

Технология программирования

П.А. Макаров



СЫКТЫВКАРСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ ПИТИРИМА СОРОКИНА

ОПОРНЫЙ ВУЗ РЕГИОНА

2021–2022 учебный год,
осенний семестр
(версия от 16 октября 2021 г.)

- 1 Введение в ООП и С++
- 2 Абстрактные типы данных и инкапсуляция
- 3 Обработка исключений

- 1 Страуструп Б. Язык программирования С++;
- 2 Подбельский В.В. Язык Си++;
- 3 Столяров А.В. Введение в язык Си++;
- 4 Столяров А.В. Программирование: введение в профессию;
- 5 Брайант Р., О'Халларон Д. Компьютерные системы: архитектура и программирование. Взгляд программиста;
- 6 Кнут Д. Искусство программирования;
- 7 Кормен Т.Х., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы. Построение и анализ;
- 8 Вирт Н. Алгоритмы и структуры данных.

Определение 1

Парадигма программирования — комплекс концепций, используемых программистом при проектировании программы.

Примеры парадигм:

- императивная,
- процедурная,
- логическая,
- функциональная,
- ...

Основные понятия

- объект (свойства и методы),
- события,
- интерфейс,
- класс.

Главные идеи ООП

- полиморфизм,
- инкапсуляция,
- наследование.

```
1 int try;  
2 long new;
```

Листинг 1 : Ключевые слова

```
1 int try;  
2 long new;
```

Листинг 1 : Ключевые слова

```
1 struct mystruct {  
2     int a, b;  
3 };
```

Листинг 2 : Структуры в C++

```
1 typedef struct mystruct {  
2     int a, b;  
3 } mystruct;
```

Листинг 3 : Структуры в C

```
1 int try;  
2 long new;
```

Листинг 1 : Ключевые слова

```
1 struct mystruct {  
2     int a, b;  
3 };
```

Листинг 2 : Структуры в C++

```
1 typedef struct mystruct {  
2     int a, b;  
3 } mystruct;
```

Листинг 3 : Структуры в C

```
1 struct point {  
2     double x, y, z;  
3 };  
4 point A, B, C; // struct point A, B, C;
```

Листинг 4 : Использование структур в C++

- 1 Введение в ООП и С++
- 2 Абстрактные типы данных и инкапсуляция
- 3 Обработка исключений

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

Листинг 5 : Пример структуры комплексного числа

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

Листинг 5 : Пример структуры комплексного числа

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

Листинг 6 : Функция вычисления модуля в C

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

Листинг 5 : Пример структуры комплексного числа

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

Листинг 6 : 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 7 : Использование методов

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

Листинг 8 : Применение ключевого слова `this`

Методы, объекты и защита

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 9 : Применение ключевых слов `public` и `private`

```
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 10 : Пример метода, инициализирующего объект

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

Листинг 11 : Применение метода, инициализирующего объект

```
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 12 : Описание конструктора объекта

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

Листинг 13 : Использование конструктора


```
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 12 : Описание конструктора объекта

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

Листинг 13 : Использование конструктора

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

Листинг 14 : Анонимные объекты

```
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 15 : Реализация комплексного числа в виде класса

Перегрузка символов стандартных операций

Пример: `operator.cpp`

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

Листинг 16 : Перегруженные операции (вариант 1)

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

Листинг 17 : Перегруженные операции (вариант 2)

Перегрузка символов стандартных операций

Пример: `operator.cpp`

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

Листинг 16 : Перегруженные операции (вариант 1)

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

Листинг 17 : Перегруженные операции (вариант 2)

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

Листинг 18 : Перегруженные операции (реалистичный вариант)

Исключения

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

Перегрузка имён функций

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 19 : Перегрузка имён функций

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

Листинг 20 : Использование перегруженных функций

Замечание 1

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 21 : Пример определения перегруженных функций, приводящий к UB

В чем подвох?

Замечание 1

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 21 : Пример определения перегруженных функций, приводящий к UB

В чем подвох?

