// Типы данных, переменные, операторы

```
#include <iostream>

// Использование пространства имён std на глобальном уровне не является
хорошим тоном
// из-за возможных конфликтов с внутренними именами программы.
// ВАЖНО!!! Конструкция using namespace <name> недопустима на глобальном
уровне
// в любых заголовочных файлах
// using namespace std;

int main()
{
    int a = 5;
    int b = 2;

    int n = 0;
    double c = 0.0;

    // Арифметические операции
    std::cout << "----- Arithmetic operations -----\n";
    // манипулятор endl и символ '\n' переводят курсор на новую строку
    std::cout << "a=5 - int, b=2 - int, n - int\n";
    n = a + b;
    std::cout << "n=a+b => n=" << n << std::endl;
    n = a - b;
    std::cout << "n=a-b => n=" << n << '\n';
    n = a * b;
    std::cout << "n=a*b => n=" << n << '\n';
    n = a % b; //остаток от деления
    std::cout << "n=a%b => n=" << n << '\n';

    // с делением несколько сложнее
    // в операции "/" участвуют две переменные типа int => результат
получается типа int,
    // т.е. 2,5 округляется до 2 (дробная часть просто отбрасывается);
    // то, что c имеет тип double - не важно
    c = a / b;
    std::cout << "c=a/b => c=" << c << '\n';

    std::cout << '\n';

    b = 6;
    c = a / b; // дробная часть отбрасывается, а не округляется, т.е даже 5/6
превращаются в 0
    std::cout << "a=5 - int, b=6 - int, c - double\n"
        << "c=a/b => c=" << c << '\n';
```

```
    // чтобы сохранить точность используем явное преобразование типа одной из переменных (a),
    // причём к типу double переменная a приводится не навсегда, а только в этом выражении
    c = static_cast<double>(a) / b;
    std::cout << "c=static_cast<double>(a)/b => c=" << c << '\n';

    // то же самое в стиле языка C ("старый" способ, НЕ РЕКОМЕНДУЕТСЯ ИСПОЛЬЗОВАТЬ!!!)
    c = (double)a / b;
    std::cout << "c=(double)a/b => c=" << c << '\n';

    // точно так же с числами вместо переменных, т.к. целочисленные литералы имеют тот же тип int
    c = 5 / 6;
    std::cout << "c=5/6 => c=" << c << '\n';

    c = static_cast<double>(5) / 6;
    std::cout << "c=static_cast< double>(5)/6 => c=" << c << '\n';

    // в случае с литералами лучше использовать литералы типа double (5.0, 6.0)
    c = 5.0 / 6.0;
    std::cout << "c=5.0/6.0 => c=" << c << '\n';

    std::cout << '\n';

    std::cout << "----- Operator sizeof and conversion between double and int -----\n";
    // проверим размеры результатов выражений (описание sizeof см. в конце файла или в интернете)
    std::cout << "11/2=" << 11 / 2 << ", sizeof(11/2)=" << sizeof(11 / 2) << '\n';
    std::cout << "11.0/2=" << 11.0 / 2 << ", sizeof(11.0/2)=" << sizeof(11.0 / 2) << '\n';

    std::cout << '\n';

    double d = 3.3;
    std::cout << "a=5 - int, b=6 - int, d=3.3 - double, c - double\n";
    c = d * a / b;
    // здесь происходит неявное преобразование типов (без явного указания),
    // т.к. первой выполняется операция "*" (слева направо), то результат верный
    std::cout << "c=d*a/b => c=" << c << '\n';

    c = a / b * d;
    // т.к. первой выполняется операция "/" (слева направо), то 5/6=0, 0*3.3=0
    // от перестановки множителей результат меняется :)
```

```
    std::cout << "c=a/b*d => c=" << c << '\n';
    std::cout << '\n';

    c = 2.4;
    d = 3.3;
    n = c * d; // результат c*d имеет тип double, но приводится к int, т.к. n
переменная типа int
    std::cout << "c=2.4 - double, d=3.3 - double, n - int\n"
        << "n=c*d => n=" << n << ", c*d=" << c * d << '\n';

    std::cout << '\n';

    std::cout << "----- Operator priority -----\n";
    // порядок выполнения операций "+", "-", "*" и "/" правильный, как в
математике,
    // скобки () меняют приоритет
    std::cout << "b - int" << '\n';
    b = 2 + 2 * 2;
    std::cout << "b=2+2*2 => b=" << b << '\n';

    b = 2 * 2 + 2;
    std::cout << "b=2*2+2 => b=" << b << '\n';

    b = (2 + 2) * 2;
    std::cout << "b=(2+2)*2 => b=" << b << '\n';

    std::cout << '\n';

    std::cout << "----- Assignment Operators -----\n";
    // Сокращённая запись арифметических операторов +=, -=, *=, /=, %=
    std::cout << "a=5 - int, c=5 - double\n";
    a = 5;
    a = a + 2; // увеличиваем значение переменной a на 2
    // обратите внимание, оператор "=" означает не "равно", а "присвоить",
    // т.е. выражение "a = a + 2" означает не "a равно a плюс 2",
    // а "переменной a присвоить новое значение, равное текущему плюс 2"
    std::cout << "a=a+2 => a=" << a << '\n';

    a = 5;
    a += 2;    // "a += b" аналогично "a = a + b"
    std::cout << "a+=2 => a=" << a << '\n';

    a = 5;
    a -= 2;
    std::cout << "a-=2 => a=" << a << '\n';

    a = 5;
    a *= 2;
    std::cout << "a*=2 => a=" << a << '\n';
```

```
    a = 5;
    a /= 2;     // не забываем о приведении типов
    std::cout << "a/=2 => a=" << a << '\n';

    c = 5;
    c /= 2;     // поэтому ещё раз
    std::cout << "c/=2 => c=" << c << '\n';

    a = 5;
    a %= 2;
    std::cout << "a%=2 => a=" << a << '\n';

    std::cout << '\n';

    std::cout << "----- Increment and decrement -----\n";
    // Инкремент и декремент
    // операции инкрементирования (увеличения на 1) и декрементирования
(уменьшения на 1)
    // выполняются так часто, что для них сделали отдельные операторы "++" и
"--"
    a = 1;
    a++;        // аналогично a = a + 1
    std::cout << "a=1 -> a++ => a=" << a << '\n';

    a = 10;
    a--;        // аналогично a = a - 1
    std::cout << "a=10 -> a-- => a=" << a << '\n';
    // существуют префиксная и постфиксная формы операторов "++" и "--": ++a,
a++, --a, a--
    a = 10;
    b = 10 * ++a;       // в префиксной форме сначала выполняется инкремент, а
потом остальные действия
    std::cout << "a=10, b=10*++a => a=" << a << ", b=" << b << '\n';

    a = 10;
    b = 10 * a++;       // в постфиксной форме создаётся копия переменной,
затем значение переменной увеличивается,
                        // а в выражении используется значение копии
    std::cout << "a=10, b=10*a++ => a=" << a << ", b=" << b << '\n';
    // с декрементом аналогично, попробуйте сами;
    // Стоит избегать применения операторов "++" и "--" внутри сложных
выражений
    // из-за сложностей определения порядка действий,  лучше вынести
инкремент/декремент в отдельную строку

    std::cout << '\n';

    std::cout << "----- Type length -----\n";
    // Типы данных
    // операндом sizeof может быть имя типа, переменная или выражение,
```

```
    // результат - длина (в памяти) в размерах типа char (в "попугаях"), т.е.
