Программирование

Лекция 4. Многомерные массивы. Структуры и классы

Использование динамической памяти

- Одно из естественных применений динамической памяти – создание динамических массивов, т.е. массивов, размер которых неизвестен во время компиляции, в том числе многомерных массивов.

Многомерные встроенные массивы

- C++ позволяет определять многомерные массивы:

```
int m2d[2][3] = { {1, 2, 3}, {4, 5, 6} };
for( size_t i = 0; i != 2; ++i ) {
    for( size_t j = 0; j != 3; ++j ) {
        cout << m2d[i][j] << ' ';
    }
    cout << endl;
}
```

Матрица

- Элементы m2d располагаются в памяти “по строчкам”.
- Размерность массивов может быть любой, но на практике редко используют массивы размерности > 4.

```
int m4d[2][3][4][5] = {};
```

Динамические массивы

- Для выделения одномерных динамических массивов обычно используется оператор `new []`.

```
int * m1d = new int[100];
```

- Какой тип должен быть у указателя на двумерный динамический массив?
  - Пусть `m` — указатель на двумерный массив типа `int`.
  - Значит `m[i][j]` имеет тип `int` (точнее `int &`).
  - `m[i][j]` ⇔ `*(m[i] + j)`, т.е. тип `m[i]` — `int *`.
  - аналогично, `m[i]` ⇔ `*(m + i)`, т.е. тип `m` — `int **`.
- Чему соответствует значение `m[i]`?

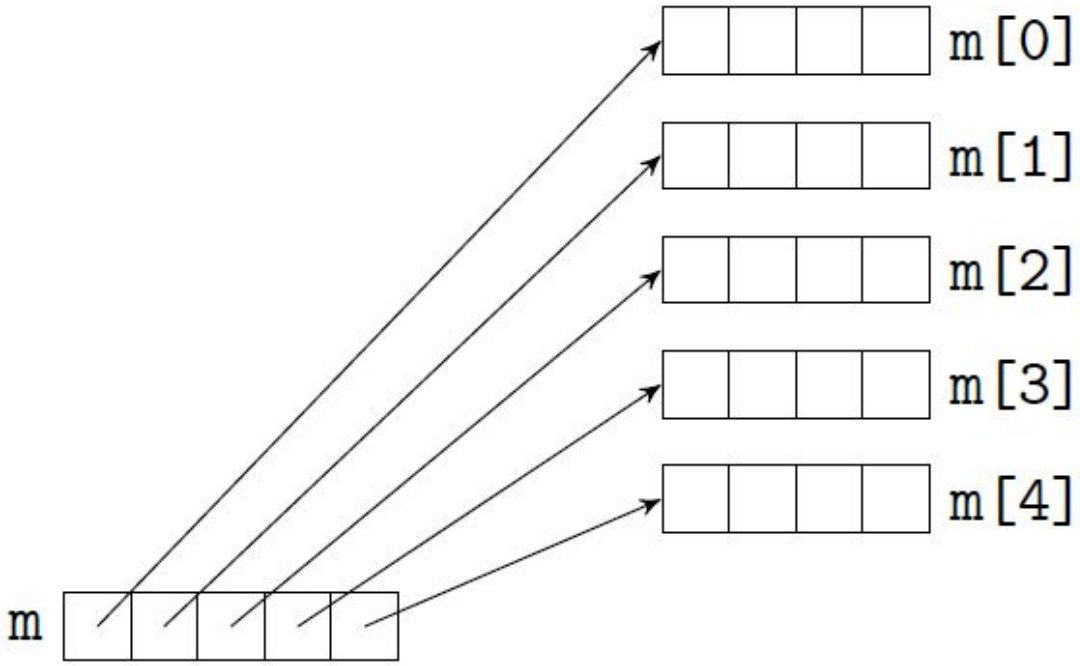
  Это адрес строки с номером `i`.

  m[i] = *(m+i)
- Чему соответствует значение `m`?

  Это адрес массива с указателями на строки.

