);
```

---

Листинг 54: Корректные и некорректные примеры

### Замечание 9

*Конструктор, допускающий вызов с разным количеством и типами параметров, может быть воспринят компилятором в разных ролях:*

- 1. без параметров — конструктор по умолчанию;*
- 2. с одним параметром, имеющим тип, отличный от описываемого — конструктор преобразования;*
- 3. с одним параметром, имеющим тип “ссылка на описываемый класс или структуру” — конструктор копирования.*

```
1 Complex(double a_re = 0, double a_im = 0) {  
2     re = a_re;  
3     im = a_im;  
4 }
```

Листинг 55: Конструктор класса Complex



```
1 class C1 {
2 // ...
3 public:
4     C1();
5     void f(int a, int b);
6     int g(const char * str) const;
7 };
```

---

Листинг 56: Пример заголовка класса

---

```
1 C1::C() {
2     // ...
3 }
4
5 void C1::f(int a, int b) {
6     // ...
7 }
8
9 void C1::g(const char * str) const {
10    // ...
11 }
```

---

Листинг 57: Пример реализации класса

# Инициализация членов класса в конструкторе

## Проблема и её решение

---

```
1 class A {
2     // ...
3 public:
4     A(int x, int y) {
5         // ...
6     }
7 };
8
9 class B {
10     A a;
11 public:
12     B();
13 };
```

---

Листинг 58: Проблема

# Инициализация членов класса в конструкторе

## Проблема и её решение

---

```
1 class A {
2     // ...
3 public:
4     A(int x, int y) {
5         // ...
6     }
7 };
8
9 class B {
10     A a;
11 public:
12     B();
13 };
```

---

### Листинг 58: Проблема

---

```
1 B::B() : a(2, 3) {
2     // ...
3 }
```

---

### Листинг 59: Решение

# Инициализация членов класса в конструкторе

## Дополнительные замечания

### Замечание 10

*Данным способом можно инициализировать любые поля, а не только поля типа класс.*

```
1 class Complex {
2     double re, im;
3 public:
4     Complex(double a_re, double a_im) : re(a_re), im(a_im) {}
5     Complex(double a_re) : re(a_re), im(0) {}
6     Complex() : re(0), im(0) {}
7     // ...
8 };
```

**Листинг 60:** Пример инициализации полей класса Complex в конструкторах

# Инициализация членов класса в конструкторе

## Дополнительные замечания

### Замечание 10

*Данным способом можно инициализировать любые поля, а не только поля типа класс.*

```
1 class Complex {
2     double re, im;
3 public:
4     Complex(double a_re, double a_im) : re(a_re), im(a_im) {}
5     Complex(double a_re) : re(a_re), im(0) {}
6     Complex() : re(0), im(0) {}
7     // ...
8 };
```

**Листинг 60:** Пример инициализации полей класса Complex в конструкторах

### Замечание 11

*Инициализаторы полей должны следовать после двоеточия в том же порядке, в котором данные поля описаны в классе.*

# Перегрузка операций функциями вне класса

## Проблема и возможное решение

---

```
1 Complex x, y;  
2 // ...  
3 y = x + 0.5;  
4 y = 0.5 + x;
```

---

Листинг 61: Проблема при использовании метода  
Complex::operator+()

# Перегрузка операций функциями вне класса

## Проблема и возможное решение

---

```
1 Complex x, y;  
2 // ...  
3 y = x + 0.5;  
4 y = 0.5 + x;
```

---

Листинг 61: Проблема при использовании метода  
Complex::operator+()

---

```
1 Complex operator+(const Complex& a, const Complex& b) {  
2     return Complex(a.get_re() + b.get_re(), a.get_im() + b.  
3         get_im());  
3 }
```

---

Листинг 62: Решение проблемы

# Дружественные функции и классы

## Определение и примеры

---