sizeof(char)==1 всегда
    std::cout << "sizeof(char)="  << sizeof(char) << '\n'
        << "sizeof(int)="         << sizeof(int) << '\n'
        << "sizeof(short)="       << sizeof(short) << '\n'
        << "sizeof(long)="        << sizeof(long) << '\n'
        << "sizeof(long long)="   << sizeof(long long) << '\n'
        << "sizeof(double)="      << sizeof(double) << '\n'
        << "sizeof(float)="       << sizeof(float) << '\n'
        << "sizeof(long double)=" << sizeof(long double) << '\n'
        << "sizeof(bool)="        << sizeof(bool) << '\n';

    std::cout << '\n';

    std::cout << "----- Conversions between int and bool -----\n";
    // Тип bool
    // преобразование bool в int
    bool e = true;
    std::cout << "e - bool, a - int" << '\n';
    a = e;
    std::cout << "a=e (e=true) => a=" << a << '\n';

    e = false;
    a = e;
    std::cout << "a=e (e=false) => a=" << a << '\n';

    // использование bool
    a = 100;
    b = 10;
    e = b > a;
    // при выводе на экран bool преобразуется в int (true - 1, false - 0),
    // это можно изменять с помощью манипуляторов boolalpha/noboolalpha
    std::cout << "a=100 - int,  b=10 - int, e - bool\n"
        << "e=b>a => e=" << std::boolalpha << e << std::noboolalpha << '\n';

    e = b < a;
    std::cout << "e=b<a => e=" << e << '\n';

    std::cout << '\n';

    // преобразование int в bool
    // 0 преобразуется в false остальные значения в true
    a = -100;
    e = a;
    std::cout << "a=-100 - int, e - bool\n"
        << "e=a => e=" << e << '\n';

    std::cout << '\n';

    a = 0;
```

```
    e = a;
    std::cout << "a=0 - int, e - bool\n"
        << "e=a => e=" << e << '\n';

    std::cout << '\n';

    std::cout << "----- char conversions -----\n";
    // Тип char
    // переменные char указываются в апострофах (а не в кавычках, как
строковые литералы)
    char ch = 'a';
    std::cout << "ch=" << ch << '\n'; // строка в кавычках, кстати,
воспринимается как неизменяемый массив символов char

    // при преобразовании char в int ASCII код символа становится числом;
    // при преобразовании int в char значение интерпретируется как ASCII код
(обрезается до одного байта)
    std::cout << "static_cast<int>(ch)=" << static_cast<int>(ch) << '\n';
    std::cout << "static_cast<char>(98)=" << static_cast<char>(98) << '\n';

    std::cout << '\n';

    // тип char также содержит управляющие символы, начинающиеся с "\"
    // '\a'     звонок
    // '\b'     возврат на один символ назад
    // '\f'     перевод страницы
    // '\n'     новая строка
    // '\r'     перевод каретки
    // '\t'     горизонтальная табуляция
    // '\v'     вертикальная табуляция
    // '\''     апостроф
    // '\"'     двойные кавычки
    // '\\'     обратная дробная черта
    // '\?'     вопросительный знак
    std::cout << "new\nline" << std::endl
        << "slash - \\" << std::endl
        << "apostrophe - \'" << std::endl;
    // и т.д.

    std::cout << '\n';

    // раскомментируйте строки ниже, чтобы увидеть всю таблицу ASCII
    //for (int i = 0; i <= 255; i++)
    //{
    //    ch = i;
    //    std::cout << "ASCII[" << i << "]=" << ch << '\n';
    //}

    return 0;
}
```

// Циклы и ветвления

```
#include <iostream>

int main()
{
    // Цикл while и оператор switch

    std::cout << "Enter 1, 2, 3 or 0 to exit:\n";
    char choice = '0';
    std::cin >> choice;
    while (choice != '0')
    {
        switch (choice)
        {
        case '1':
            std::cout << "\"ONE\"\n";
            // обратите внимание на отсутствие break
        case '2':
            std::cout << "\"TWO\"\n";
            break;
        case '3':
            std::cout << "\"THREE\"\n";
            break;
        default:
            std::cout << "Try again\n";
        }
        std::cin >> choice;
    }
    std::cout << "The End\n";

    std::cout << '\n';

    // Бесконечный цикл while(true), операторы break и continue

    // обратите внимание, как условиями if в сочетании с операторами break и
continue можно
    // управлять выполнением цикла;
    // в реальности следует использовать не бесконечный цикл + break,
    // а цикл с правильно сформулированным условием

    // если вводится нечётное число, то оно выводится и выполнение
продолжается,
    // если число чётное - выводится оно само и его половина;
    std::cout << "Enter integer number or \"0\" to exit: ";
    while (true)
    {
        int num = 0;
        std::cin >> num;
        // если введён 0, то бесконечный цикл прервётся из-за break
```

```
        if (num == 0)
        {
            break;
        }
        // если число окажется нечётным, то после вывода начнётся новая
итерация цикла из-за continue
        if (num % 2 != 0)
        {
            std::cout << num << " is odd\n";
            continue;
        }
        // условие if (num % 2 == 0) не требуется,
        // т.к. если это условие не выполнено (т.е. num == 0 или num % 2 !=
0),
        // то программа просто не дойдёт до выполнения следующей строки
        std::cout << num << " is even, half of " << num << " is " << num / 2
<< '\n';
    }

    std::cout << "The End\n";

    std::cout << '\n';

    // Цикл do..while

    // рекомендуется избегать цикла do..while, предпочитая ему цикл с
предусловием while

    int i = 0;
    std::cout << "Enter integer number or \"0\" to exit:\n";
    // первая итерация цикла будет выполнена независимо от начального
значения i
    do
    {
        std::cin >> i;
        std::cout << "your number is: " << i << '\n';
    } while (i != 0);

    std::cout << "The End\n";

    std::cout << '\n';

    // Вложенные циклы

    const int N = 7;
    for (int i = 0; i < N; i++)
    {
        // внутренний цикл запускается заново на каждой итерации внешнего
        for (int j = 0; j < N; j++)
        {
```

```
            // обратите внимание на то, в каких комбинациях и в каком порядке
будут выводиться индексы
            std::cout << "  (i=" << i << ";j=" << j << ")";
        }
        std::cout << '\n';
    }

    std::cout << '\n';

    // вывод треугольника из '*'
    const int SIZE = 10;
    for (int i = 0; i < SIZE; i++)
    {
        // обратите внимание, что количество итераций внутреннего цикла
зависит от i
        for (int j = i; j < SIZE; j++)
        {
            std::cout << '*';
        }
        std::cout << '\n';
    }

    return 0;
}
```

// Функции

```
// В представленных ниже примерах ошибки ввода для краткости НЕ
обрабатываются и могут вызвать некорректное повдение программы!

#include <iostream>

// глобальная переменная
int globalVariable = 1;

// Объявления функций  (определения в конце файла)
// Имена параметров при объявлении можно опускать, но лучше указывать, т.к.
это улучшает читаемость.

// функция, возводящая base в целую степень pow
double power(double base, int pow);

// функция, возвращающая значение модуля числа;
inline double absolute(double n);

// функция, возвращающая значение корня n-ной (целой) степени числа number с
точностью error
double nRoot(double number, int n, double error);

// рекурсивная функция, возвращающая наибольший общий делитель (НОД, англ.