Динамические массивы

Давайте рассмотрим создание массива 5×4.




```
int ** m = new int * [5]; // массив указателей на строки int
for (size_t i = 0; i != 5; ++i)
    m[i] = new int[4];
```

В каждой строке может быть разное количество элементов

Динамические массивы

Выделение и освобождение двумерного массива размера $a \times b$.

```
int ** create_array2d(size_t a, size_t b) {
    int ** m = new int *[a];
    for (size_t i = 0; i != a; ++i)
        m[i] = new int[b];
    return m;
}
// удаление массива
void free_array2d(int ** m, size_t a, size_t b) {
    for (size_t i = 0; i != a; ++i)
        delete [] m[i]; // удаляем строчки
    delete [] m; // удаляем массив указателей на строки
}
```

size_t – спец. тип для указания размера массива

Данная переменная не используется

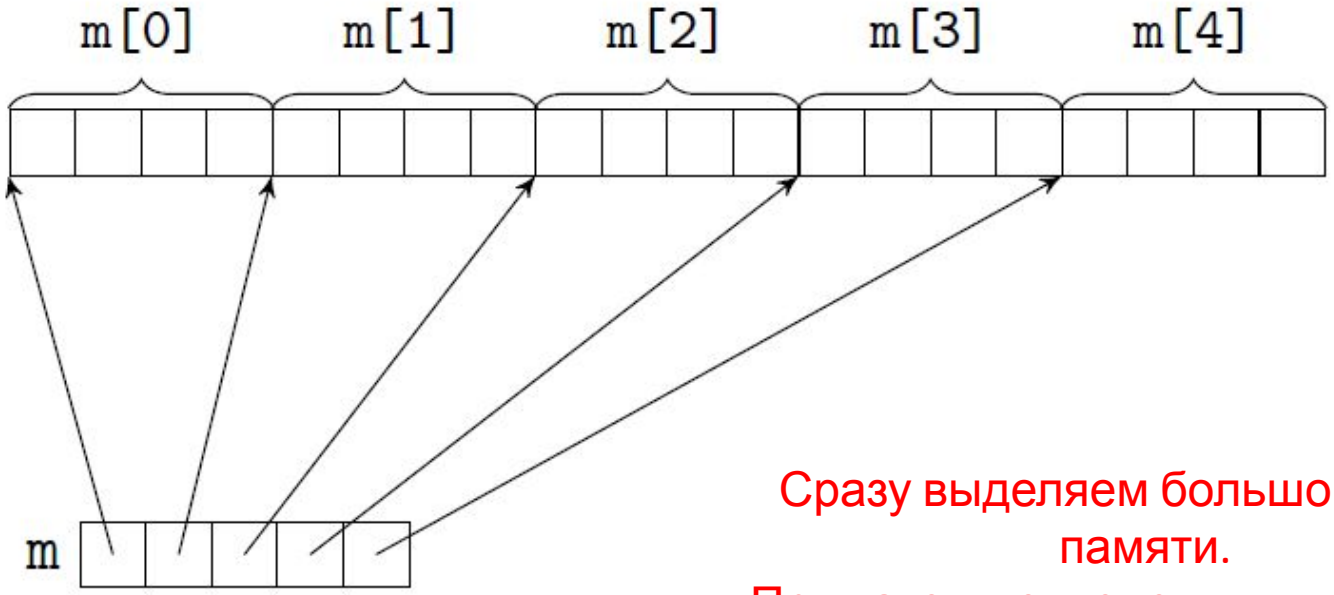
При создании массива оператор new вызывается $(a + 1)$ раз.

Недостаток – фрагментация памяти

Двумерные массивы: эффективная схема

- Будем выделять строчки массива не по отдельности, а все сразу.

Рассмотрим эффективное создание массива 5×4.



Сразу выделяем большой объем памяти.

При таком подходе new вызываем 2 раза

```
int ** m = new int * [5];
m[0] = new int[5 * 4];
for (size_t i = 1; i != 5; ++i)
    m[i] = m[i - 1] + 4;
```

Указатель на 0 ячейку

Указатели на 4, 8, 12, .. ячейки

Двумерные массивы: эффективная схема

Эффективное выделение и освобождение двумерного массива размера $a \times b$.

```
int ** create_array2d(size_t a, size_t b) {
    int ** m = new int *[a];
    m[0] = new int[a * b];
    for (size_t i = 1; i != a; ++i)
        m[i] = m[i - 1] + b;
    return m;
}

void free_array2d(int ** m, size_t a, size_t b) {
    delete [] m[0];
    delete [] m;
}
```

Возвращаем указатель на указатель на int

Освободим большой массив

Удалить массив указателей на строки

При создании массива оператор new вызывается 2 раза.

Структуры

- Структуры – это способ объединить несколько переменных в одну.