```
1 class Complex {
2     friend Complex operator+(const Complex&, const Complex&);
3     // ...
4 };
5
6 Complex operator+(const Complex& a, const Complex& b) {
7     return Complex(a.re + b.re, a.im + b.im);
8 }
```

---

Листинг 63: Пример дружественной функции

---

```
1 class A {
2     friend class B;
3     // ...
4 };
```

---

Листинг 64: Пример дружественного класса



### Определение 5

*Статическое поле класса — это переменная, входящая в область видимости класса, время жизни которой совпадает с временем выполнения программы.*

```
1 class Cls {  
2     // ...  
3     static int the_static_field;  
4     // ...  
5 };
```

Листинг 65: Декларация статического поля

```
1 int Cls::the_static_field = 0;
```

Листинг 66: Определение (и инициализация) статического поля

### Замечание 12

*Если описание класса вынесено в заголовочный файл, то определения статических полей обязательно необходимо поместить в файл реализации одного из модулей.*

---

```
1 Cls a;  
2 a.the_static_field = 10;  
3 Cls::the_static_field = 10;
```

---

**Листинг 67:** Обращение к открытым статическим полям

### Определение 6

*Статический метод — это метод, который, являясь методом класса и имея доступ к закрытым деталям его реализации, при этом вызывается независимо от объектов класса.*

```
1 class Cls {  
2     static int TheStaticMethod(int a, int b);  
3 };
```

Листинг 68: Декларация статического метода

```
1 Cls c;  
2 c.TheStaticMethod(5, 10);  
3 Cls::TheStaticMethod(5, 10);
```

Листинг 69: Обращение к статическому методу

### Замечание 13

*Так как статический метод может быть вызван без объекта, то у него отсутствует неявный параметр `this`. Это означает, что статический метод не может обращаться к полям объекта и вызывать нестатические методы.*

### Исключение

Статический метод может получить доступ к объекту своего класса в следующих случаях:

- при передаче объекта через один из параметров;
- при получении доступа к объекту через глобальные переменные;
- статическая функция может создать объект сама.

# А как с этим в Python?

```
1 from math import sqrt
2
3 class Complex:
4     def __init__(self, re=0, im=0):
5         self.re = re
6         self.im = im
7
8     def modulo(self):
9         return sqrt(self.re*self.re + self.im*self.im)
10
11    def __str__(self):
12        return "{0},{1}".format(self.re, self.im)
13
14    def __add__(self, other):
15        re = self.re + other.re
16        im = self.im + other.im
17        return Complex(re, im)
18
19 z1 = Complex(1, 2)
20 z2 = Complex(2, 3)
21 #z2 = 2
22
23 print(z1+z2)
24 print((z1+z2).modulo())
```

**Листинг 70:** Несложный пример работы с комплексными числами с позиций ООП на языке Python

# Контрольные вопросы и задания

1. Изучите самостоятельно механизм декорирования имён (name mangling) в C++. В чём его необходимость и как он связан с перегрузкой имён функций? Какие при этом потенциальные проблемы могут возникнуть при трансляции модульных программ? Как эти проблемы решают?
2. Разберитесь с особенностями переопределения в C++ следующих операций: присваивания =, += и т. п.; индексирования []; инкремента ++ и декремента --; косвенного выбора ->; вызова функций (); преобразования типа (type). Что такое функторы?
3. Найдите и самостоятельно изучите информацию об inline-функциях в C++.
4. Изучите исходный текст, приведённый в Листинге 70. Что вы можете сказать о принципе его работы? Что изменится (и почему?), если закомментировать строку 20 и раскомментировать строку 21?
5. Прочитайте официальную документацию о ссылках в языке Python. Что такое переменная в Python? Имеются ли в Python указатели?
6. Разберитесь с перегрузкой операций в языке Python.
7. Ознакомьтесь с переопределением (overriding) и перегрузкой (overloading) методов в Python. Как эти понятия соотносятся со статическим и динамическим полиморфизмом?