GCD) двух чисел
int gcd(int n1, int n2);

// функция, обменивающая значения аргументов
void swap(int &n1, int &n2);

// функция, возвращающая ссылку
int & strange();

int main()
{
    int choice = 1;
    while (choice != 0)
    {
        std::cout << "Select function:\n"
            << " 1 - power\n"
            << " 2 - nRoot\n"
            << " 3 - gcd\n"
            << " 0 - exit\n";
        std::cin >> choice;

        switch (choice)
        {
        case 0:
            std::cout << "Exit\n";
```

```
            break;
        case 1:
        {
            std::cout << "Enter base: ";
            // эта переменная объявлена внутри блока {...} и существует
только внутри него
            double base = 0.0;
            std::cin >> base;

            std::cout << "Enter power: ";
            int pow = 0;
            std::cin >> pow;

            std::cout << "The number " << base << " to the power of " << pow
<< " is "
                << power(base, pow) << '\n';

            break;
        }
        case 2:
        {
            std::cout << "Enter base: ";
            // переменная с именем base уже есть в case 1,
            // но т.к. содержимое каждого case находится в собственном блоке
{...},
            // в них могут быть объявлены локальные переменные с совпадающими
именами
            double base = 0.0;
            std::cin >> base;

            std::cout << "Enter root degree: ";
            int rootDegree = 0;
            std::cin >> rootDegree;

            std::cout << "Enter error: ";
            double error = 0.0;
            std::cin >> error;

            std::cout << "The root " << rootDegree << " degrees of " << base
<< " is "
                << nRoot(base, rootDegree, error) << '\n';

            break;
        }
        case 3:
        {
            std::cout << "Enter first number: ";
            int a = 0;
            std::cin >> a;
```

```
            std::cout << "Enter second number: ";
            int b = 0;
            std::cin >> b;

            std::cout << "Numbers " << a << " and " << b << " have greatest
common divisor "
                << gcd(a, b) << '\n';

            break;
        }
        default:
            std::cout << "Try again\n";
        }
    }

    std::cout << '\n';
```

// Ссылки

```
    // Ссылку можно рассматривать как еще одно имя объекта.
    // В основном ссылки используются для задания параметров функций и возвращаемых функциями значений,
    // а также для перегрузки операций.
    // Запись T& обозначает ссылку на переменную типа T.
    int  i = 1;
    int &r = i;     // r и i  ссылаются на одно и то же целое (т.е. r становится как бы вторым именем i)

    int  x = r;     // x = 1
    std::cout << "variable i=" << i << "; reference r=" << r << "; variable x(=r)=" << x << '\n';

    r = 2;          // i = 2;
    std::cout << "variable i=" << i << "; reference r=" << r << '\n';

    // Ссылка должна быть инициализирована, т.е. должно быть нечто, что она может обозначать.
    // Следует помнить, что инициализация ссылки совершенно отличается от операции присваивания,
    // хотя можно указывать операции над ссылкой, ни одна из них на саму ссылку не действует.
    int  ii = 0;
    int &rr = ii;   // инициализация ссылки обязательна, иначе будет ошибка
    std::cout << "variable ii=" << ii << "; reference rr=" << rr << '\n';

    ++rr;           // ii увеличивается на 1
    std::cout << "variable ii=" << ii << "; reference rr=" << rr << '\n';

    // Здесь операция ++ допустима, но ++rr не увеличивает саму ссылку rr.
    // Вместо этого ++ применяется к переменной ii.
    // Следовательно, после инициализации ссылка всегда указывает на тот объект, к которому была привязана.

    std::cout << '\n';

    // применение функции swap, обменивающей значения аргументов
    int n1 = 5;
    int n2 = 10;

    std::cout << "Before: n1=" << n1 << ", n2=" << n2 << '\n';
    swap(n1, n2);
    std::cout << "After swap(n1, n2) : n1=" << n1 << ", n2=" << n2 << '\n';

    std::cout << '\n';

    // теперь небольшая "интересность", попытайтесь понять, почему так происходит
```

```
    std::cout << "before globalVariable=" << globalVariable << '\n';

    strange() = 5;     // используем функцию слева от оператора присваивания
O_o

    std::cout << "after globalVariable=" << globalVariable << '\n'; //
значение globalVariable изменилось

    // в то же время нашу функцию можно использовать и привычным образом,
    // всё дело в том, что strange() возвращает не значение а ссылку на
globalVariable,
    // т.е. её можно использовать вместо этой переменной
    std::cout << "normal usage strange(): " << strange() << '\n';

    return 0;
}

// Функция, возвращающая значение модуля числа (понадобится в следующих
функциях)
// Ключевое слово inline даёт компилятору рекомендацию заменить вызов функции
кодом из тела этой функции
inline double absolute(double n)
{
    return n < 0 ? -n : n;
}

// Функция, возводящая base в целую степень pow
// При возведении нуля в нулевую или отрицательную степень результат не
определён
//
https://ru.wikipedia.org/wiki/%D0%92%D0%BE%D0%B7%D0%B2%D0%B5%D0%B4%D0%B5%D0%B
D%D0%B8%D0%B5_%D0%B2_%D1%81%D1%82%D0%B5%D0%BF%D0%B5%D0%BD%D1%8C#%D0%A6%D0%B5%
D0%BB%D0%B0%D1%8F_%D1%81%D1%82%D0%B5%D0%BF%D0%B5%D0%BD%D1%8C

double power(double base, int pow)
{
    double res = 1.0;
    int powAbs = absolute(pow);
    for (int i = 0; i < powAbs; ++i)
    {
        res *= base;
    }
    return (pow > 0) ? res : (1.0 / res);
}

// Функция, возвращающая значение корня n-ной (целой положительной) степени
числа number с точностью error
// чтобы написать функцию воспользуемся алгоритмом нахождения корня n-ной
степени
```

```
// алгоритм (посмотрите ссылку выше, там формула выглядит понятнее):
// 1. задать произвольное X(0)
// 2. задать X(k+1) = ((N-1)*X(k) + M/X(k)^(N-1)) / N, где M - число, из
которого необходимо извлечь корень
//
                                    N - степень корня
//
                                    X(k) - значение X на текущей итерации
//
                                    X(k+1) - значение X на следующей итерации
//
                                    ^(N-1) - возведение в степень N-1
// 3. повторять шаг 2, пока не будет достигнута необходимая точность.
// Так для квадратного корня формула принимает вид X(k+1) = (X(k) + M/X(k)) /
2 - итерационная формула Герона
// Точность будем определять так:
// для указанной точности e результат функции res должен удовлетворять
условию |(res^N) - M| < e,
// т.е. результат, возведённый в степень корня, должен отличаться от
исходного числа не более,
// чем на e

double nRoot(double number, int n, double error)
{
    // если функция вызвана с некорректными аргументами - выбросим
исключение;
    // об исключениях рассказывается в другом примере
    if (number < 0 || n <= 0 || error <= 0)
    {
        throw std::invalid_argument("Invalid argument");
    }
    double res = number;       // примем X(0) = number
    // повторяем пока модуль больше допустимой ошибки
    while (absolute(power(res, n) - number) > error)
    {
        res = ((n - 1) * res + number / power(res, n - 1)) / n;
    }
    return res;
}

// Рекурсивная функция, возвращающая наибольший общий делитель двух чисел
(НОД)
// Алгоритм Евклида:
// Пусть a и b — целые числа, не равные одновременно нулю, и
последовательность чисел
//    a > b > r1 > r2 > r3 > r4 > ... >rn
// определена тем, что каждое rk — это остаток от деления предпредыдущего
числа на предыдущее,
// а предпоследнее делится на последнее нацело, то есть
//    a = b * q0 + r1
```

```
//    b = r1 * q1 + r2
//    r1 = r2 * q2 + r3
//      ...