Какая должна быть сигнатура у функции, которая вычисляет длину отрезка на плоскости?

```
double length(double x1, double y1,
              double x2, double y2);
```

А сигнатура функции, проверяющей пересечение отрезков?

```
bool intersects(double x11, double y11,
                double x12, double y12,
                double x21, double y21,
                double x22, double y22,
                double * xi, double * yi);
```

Координаты точки пересечения xi, yi

Координаты точек являются логически связанными данными, которые всегда передаются вместе.

Аналогично связанны координаты точек отрезка.

Структуры

Структуры — это способ синтаксически (и физически) сгруппировать логически связанные данные.

```
struct Point {
    double x;
    double y;
};    // не забываем «;»

struct Segment {        // Структура
    Point p1;              Отрезок
    Point p2;
};

double length(Segment s);     // Функции упрощаются

bool intersects(Segment s1,   // 3 аргумента вместо 10-
                Segment s2, Point *p);   ти
```

Работа со структурами

Доступ к полям структуры осуществляется через оператор '.':

```
#include <cmath>   // мат. библиотека

double length(Segment s) {  // функция определения длины отрезка
    double dx = s.p1.x - s.p2.x;
    double dy = s.p1.y - s.p2.y;
    return sqrt(dx * dx + dy * dy);
}
```

Для указателей на структуры используется оператор '->'.

```
double length(Segment * s) {    // чтобы не копировать всю структуру
    double dx = s->p1.x - s->p2.x;
    double dy = s->p1.y - s->p2.y;
    return sqrt(dx * dx + dy * dy);   // p1 – уже не указатель, поэтому оператор "."
}
```

Можно использовать разыменование: (*s).p1.x ~ s->p1.x

Инициализация структур

Поля структур можно инициализировать подобно массивам:

```
Point p1   = { 0.4, 1.4 };
Point p2   = { 1.2, 6.3 };
Segment s  = { p1,  p2  };
```

x y

Структуры могут хранить переменные разных типов.

```
struct IntArray2D {
    size_t a;// количество строк
    size_t b;// количество столбцов
    int ** data;      // указатель на
                      двумерный массив
};
```

Отличие от массивов

// структура хранит информацию о двумерном массиве

```
IntArray2D a = {n, m, create_array2d(n, m)};
```

// инициализация переменной

Функция выделения двумерного массива

Задача

В коде определена следующая структура:

```
struct ivector3d
{
	int array[3];
};
```

И определена следующая функция:

```
void scale(ivector3d *v, int k)
{
	for (int i = 0; i != 3; ++i)
		v->array[i] *= k;
}
```

Пусть у вас есть экземпляр *iv3d* структуры *ivector3d*.

Изначально массив *array* экземпляра iv3d заполнен единицами, и вы вызываете функцию *scale* следующим образом:

```
scale(&iv3d, 2);
```

Какова будет сумма элементов массива *array* внутри *iv3d* по завершению функции?

Задача

Ответ к предыдущей задаче: 6.

У вас есть та же самая структура *ivector3d* и экземпляр *iv3d* этой структуры, массив *array* которого заполнен единицами. Но теперь функция *scale* определена следующим образом:

```
void scale(ivector3d v, int k)
{
    for (int i = 0; i != 3; ++i)
        v.array[i] *= k;
}
```

И вы вызываете функцию *scale* следующим образом:

```
scale(iv3d, 2);
```

Вопрос тот же самый: какова будет сумма элементов внутри экземпляра *iv3d* после завершения функции?

- Ответ к предыдущей задаче: 3.

Методы

Важное отличие структур на языке C++ от структур в C – возможность определения методов.

Метод — это функция, определённая внутри структуры.

```
struct Segment {
    Point p1;
    Point p2;
    double length() {
        double dx = p1.x - p2.x;
        double dy = p1.y - p2.y;
        return sqrt(dx * dx + dy * dy);
    }
};
int main() {
    Segment s = { { 0.4, 1.4 }, { 1.2, 6.3 } };
    cout << s.length() << endl;
    return 0;
}
```

Отличие от функций – внутри методов есть доступ к полям структуры

Инициализация структуры

Вызов метода

Оператор обращения к полям структуры

Что выведет программа?

- Ответ к предыдущей задаче: $\sqrt{24.65}$.