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

Листинг 22 : Вызовы функций, приводящие к UB

Вернёмся к Листингу 15.

```
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 };
```

Вернёмся к Листингу 15.

```
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 23 : Ошибка определения объекта

```
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 24 : Перегрузка конструкторов

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

Листинг 25 : Примеры использования

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

Листинг 26 : Простейший пример преобразования типов

(int) → (float) → (double)

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

Листинг 26 : Простейший пример преобразования типов

(int) → (float) → (double)

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

Листинг 27 : Второй пример преобразования типов

`(type A) → (type B)`

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

(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 28 : 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 29 : Примеры использования конструктора преобразования

Определение 1

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 30 : Простейший пример на указатели и ссылки

```
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 31 : Реализация с помощью указателей

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

Листинг 32 : Вызов функции


```
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 33 : Реализация с помощью ссылок

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

Листинг 34 : Вызов функции

```
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 35 : Применение ссылки в качестве возвращаемого значения

Определение 1

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 36 : Пример

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

Модификатор const

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 37 : 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 38 : Константные указатели

Модификатор const

Константные ссылки

```
1 int i;
2 const int &r = i;    // p - pointer 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 39 : Константные ссылки

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

Листинг 40 : Применение константных ссылок (нужно сравнить с Примером operator.cpp)

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 41 : Синтаксис описания константного метода

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

Листинг 42 : Пример вызова метода константного объекта

Замечание 1

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

Определение 1

Деструктор — это метод класса, 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 43 : Класс File, инкапсулирующий дескриптор файла

Операции работы с динамической памятью

Операции `new` и `delete`

- 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 44 : Пример использования операций `new` и `delete`

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

Листинг 45 : Векторная форма операций `new` и `delete`

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

Листинг 46 : Пример класса, использующего динамический массив

Конструктор копирования

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 47 : Передача объекта как параметра функции

Конструктор копирования

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 47 : Передача объекта как параметра функции

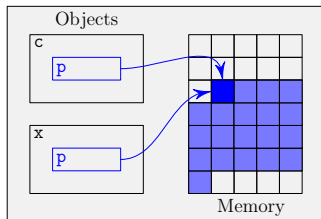


Рис. 1 : Состояние памяти после копирования объекта с

```
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 48 : Описание конструктора копирования в классе Class1

Конструктор копирования

Состояние памяти после вызова конструктора копирования

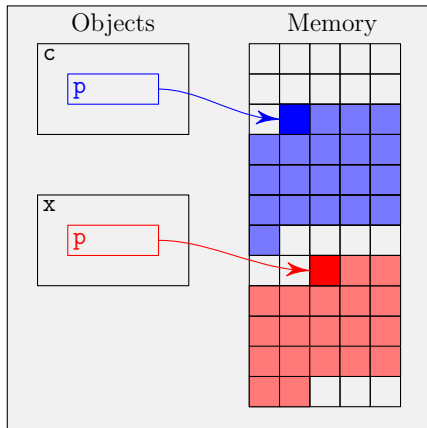


Рис. 2 : Состояние памяти после корректного копирования объекта с

Временные и анонимные объекты

Примеры и свойства временных и анонимных объектов

```
1 Complex x, y, z, Imag_1(0,1);  
2 y = x * Image_1;
```

Листинг 49 : Пример

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

Листинг 50 : Пример анонимного объекта

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

Листинг 51 : Пример временного объекта

Свойства временных и анонимных объектов:

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

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 1

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 2

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

Неявные конструкторы

Конструкторы, генерируемые компилятором неявно

Компилятор C++ неявно генерирует два вида конструкторов:

- конструктор копирования;
- конструктор по умолчанию.

Замечание 1

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

```
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

Постановка проблемы 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

Постановка проблемы 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

Замечание 1

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 1

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 2

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

```
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 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.  
   get_im());  
3 }
```

Листинг 62 : Решение проблемы

```
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 1

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::static int the_static_field = 0;
```

Листинг 66 : Определение (и инициализация) статического поля

Замечание 1

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 2

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 2

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.

- 1 Введение в ООП и C++
- 2 Абстрактные типы данных и инкапсуляция
- 3 Обработка исключений

makarovpa.ru/cpp/17/index.html