//    r{k-2} = r{k-1} * q{k-1} + rk
//      ...
//    r{n-1} = rn * qn
// Тогда НОД(a,b), наибольший общий делитель a и b, равен rn, т.е. последнему ненулевому члену этой последовательности.
// Отсюда вытекают три факта:
// 1)НОД(n1,n2)=НОД(n2,n1)
// 2)НОД(0,n2)=n2
// 3)НОД(n1,n2)=НОД(n2,n3), где n3=n1%n2
//
// Из этих правил можно построить небольшую рекурсивную функцию

int gcd(int n1, int n2)
{
    if (n1 == 0)
    {
        return n2;
    }
    return gcd(n2 % n1, n1);
}

// Функция, обменивающая значения аргументов.
// При передаче аргументов по ссылке значения аргументов не копируются,
// вместо этого функция работает с настоящими переменными и может изменять их значения
void swap(int &n1, int &n2)
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}

// Функция, возвращающая ссылку
int & strange()
{
    return ::globalVariable;
}
```

```
#include <iostream>
#include <cmath>

typedef double(*IntegrateFunction)(double);

// Метод средних прямоугольников
double integrateRectangle(IntegrateFunction f, double a, double b, int n)
{
	double sum = 0.0;
	double step = (b - a) / n;
	for (int i = 0; i < n; ++i)
	{
		sum += f(a + i * step + step / 2.0);
	}
	return sum * step;
}

// Точность определяется по правилу Рунге
//
https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B0%D0%B2%D0%B8%D0%BB%D0%BE_%D0%
A0%D1%83%D0%BD%D0%B3%D0%B5
double integrate(IntegrateFunction f, double a, double b, double error)
{
	if (error <= 0)
	{
		throw std::invalid_argument("Error should be positive");
	}
	const double THETA = 1.0 / 3.0;
	int nSteps = 1;
	double res1 = integrateRectangle(f, a, b, nSteps);
	nSteps *= 2;
	double res2 = integrateRectangle(f, a, b, nSteps);
	while (std::abs(res2 - res1) * THETA > error)
	{
		res1 = res2;
		nSteps *= 2;
		res2 = integrateRectangle(f, a, b, nSteps);
	}
	return res2;
}

double f1(double x)
{
	return x * x;
}

double f2(double x)
{
	return x * x * x;
}
```

```
int main()
{
        double lower;
        std::cout << "Lower bound: ";
        std::cin >> lower;
        double upper;
        std::cout << "Upper bound: ";
        std::cin >> upper;
        double error;
        std::cout << "Error: ";
        std::cin >> error;

        std::cout << "x^2: " << integrate(f1, lower, upper, error) << '\n';
        std::cout << "x^3: " << integrate(f2, lower, upper, error) << '\n';
        std::cout << "1/x: " << integrate([](double x) { return 1.0 / x; },
lower, upper, error) << '\n';

        return 0;
}
```

// Массивы

```
#include <iostream>
#include <iomanip>     // для манипулятора setw()
#include <vector>

#include <cmath>    // для функции sqrt(), которая возвращает квадратный
корень аргумента
#include <ctime>    // для установки генератора случайных чисел: функции
time(0), rand()

// Объявления функций (определения в конце файла)
// Для указания размера массива можно также использовать беззнаковые типы
// unsigned int или size_t, но в любом случае, если размер указан неверно,
// то проверить это внутри функции невозможно

// Функция сортировки массива по убыванию методом выбора
// int* array - передаваемый массив
// int n - количество элементов массива
void selectionSort(int* array, int n);    // можно и так: void
selectionSort(int array[], int n);

// Функция сортировки массива по возрастанию методом пузырька
// int* array - передаваемый массив
// int n - количество элементов массива
void bubbleSort(int* array, int n);

// Функция, возвращающая евклидову норму вектора (она же длина n-мерного
вектора)
// norm_e=(v1^2+v2^2+v3^2+...+vn^2)^(1/2)
// const double* array - передаваемый массив
// int n - количество элементов массива
// здесь const зпрещает изменение элементов массива внутри функции
double euklidNorm(const double* array, int n);

// Функция возвращает сумму элементов массива std::vector<>
// const std::vector<int>& array - передаваемый массив
int getSum(const std::vector<int>& array);

int main()
{
    // установка генератора псевдослучайных чисел; функция time(0) объявлена
в <ctime>
    std::srand(std::time(0));

    // Указатели

    // Указатель - это переменная, значением которой является адрес в памяти.
    // При работе с типизированным указателем мы можем получать доступ как к
самому адресу,
```

```
    // так и к значению, находящемуся по этому адресу.

    int ivar1 = 10;
    int *iptr1 = &ivar1;       // объявление и инициализация указателя
значением адреса переменной var1
    int *iptr2(&ivar1);                // другой вариант (равнозначный)
объявления и инициализации указателя
    int *iptr3 = iptr1;                // объявление и инициализация указателя
значением другого указателя
    // здесь:
    // переменная ivar1 имеет тип int,
    // выражение &ivar1 возвращает адрес переменной ivar1 (т.е. значение типа
int*),
    // переменная iptr1 имеет тип int* (указатель на int),
    // выражение *iptr1 возвращает значение по адресу из iptr1 (т.е. значение
типа int, показано ниже)

    // выведем все переменные и адреса
    std::cout << "&var1=" << &ivar1 << ", var1=" << ivar1 << '\n'
        << "ptr1=" << iptr1 << ", *ptr1=" << *iptr1 << '\n'
        << "ptr2=" << iptr2 << ", *ptr2=" << *iptr2 << '\n'
        << "ptr3=" << iptr3 << ", *ptr3=" << *iptr3 << '\n';
    std::cout << std::endl;

    *iptr3 = 20;       // изменение значения по адресу
    std::cout << "&var1=" << &ivar1 << ", var1=" << ivar1 << '\n'
        << "ptr1=" << iptr1 << ", *ptr1=" << *iptr1 << '\n'
        << "ptr2=" << iptr2 << ", *ptr2=" << *iptr2 << '\n'
        << "ptr3=" << iptr3 << ", *ptr3=" << *iptr3 << '\n';
    std::cout << std::endl;

    double dvar1 = 2.5;
    // указателю на void (типу void*) можно присвоить значение уазателя на
любой другой тип;
    // разыменовывать указатель void* нельзя,
    // т.к. компилятор не может знать, как интерпретировать значение по
данному адресу
    void* vptr = &dvar1;
    // использовать указатель void* можно совместно с явным приведением типов
    double* dptr1 = static_cast<double*>(vptr);
    std::cout << "vptr=" << vptr << ", *(static_cast<double*>(vptr))=" <<
*(static_cast<double*>(vptr)) << '\n'
        << "&dvar1=" << &dvar1 << ", dvar1=" << dvar1 << '\n'
        << "dptr1=" << dptr1 << ", *dptr1=" << *dptr1 << '\n';
    std::cout << std::endl;

    // Массивы

    const int N_FOO_ARRAY = 10;
    // объявление массива с инициализацией;
```

```
    // неуказанные в инициализаторе элементы заполняются нулями (но только
если инициализатор есть!)