Методы реализованы как функции с неявным параметром this, который указывает на текущий экземпляр структуры.

```
struct Point
{
    double x;
    double y;

    void shift(/* Point * this, */
               double x, double y) {
        this->x += x;
        this->y += y;
    }
};
```

Точка на плоскости

Сдвигает точку

Имена аргументов совпадают с именами полей структуры

Поле текущего экземпляра

Методы: объявление и определение

(в заголовочный файл)

(в .cpp-файл)

Методы можно разделять на объявление и определение:

```
struct Point
{
    double x;
    double y;

    void shift(double x, double y);
};
```

объявление

```
void Point::shift(double x, double y)
{
    this->x += x;
    this->y += y;
}
```

реализация

Методы

Использование методов позволяет объединить данные и функции для работы с ними.

```
struct IntArray2D {
    int & get(size_t i, size_t j) {
        return data[i * b + j];
    }
    size_t a;
    size_t b;
    int * data;
};
```

Данные о двумерном массиве

Метод переводит адреса из двумерного массива в соответствующие адреса для одномерного массива

Размерность массива a×b

Указатель на данные массива

Одномерный массив

```
IntArray2D m = foo();
for (size_t i = 0; i != m.a; ++i)
    for (size_t j = 0; j != m.b; ++j)
        if (m.get(i, j) < 0) m.get(i,j) = 0;
```

Функция возвращает двумерный массив

В данном случае обнуляем отрицательные элементы массива

Пример

point.cpp 〉 No Selection

```cpp
#include "point.hpp"

void Point::shift(double x, double y)
{
    this->x += x;
    this->y += y;
}
```

point.hpp 〉 No Selection

```cpp
#pragma once

struct Point
{
    double x;
    double y;

    void shift(double x, double y);
};
```

Конструкторы

Конструкторы — это методы для инициализации структур.

```
struct Point {
    Point() {
        x = y = 0;
    }
    Point(double x, double y) {
        this->x = x;
        this->y = y;
    }
    double x;
    double y;
};
```

Определить конструктор – значит определить функцию с именем, совпадающим с именем Структуры и не возвращающую никакого значения (но не void)

2 аргумента

this – так как имена совпадают

```
Point p1; = {0, 0}
Point p2(3,7);
```

Конструктор вызывается автоматически при определении переменных

Список инициализации

- Чаще всего в конструкторах происходит инициализация полей структур.

Список инициализации позволяет проинициализировать поля до входа в конструктор.

```
struct Point {
    Point() : x(0), y(0)
    {}
    Point(double x, double y) : x(x), y(y)
    {}

    double x;
    double y;
};
```

До входа в тело конструктора инициализируем x = 0, y = 0

Формальные параметры

Сначала инициализируется x, потом y. **Point (): y(0), x(y)**

Инициализации полей в списке инициализации происходит в *порядке объявления полей* в структуре.

Значения по умолчанию

а также методы и конструкторы

- Функции могут иметь значения параметров *по умолчанию*.
- Значения параметров по умолчанию нужно указывать в *объявлении функции*. (т.е. в заголовочный файл)

```
struct Point {
    Point(double x = 0, double y = 0)
        : x(x), y(y)
    {}
    double x;
    double y;
};
```

Такой конструктор будет работать как конструктор с 0, 1 или 2 параметрами.

```
    Point p1;    = {0, 0}
    Point p2(2);    = {2, 0}
    Point p3(3,4);    = {3, 4}
```

Конструкторы от одного параметра

Такие конструкторы задают преобразование от значения типа аргумента к значению типа Структура.

Конструкторы от одного параметра задают *неявное* пользовательское преобразование:

```
struct Segment {
    Segment() {}
    Segment(double length)
        : p2(length, 0)
    {}
    Point p1;
    Point p2;
};
```

Пустое тело, однако, Point имеет конструктор

```
    Segment s1;
    Segment s2(10);
    Segment s3 = 20;
```

= {(0, 0), (0, 0)}

= {(0, 0), (10, 0)}

= {(0, 0), (20, 0)}

- Неявное преобразование

Конструкторы от одного параметра

Для того, чтобы запретить *неявное* пользовательское преобразование, используется ключевое слово `explicit`.

```
struct Segment {
    Segment() {}
    explicit Segment(double length)
        : p2(length, 0)
    {}
    Point p1;
    Point p2;
};
```

Используется только для конструкторов одного параметра

```
    Segment s1;
    Segment s2(10);
    Segment s3 = 20; // error
```

Можно вызвать явно

Неявно нельзя. Защита от неявных преобразований

Конструкторы от одного параметра

Неявное пользовательское преобразование, задаётся также конструкторами, которые могут принимать один параметр.

```
struct Point {
    explicit Point(double x = 0, double y = 0)
        : x(x), y(y)
    {}
    double x;
    double y;
};
```