    int foo[N_FOO_ARRAY]{ 1, 2, 3, 4, 5 };
    // если размер не указан явно, он вычисляется из инициализатора
    int bar[]{ 4, 3, 2, 1 };

    // количество элементов в массиве можно вычислить с помощью оператора
sizeof,
    // но это возможно только когда доступен сам массив, а не указатель на
его первый элемент
    std::cout << "Size of bar is " << sizeof(bar) / sizeof(bar[0]) << '\n';

    // имя массива может быть использовано в качестве указателя на его первый
элемент
    std::cout << "Address of foo: " << foo << '\n';
    std::cout << "First element of foo : " << *foo << '\n';

    // к элементам массива можно обращаться с помощью оператора индексации
foo[i],
    // а можно с помощью адресной арифметики *(foo + i),
    // что для компилятора - одно и то же
    std::cout << "foo[2]: " << foo[2] << '\n';
    std::cout << "*(foo + 2): " << *(foo + 2) << '\n';

    std::cout << '\n';

    // Сортировка массива выбором

    const int N_FIRST_ARRAY = 10;
    int firstArray[N_FIRST_ARRAY];

    std::cout << "Selection sort\n";
    // инициализация массива случайными числами от -50 до 49;
    // функция rand() используется для получения псевдослучайных чисел в
диапазоне от 0 до RAND_MAX
    // константа RAND_MAX определена в stdlib
    for (int i = 0; i < N_FIRST_ARRAY; ++i)
    {
        firstArray[i] = std::rand() % 100 - 50;
    }

    // вывод массива
    std::cout << "Array before sorting: ";
    for (int i = 0; i < N_FIRST_ARRAY; ++i)
    {
        std::cout << std::setw(4) << firstArray[i];
    }
    std::cout << '\n';

    // сортировка массива методом выбора максимума;
```

```
    // в функцию передаётся только имя массива,
    // массив при передаче в функцию не копируется, после работы функции
массив остаётся изменённым
    selectionSort(firstArray, N_FIRST_ARRAY);

    std::cout << "Array after sorting: ";
    for (int i = 0; i < N_FIRST_ARRAY; ++i)
    {
        std::cout << std::setw(4) << firstArray[i];
    }
    std::cout << "\n\n";

    // Вычисление евклидовой нормы вектора

    const int N_SECOND_ARRAY = 3;
    double secondArray[N_SECOND_ARRAY];

    for (int i = 0; i < N_SECOND_ARRAY; ++i)
    {
        secondArray[i] = std::rand() % 41 - 20;
    }

    std::cout << "Vector:";
    for (int i = 0; i < N_SECOND_ARRAY; ++i)
    {
        std::cout << std::setw(4) << secondArray[i];
    }
    std::cout << "  Norm: " << euklidNorm(secondArray, N_SECOND_ARRAY);

    std::cout << "\n\n";

    // Динамическое выделение памяти. Сортировка пузырьком

    std::cout << "Bubble sort\n";

    int nDynArray = 15;        // размер массива (переменная!)

    // выделение памяти под массив из nDynArray элементов типа int
    // и помещение указателя на эту память в переменную-указатель dynArray;
    int* dynArray = new int[nDynArray];

    for (int i = 0; i < nDynArray; ++i)
    {
        dynArray[i] = std::rand() % 10;
    }

    std::cout << "Array before sorting: ";
    for (int i = 0; i < nDynArray; ++i)
    {
        std::cout << std::setw(3) << dynArray[i];
```

```
    }
    std::cout << '\n';

    // обратите внимание, что функция не знает, в какой области памяти
находятся элементы массива
    // и может работать как с динамическим массивом, так и с обычным
    bubbleSort(dynArray, nDynArray);

    std::cout << "Array after sorting: ";
    for (int i = 0; i < nDynArray; ++i)
    {
        std::cout << std::setw(3) << dynArray[i];
    }

    // оператор delete освобождает память, привязанную к указателю;
    // для массивов требуются скобки delete[]
    delete[] dynArray;

    std::cout << "\n\n";

    // Динамические массивы vector из стандартной библиотеки шаблонов (STL;
необходимо подключить <vector>)

    std::vector<int> vectArray; // создаем массив целых чисел,
                                // синтаксис может сейчас казаться странным,
но потом станет понятно почему так

    std::cout << "Vector(1) size: " << vectArray.size() << '\n';     //
встроенный метод size возвращает размер массива

    for (int i = 0; i < 5; ++i)
    {
        vectArray.push_back(i);       // метод push_back помещает аргумент в
конец массива
    }

    std::cout << "Vector(2): [";
    for (size_t i = 0; i < vectArray.size(); ++i)
    {
        std::cout << std::setw(4) << vectArray[i];
    }
    std::cout << "]; size: " << vectArray.size() << '\n';

    vectArray.push_back(5);
    vectArray.push_back(6);
    vectArray.insert(vectArray.begin() + 5, 15);     // vectArray.begin() -
итератор (указатель) на начало массива

    // обратите внимание на то, что размер vector может меняться
    // посмотрите, куда вставился элемент 15
```

```
    std::cout << "Vector(3): [";
    for (size_t i = 0; i < vectArray.size(); ++i)
    {
        std::cout << std::setw(4) << vectArray[i];
    }
    std::cout << "]; size: " << vectArray.size() << '\n';

    std::cout << "Sum: " << getSum(vectArray) << '\n';

    return 0;
}

// Функция сортировки массива по убыванию методом выбора
// int* array - передаваемый массив (аналогично int array[])
// int n - количество элементов массива
// ВАЖНО!!! т.к. в функцию передаётся не массив, а только указатель на его первый элемент,
// внутри функции невозможно узнать размер массива с помощью оператора sizeof(),
// а таже использовать цикл по диапазону for (int i : array)
void selectionSort(int* array, int n)
{
    for (int i = 0; i < n - 1; ++i)   // i < n - 1, т.к. последний элемент отсортируется автоматически
    {
        // инициализируем iMax в начале каждого цикла первым элементом из оставшейся части массива
        int iMax = i;
        // ищем максимальный элемент в оставшейся части массива
        for (int j = i + 1; j < n; ++j)
        {
            if (array[j] > array[iMax])
            {
                iMax = j;
            }
        }
        // помещаем максимальный элемент на место первого элемента оставшейся части массива (меняем местами)
        if (iMax != i)
        {
            int temp = array[i];
            array[i] = array[iMax];
            array[iMax] = temp;
            // можно было бы использовать стандартную функцию std::swap
        }
    }
}

// Функция сортировки массива по возрастанию методом пузырька
// int* array - передаваемый массив
```

```
// int n - количество элементов массива
void bubbleSort(int* array, int n)
{
    for (int i = 0; i < n - 1; ++i)
    {
        for (int j = 0; j < n - i - 1; ++j)
        {
            // если текущий элемент больше следующего за ним, то меняем их
местами
            if (array[j] > array[j + 1])
            {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}

// Функция, возвращающая евклидову норму вектора (она же длина n-мерного
вектора)
// norm_e=(v1^2+v2^2+v3^2+...+vn^2)^(1/2)
// const double* array - передаваемый массив
// int n - количество элементов массива
double euklidNorm(const double* array, int n)
{
    double res = 0;
    for (int i = 0; i < n; ++i)
    {
        res += array[i] * array[i];
    }
    return std::sqrt(res);  // функция sqrt() возвращает квадратный корень
аргумента (описана в <cmath>)
}

// Функция возвращает сумму элементов массива std::vector<>
// const std::vector<int>& array - передаваемый массив
// массивы std::vector<>, как и другие объекты слдует передавать в функции по
ссылкам, чтобы избежать копирования;
// размер массива отдельно передавать не нужно, т.к. объект std::vector<>
содержит в себе эту информацию
int getSum(const std::vector<int>& array)
{
    int sum = 0;
    for (int n : array)
    {
        sum += n;
    }
    return sum;
}
```

// Строки

```
#include <iostream>
#include <iomanip>     // для манипулятора setw()
#include <cstring>     // для функций, работающих с char*-строками
#include <string>   // для класса std::string

// Объявления функций (определения в конце файла)

// Обратите внимание, что при работе с char*-строками в функции обычно не передаётся размер,
// т.к. конец строки определяется не размером массива, а символом '\0',
// однако, в некоторых случаях можно для большей безопасности передать размер массива,
// в который записывается результат

// Функция, возвращающая количество символов в строке
// const char* str - передаваемая строка
int length(const char* str);

// Функция, копирующая строку
// const char* src - копируемая строка
// const char* dest - строка, в которую происходит копирование
// возвращаемое значение - указатель на начало скопированной строки
char* copy(char* dest, const char* src);

// Функция, переворачивающая строку
// const std::string& str - исходная строка
std::string reverse(const std::string& str);

int main()
{
    // Массивы символов

    // Строки в C и C++ реализуются как массивы элементов char (так называемые char*-строки)
    // Важно помнить, что все char*-строки завершаются символьной константой '\0', ASCII код которой равен 0
    // '\0' называется нулевым символом и обозначает логический конец строки

    const int MAX_N_STRING = 20;
    char strChar[MAX_N_STRING] = "";

    std::cout << "Enter string without spaces: ";
    std::cin >> std::setw(MAX_N_STRING) >> strChar;
    std::cout << "You entered: " << strChar << '\n';
    std::cout << "String length: " << length(strChar) << '\n';

    // Обратите внимание:
    // 1. У строк также есть размер, который должен быть указан.
```

```
    //    Если размер заранее неизвестен - нужно взять с запасом
    // 2. Чтобы избежать переполнения в программе использован манипулятор
setw.
    //    Если введённая строка окажется больше MAX_STRING_SIZE,
    //    то она будет обрезана и последним символом strChar[19] будет '\0'
    //    (попробуйте убрать  setw и ввести длинную строку)
    // 3. Операторы "<<" и ">>" знают, что ожидается ввод/вывод строки
(циклов не требуется)
    // 4. Строка не может содержать пробелов (всё, что после пробела просто
отбрасывается),
    //    т.к. оператор ">>" считает пробел символом-разделителем

    // В стандартной библиотеке языка имеется файл <cstring>, содержащий
функции для работы с char*-строками.
    // Например, в нём есть стандартная функция, возвращающая длину строки:
    std::cout << "String length (cstring): " << std::strlen(strChar) << '\n';

    std::cout << '\n';

    // Ввод строки с пробелами

    std::cout << "Enter string with spaces: ";

    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    std::cin.getline(strChar, MAX_N_STRING);

    std::cout << "You entered: " << strChar << '\n';
    std::cout << "String length: " << length(strChar) << '\n';
    // getline - метод класса istream, аргументы - массив и его размер (для
предотвращения переполнения)
    // ignore - метод объекта istream для очистки потока ввода после операции
">>",
    // аргументы - кол-во символов для очистки и символ-разделитель (тоже
будет очищен)
    // выражение std::numeric_limits<std::streamsize>::max() возвращает
максимально возможный размер буфера,
    // т.е. в этом случае удалятся все символы до указанного разделителя

    std::cout << '\n';

    // скопируем strChar в strCharCopy

    char strCharCopy[MAX_N_STRING] = "";
    copy(strCharCopy, strChar);

    std::cout << "strChar: " << strChar << '\n';
    std::cout << "strCharCopy: " << strCharCopy << '\n';

    std::cout << '\n';
```

```
    // Класс string из стандартной библиотеки шаблонов (STL; необходимо
подключить <string>)

    // с объектами string можно обращаться как с обычными переменными
    std::string firstStr("firstStr");     // 1 способ инициализации
    std::string secondStr = "secondStr";  // 2 способ инициализации
    std::string thirdStr;                 // без инициализации

    thirdStr = firstStr + " plus " + secondStr;
    std::cout << "thirdStr: " << thirdStr << '\n';

    std::cout << "Reversed thirdStr: " << reverse(thirdStr) << '\n';

    // Класс string содержит множество встроенных методов
    // Один из них - метод find возвращает позицию вхождения подстроки в
строку либо -1 (std::string::npos)

    std::cout << "Enter first string: ";
    std::cin >> firstStr;

    std::cout << "Enter second string: ";
    std::cin >> secondStr;

    int n = firstStr.find(secondStr);

    if (n != std::string::npos)
    {
        std::cout << "Second string is in the first string on position " << n
<< '\n';
    }
    else
    {
        std::cout << "Second string is not in the first string\n";
    }

    return 0;
}

// Функция, возвращающая количество символов в строке
// const char* str - передаваемая строка
int length(const char* str)
{
    int result = 0;
    while (*str)
    {
        ++result;
        ++str;
        // обратите внимание, что запись const char* str запрещает менять
значение по указателю,
        // но не мешает менять сам указатель
```

```
    }
    return result;
}

// Функция, копирующая строку
// const char* src - копируемая строка
// const char* dest - строка, в которую происходит копирование
// возвращаемое значение - указатель на начало скопированной строки
// ВАЖНО!!! строка dest должна быть достаточного размера, чтобы вместить
строку src (включая '\0'),
// иначе возникнет неопределённое поведение
// (для большей защиты можно было бы добавить размер dest в качестве
параметра функции)
char* copy(char* dest, const char* src)
{
    int i = 0;
    while (*src)
    {
        *(dest + i) = *src;
        ++i;
        ++src;
    }
    *(dest + i) = '\0';
    return dest;
}

// Функция, переворачивающая строку
// const std::string& str - исходная строка
// строки std::string, как и другие объекты, следует передавать в функции по
ссылкам, чтобы избежать копирования
std::string reverse(const std::string& str)
{
    // класс std::string сам управляет размещением символов строки в памяти,
    // а также обеспечивает корректное копирование строки при присваивании
или возврате из функции
    std::string result;
    // резервируем необходимое количество памяти,
    // чтобы избежать лишних действий
    result.reserve(str.size());
    for (int i = str.length() - 1; i >= 0; --i)
    {
        result += str[i];
    }
    return result;
}
```

// Двумерные массивы

```
#include <iostream>
#include <iomanip>
#include <ctime>

// глобальные константы для массива постоянной размерности
const int N_ROW = 3;
const int N_COL = 5;

// Объявления функций (определения в конце файла)

// Функция, возвращающая максимум двумерного массива;
// обратите внимание, что в случае двумерного массива постоянной размерности
(не динамического),
// второй размер обязтельно(!) должен быть указан
// int array[][N_COL] - передаваемый массив
// int nRow - количество строк массива
// int nCol - количество столбцов массива
int getMaxValue(const int array[][N_COL], int nRow, int nCol);

// Функция возвращает среднее значение элементов двумерного массива
постоянной размерности
// const int* array - указатель на первый элемент двумерного массива
// int nRow - количество строк массива
// int nCol - количество столбцов массива
double getAverage(const int* array, int nRow, int nCol);

// Функция выделяет динамическую память для двумерного массива
// int**& array - указатель на двумерный динамический массив, передаётся по
ссылке
// int nRow - количество строк массива
// int nCol - количество столбцов массива
void allocateArray(int**& array, int nRow, int nCol);

// Функция освобождает динамическую память, выделеную для двумерного массива
// int** array - указатель на двумерный динамический массив
// int nRow - количество строк массива
void deallocateArray(int** array, int nRow);

// Функция меняет местами две строки матрицы без непосредственного
перемещения элементов в памяти
// !!! защиты от выхода из допустимого диапазона номеров нет
// int** array - указатель на двумерный динамический массив
// int iLine1 - индекс первой строки
// int iLine2 - индекс второй строки
void swapRows(int** array, int iLine1, int iLine2);

int main()
{
```

```
  std::srand(std::time(0));

  // Поиск максимума двумерного массива (матрицы)

  // объявление двумерного массива с N_ROW строками и N_COL столбцами
  int matrix[N_ROW][N_COL];

  // задание значений элементов массива и вывод на экран
  for (int i = 0; i < N_ROW; ++i)
  {
    for (int j = 0; j < N_COL; ++j)
    {
      matrix[i][j] = std::rand() % 100 - 50;
      std::cout << std::setw(4) << matrix[i][j];
    }
    std::cout << '\n';
  }
  std::cout << '\n';

  std::cout << "Max element of matrix is " << getMaxValue(matrix, N_ROW,
N_COL) << '\n';

  std::cout << '\n';

  // Передача двумерного массива в функцию при помощи указателя на его первый
элемент (хак)

  const int N_ARRAY = 5;
  int matrix2[N_ARRAY][N_ARRAY];

  for (int i = 0; i < N_ARRAY; ++i)
  {
    for (int j = 0; j < N_ARRAY; ++j)
    {
      matrix2[i][j] = std::rand() % 10;
      std::cout << std::setw(2) << matrix2[i][j];
    }
    std::cout << '\n';
  }

  // вывод среднего значения элементов массива
  std::cout << "average value: " << getAverage(*matrix2, N_ARRAY, N_ARRAY) <<
'\n';
  // обратите внимание на то, что в функцию передаётся *matrix2, а не
matrix2;
  // *matrix2 имеет тип int* (чего и требует наша функция),
  // тогда как matrix2 имеет тип int(*)[N_ARRAY], т.е. "указатель на массив
int размером N_ARRAY"

  std::cout << '\n';
```

```
  // Создание динамического двумерного массива и передача его в функцию

  int nRow = 7;        // размеры массива (переменные!)