Такой конструктор может быть вызван как конструктор с одним параметром. Поэтому может возникнуть неявное преобразование. Explicit защитит от неявных преобразований

```
    Point p1;
    Point p2(2);
    Point p3(3,4);
    Point p4 = 5; // error
```

Конструктор по умолчанию

Если у структуры нет конструкторов, то конструктор без параметров, *конструктор по умолчанию*, генерируется компилятором.

```
struct Segment {
    Segment(Point p1, Point p2)
        : p1(p1), p2(p2)
    {}
    Point p1;
    Point p2;
};
```

Для такой структуры невозможно определить переменную, не указав аргументы конструктора

```
    Segment s1; // error
    Segment s2(Point(), Point(2,1));
```

= {(0, 0), (2, 1)}

Деструктор

Деструктор — это метод, который вызывается при удалении структуры, генерируется компилятором.

```
struct IntArray {
    explicit IntArray(size_t size)
        : size(size)
        , data(new int[size])
    { }

    ~IntArray() {
        delete [] data;
    }

    size_t size;
    int *  data;
};
```

Однако, деструктор можно переопределить

«Тильда»

В деструкторе удалим дин. память, выделенную в

Деструктор всегда 1, он не имеет параметров

Размер массива

Указатель на динамический массив

Время жизни

Время жизни — это временной интервал между вызовами конструктора и деструктора.

```
void foo()
{
    IntArray a1(10); // создание a1
    IntArray a2(20); // создание a2
    for (size_t i = 0; i != a1.size; ++i) {
        IntArray a3(30); // создание a3
        ...
    } // удаление a3
} // удаление a2, потом a1
```

Переменная определена в теле цикла

Удаление на каждой итерации цикла

Деструктор вызывается при выходе из функции

Деструкторы переменных на стеке вызываются в обратном порядке (по отношению к порядку вызова конструкторов).

Объекты в динамической памяти

Для создания объекта в динамической памяти используется оператор new, он отвечает за вызов конструктора.

```
struct IntArray {
    explicit IntArray(size_t size);
    ~IntArray();

    size_t size;
    int *  data;
};
```

Удаление

При вызове оператора `delete` вызывается деструктор объекта.

```
// выделение памяти и создание объекта
IntArray * pa = new IntArray(10);

// вызов деструктора и освобождение памяти
delete pa;
```

Операторы `new []` и `delete []` работают аналогично

```
// выделение памяти и создание 10 объектов
// (вызывается конструктор по умолчанию)
IntArray * pa = new IntArray[10];

// вызов деструкторов и освобождение памяти
delete [] pa;
```

Модификаторы доступа

Модификаторы доступа позволяют ограничивать доступ к методам и полям класса.

```
struct IntArray {
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size])
    {}
    ~IntArray() { delete [] data_;  }

    int &  get(size_t i) { return data_[i]; }
    size_t size()        { return size_; }

private:
    size_t size_;
    int *  data_;
};
```

Никакая внешняя функция не сможет обращаться к полям private. Только методы класса

Ключевое слово class

Ключевое слово `struct` можно заменить на `class`, тогда поля и методы по умолчанию будут private.

```
class IntArray {
public:
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size])
    {}
    ~IntArray() { delete [] data_;  }

    int &  get(size_t i) { return data_[i]; }
    size_t size()        { return size_; }

private:
    size_t size_;
    int *  data_;
};
```

По умолчанию все поля класса являются private

Размер массива

Указатель на динамический массив

Публичный интерфейс

- Это набор методов, доступных внешнему пользователю класса.

```
struct IntArray {
    ...
    void resize(size_t nsize) {
        int * ndata = new int[nsize];
        size_t n = nsize > size_ ? size_ : nsize;
        for (size_t i = 0; i != n; ++i)
            ndata[i] = data_[i];
        delete [] data_;
        data_ = ndata;
        size_ = nsize;
    }
private:
    size_t size_;
    int *  data_;
};
```

Функция изменяет размер массива

Поля закрытые, т.к. иначе пользователь мог бы изменить размер массива size_, не изменив при этом сам массив data_

Абстракция

- Выделение публичного интерфейса позволяет абстрагироваться от конкретной реализации данного класса.
- Допустим, мы хотим оптимизировать класс IntArray

```
struct IntArray {
public:
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size])
    {}
    ~IntArray() { delete [] data_;  }

    int &  get(size_t i) { return data_[i]; }
    size_t size()        { return size_; }

private:
    size_t size_;
    int *  data_;
};
```