  int nCol = 5;

  // объявление указателя на указатель на int (адрес двумерного массива)
  int** dynTwoArray = nullptr;
  // при выделении памяти может возникнуть исключение bad_alloc,
  // поэтому работу с массивом производим в блоке try, чтобы избежать утечек
памяти
  try {
    // выделение памяти
    allocateArray(dynTwoArray, nRow, nCol);

    // задание значений элементов массива и вывод на экран
    for (int i = 0; i < nRow; ++i)
    {
      for (int j = 0; j < nCol; ++j)
      {
        dynTwoArray[i][j] = std::rand() % 100;   // соответственно
*(*(dynTwoArray + i) + j) эквивалентно dynTwoArray[i][j]
        std::cout << std::setw(3) << dynTwoArray[i][j];
      }
      std::cout << '\n';
    }

    std::cout << '\n';

    // поменяем местами вторую и шестую строки матрицы (индексы начинаются с
0)
    swapRows(dynTwoArray, 1, 5);

    for (int i = 0; i < nRow; ++i)
    {
      for (int j = 0; j < nCol; ++j)
      {
        std::cout << std::setw(3) << dynTwoArray[i][j];
      }
      std::cout << '\n';
    }
    std::cout << std::endl;
  }
  // перехватываем любые возможные исключения и выводим сообщение об ошибке
  catch (...)
  {
    std::cerr << "Error!\n";
  }

  // освобождение памяти
```

```
  deallocateArray(dynTwoArray, nRow);

  return 0;
}

// Функция, возвращающая максимум двумерного массива;
// обратите внимание, что в случае двумерного массива постоянной размерности
(не динамического),
// второй размер обязательно(!) должен быть указан
// int array[][N_COL] - передаваемый массив
// int nRow - количество строк массива
// int nCol - количество столбцов массива
int getMaxValue(const int array[][N_COL], int nRow, int nCol)
{
  int maxValue = array[0][0];
  for (int i = 0; i < nRow; ++i)
  {
    for (int j = 0; j < nCol; ++j)
    {
      if (array[i][j] > maxValue)
      {
        maxValue = array[i][j];
      }
    }
  }
  return maxValue;
}

// Функция возвращает среднее значение элементов двумерного массива.
// Обратите внимание, что размеры массива передаются функции как параметры, а
не жёстко указываются в объявлении.
// Стоит вспомнить, что двумерный массив представляет собой массив одномерных
массивов,
// а его элементы располагаются в памяти последовательно.
// Первый аргумент функции типа int* будем понимать, как указатель на первый
элемент первого одномерного массива.
// Именно поэтому в функцию из main был передан *matrix2, а не matrix2,
// т.к. matrix2 представляет собой указатель на первый одномерный массив, а
не на первый элемент
// const int* array - указатель на первый элемент двумерного массива
// int nRow - количество строк массива
// int nCol - количество столбцов массива
double getAverage(const int* array, int nRow, int nCol)
{
  double sum = 0;
  for (int i = 0; i < nRow; ++i)
  {
    for (int j = 0; j < nCol; ++j)
    {
```

```
      // вычисляем адрес элемента array[i][j] как адрес начала массива
(array) плюс
      // смещение на кол-во строк (nCol * i) плюс смещение внутри строки (j)
      sum += *(array + nCol * i + j);
    }
  }
  return sum / nRow / nCol;
}

// Функция выделяет динамическую память для двумерного массива
// int**& array - указатель на двумерный динамический массив, передаётся по
ссылке
// int nRow - количество строк массива
// int nCol - количество столбцов массива
void allocateArray(int**& array, int nRow, int nCol)
{
  // выделение памяти на массив из nRow указателей на int; new вернёт int**
  // массив будет инициализирован нулями (C++11);
  // это нужно для того, чтобы в случае, если при выделении памяти под строки
возникнет bad_alloc,
  // можно было бы безопасно применить delete[] к указателям nullptr
  array = new int* [nRow] {nullptr};
  // выделение памяти для каждой строки массива
  for (int i = 0; i < nRow; ++i)
  {
    array[i] = new int[nCol];           // можно и *(array + i) = new
int[nCol];
                                        // выражения *(array + i) и array[i]
эквивалентны
  }
}

// Функция освобождает динамическую память, выделеную для двумерного массива
// int** array - указатель на двумерный динамический массив
// int nRow - количество строк массива
void deallocateArray(int** array, int nRow)
{
  // если указатель array пустой, то освобождать ничего не нужно
  if (array)
  {
    // освобождение памяти производится в порядке обратном выделению
    for (int i = 0; i < nRow; ++i)
    {
      delete[] array[i];
    }
    delete[] array;
  }
}
```

```
// Функция меняет местами две строки матрицы без непосредственного
перемещения элементов в памяти
// !!! Защиты от выхода из допустимого диапазона номеров не сделано
// int** array - указатель на двумерный динамический массив
// int iLine1 - индекс первой строки
// int iLine2 - индекс второй строки
void swapRows(int** array, int iLine1, int iLine2)
{
  int* temp = array[iLine1];      // array[iLine1] аналогично *(array +
iLine1);
  array[iLine1] = array[iLine2];
  array[iLine2] = temp;
}
// эта функция прекрасно показывает, что указатели позволяют нам быстро
логически перемещать
// большие объёмы данных без их физического переноса в памяти
```

// Класс Person

```
// В VisualStudio стандартная функция std::localtime считается небезопасной,
// можно использовать вместо неё localtime_s или объявить идентификатор
_CRT_SECURE_NO_WARNINGS
#define _CRT_SECURE_NO_WARNINGS

#include <iostream>
#include <string>
#include <ctime>

// класс, содержащий базовую информацию о человеке
class Person
{
public:
  // конструктор - особый метод класса, использующийся для инициализации
свойств при создании объекта
  Person();                                  // конструктор без параметров
  Person(const std::string&, short);         // конструктор с двумя параметрами

  void setName(const std::string&);          // задать имя
  std::string getName() const;               // получить имя
  void setYear(short);                       // задать год рождения
  short getYear() const;                     // получить год рождения
  short getAge() const;                      // возвращает возраст человека в
текущем году

private:
  std::string name_;   // имя человека
  short birthYear_;    // год рождения
};

// функция сортировки массива указателей на объекты Person
void sortPerson(Person**, int);

// функция для вывода информации о человеке в указанный поток
void printInfo(const Person& person, std::ostream& stream);

int main()
{
  const int SIZE = 5;
  Person* persons[SIZE]{ nullptr };  // массив указателей на объекты Person
  try
  {
    persons[0] = new Person("Ilya", 1989);
    persons[1] = new Person("Daniil", 1990);
    persons[2] = new Person("Mikhail", 1985);
    persons[3] = new Person("Olga", 1995);
    persons[4] = new Person("Anastasiya", 1983);
```

```
    for (int i = 0; i < SIZE; ++i)
    {
      printInfo(*persons[i], std::cout);
    }

    sortPerson(persons, SIZE);

    std::cout << std::endl;

    for (int i = 0; i < SIZE; ++i)
    {
      printInfo(*persons[i], std::cout);
    }
  }
  catch (...)