Размер массива

Указатель на динамический массив

Т.к. массив целочисленный, то можно хранить размер массива в самом массиве

Абстракция

- Внешние пользователи класса работают только с публичным интерфейсом, можно изменить реализацию класса, не меняя публичный интерфейс

```
struct IntArray {
public:
    explicit IntArray(size_t size)
        : data_(new int[size + 1])
    {
        data_[0] = size;
    }
    ~IntArray() { delete [] data_;  }

    int &  get(size_t i) { return data_[i + 1]; }
    size_t size()        { return data_[0]; }

private:
    int *  data_;
};
```

На 1 элемент больше

Оставляем только 1 поле

При этом сигнатура публичных методов не изменилась

Определение констант

Иногда возникает необходимость защитить данные от случайного или специального изменения

- Ключевое слово `const` позволяет определять типизированные константы.

```
double const pi = 3.1415926535;
int const day_seconds = 24 * 60 * 60;
// массив констант
int const days[12] = {31, 28, 31,
                      30, 31, 30,
                      31, 31, 30,
                      31, 30, 31};
```

Кол-во дней в месяцах в невесокосном году

- Попытка изменить константные данные приводит к неопределённому поведению.

```
int * may = (int *) &days[4];
*may = 30;
```

Адрес месяца «май»

Указатели и const

В C++ можно определить как константный указатель, так и указатель на константу:

```
int a = 10;
const int * p1 = &a; // указатель на константу
int const * p2 = &a; // указатель на константу
*p1 = 20; // ошибка
p2  = 0;  // OK

int * const p3 = &a; // константный указатель
*p3 = 30; // OK
p3  = 0;  // ошибка

// константный указатель на константу
int const * const p4 = &a;
*p4 = 30; // ошибка
p4  = 0;  // ошибка
```

Не можем поменять значение, но можем поменять адрес

Сможем менять значение по данному указателю (зн-е переменной a), но не сможем менять тот адрес, который хранится в данном указателе

Указатели и const

- Для того, чтобы избежать путаницы при использовании указателей с ключевым словом const, можно использовать следующее правило:

  “слово const делает неизменяемым тип слева от него”.

```
int a = 10;
int * p = &a;

// указатель на указатель на const int
int const ** p1 = &p;

// указатель на константный указатель на int
int * const * p2 = &p;

// константный указатель на указатель на int
int ** const p3 = &p;
```

p хранит адрес переменной *a*

Ключевое слово *const* сделает константным значение типа *int*

Константным станет значение *int ** (не сможем менять значение, на к-рое указывает p2, т.е. зн-е *p*)

Не сможем перенаправить p3 на другой адрес, но сможем менять значения

Книги и интернет-источники на заметку

- Дейтел Х., Дейтел П. Как программировать на C++. — Бином-Пресс, 2009. — 800 с.
- Липпман С., Лажойе Ж. Язык программирования C++. Вводный курс. — Невский Диалект, ДМК Пресс. — 1104 с.
- Липпман С., Лажойе Ж., Му Б. Язык программирования C++. Вводный курс. — Вильямс, 2007. — 4-е изд. — 896 с.
- Мейерс С. Эффективный и современный C++: 42 рекомендации по использованию C++11 и C++14. — Вильямс, 2015.
- Петров А.В. Достижима ли в C++ эффективность языка «среднего уровня»? / DEV Labs C++ 2013.
- Прата С. Язык программирования C++. Лекции и упражнения. — Вильямс, 2012. — 6-е изд. — 1248 с.: ил.
- Саттер Г. Новые сложные задачи на C++. — Вильямс, 2005. — 272 с.
- Саттер Г. Решение сложных задач на C++. — Вильямс, 2008. — 400 с.
- Саттер Г., Александреску А. Стандарты программирования на C++. — Вильямс, 2008. — 224 с.
- Седжвик Р. Алгоритмы на C++. — Вильямс, 2011. — 1056 с.

Книги и интернет-источники на заметку

- Страуструп Б. Программирование. Принципы и практика использования C++. — Вильямс, 2011. — 1248 с.
- Страуструп Б. Язык программирования C++. — Бином, 2011. — 1136 с.
- Шилдт Г. C++: базовый курс. — Вильямс, 2008. — 624 с.
- Шилдт Г. C++. Методики программирования Шилдта. — Вильямс, 2009. — 480 с.
- Шилдт Г. Полный справочник по C++. — Вильямс, 2007. — 800 с.
- Abrahams, D., Gurtovoy, A. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond* (Addison Wesley Professional, 2004).