  {
    std::cout << "Error!\n";
  }

  for (int i = 0; i < SIZE; ++i)
  {
    delete persons[i];
  }

  return 0;
}

// методы класса должны быть объявлены внутри класса, но определены могут
быть вне его;
// для этого применяется такой синтаксис <тип_возвр_значения>
<имя_класса>::<имя_метода>(<параметры>)
Person::Person() :
  name_(""),
  birthYear_(0)
{}

Person::Person(const std::string& name, short birthYear) :
  name_(name),
  birthYear_(birthYear)
{}

// обратите внимание, что передаётся ссылка на объект, это позволяет избежать
копирования строки
void Person::setName(const std::string& name)
{
  name_ = name;
}

// методы, отмеченные ключевым словом const не могут изменять значений
свойств объекта
```

```
std::string Person::getName() const
{
  return name_;
}

void Person::setYear(short birthYear)
{
  birthYear_ = birthYear;
}

short Person::getYear() const
{
  return birthYear_;
}

// получение текущего года - не такое уж простое действие, используются типы
time_t и tm
short Person::getAge() const
{
  std::time_t ctime;
  std::time(&ctime);
  std::tm* parsedTime = std::localtime(&ctime);
  // в поле tm_year структуры tm хранится количество лет после 1900 года
  return (parsedTime->tm_year + 1900) - birthYear_;
}

// сортировка людей по возрасту
// persons - указатель на массив указателей на объекты Person
// обратите внимание, что переставляются только указатели, а не сами объекты
void sortPerson(Person** persons, int size)
{
  for (int i = 0; i < size - 1; ++i)
  {
    for (int j = 0; j < size - i - 1; ++j)
    {
      if (persons[j]->getYear() > persons[j + 1]->getYear())
      {
        Person* temp = persons[j];
        persons[j] = persons[j + 1];
        persons[j + 1] = temp;
      }
    }
  }
}

void printInfo(const Person& person, std::ostream& stream)
{
  stream << person.getName() << " is " << person.getAge() << " years old\n";
}
```

// Класс комплексного числа, пример перегрузки операторов

```
#include <iostream>
#include <cmath>

// класс для работы с комплексными числами
class Complex
{
public:
  // если в классе определён конструктор с параметрами, то конструктор по
умолчанию не создаётся,
  // это значит, что нельзя создать комплексное число без явной инициализации
  Complex(double re, double im = 0) :
    re_(re),
    im_(im)
  {}

  // методы для задания полей объекта отсутствуют,
  // т.е. после создания объект нельзя изменить

  double getRe() const
  {
    return re_;
  }

  double getIm() const
  {
    return im_;
  }

  // модуль комплексного числа
  double abs() const;

  // метод для сложения двух комплексных чисел;
  Complex add(const Complex& c) const;

  // объявления дружественных функций для перегрузки операторов

  // сложение (как и другие операции) может быть реализовано в виде
перегруженного оператора +
  friend Complex operator+(const Complex& c1, const Complex& c2);

  // можно перегрузить оператор вывода << с помощью внешней дружественной
функции
  friend std::ostream& operator<<(std::ostream& stream, const Complex& c);

private:
  // поля класса, помеченные словом const, можно инициализировать только в
конструкторе
  const double re_;
```

```
  const double im_;
};

// нетривиальные методы класса определим вне его тела

double Complex::abs() const
{
  return std::sqrt(re_ * re_ + im_ * im_);
}

Complex Complex::add(const Complex& c) const
{
  // возвращаем созданный "на лету" анонимный объект,
  // для этого нужен конструктор с двумя параметрами
  return Complex(re_ + c.re_, im_ + c.im_);
}

// вывод комплексного числа в поток
void print(const Complex& c, std::ostream& stream)
{
  stream << c.getRe()
    << (c.getIm() < 0 ? "-i*" : "+i*")
    << std::abs(c.getIm());
}

// определения дружественных функций для перегрузки операторов

Complex operator+(const Complex& c1, const Complex& c2) {
  return Complex(c1.re_ + c2.re_, c1.im_ + c2.im_);
}

// дружественные функции могут обращаться к закрытым (private) членам класса
std::ostream& operator<<(std::ostream& stream, const Complex& c)
{
  return stream << c.re_
    << (c.im_ < 0 ? "-i*" : "+i*")
    << std::abs(c.im_);
}

int main()
{
  Complex c1(7, -2.5);
  Complex c2(9, 2.2);
  // обратите внимание, это не оператор присваивания, а конструктор
копирования (создан по умолчанию)
  Complex c3 = c1.add(c2);
  print(c3, std::cout);
  std::cout << '\n';
  std::cout << c3.abs() << '\n';
```

```
  // использование перегруженных операторов
  Complex c4 = c1 + c2;
  Complex c5 = c1 + 5.8;
  Complex c6 = 5.0 + c2;
  std::cout << c4 << '\n';
  std::cout << c5 << '\n';
  std::cout << c6 << '\n';

  return 0;
}

// В примере демонстрируются копирование и перемещение объектов.

// Сам класс не выполняет никакой полезной работы и содержит два поля:
// int *a_ - указатель на массив в динамической памяти; массив создаётся и
инициализируется в конструкторе
// int id_ - идентификатор объекта; присваивается в конструкторах
// Все методы выводят на экран вспомогательную информацию, показывающую,
какой метод и для какого объекта вызывается
// Попробуйте самостоятельно воспользоваться различными вариантами создания и
использования объектов,
// чтобы посмотреть на работу конструкторов и операторов присваивания.
// Попробуйте закомментировать те или иные конструкторы и операторы
присваивания, посмотрите, как изменится поведение программы

// Возврат объекта из функции (не по ссылке) должен вызывать создание
временного объекта,
// но компилятор, в зависимости от настроек, может применять определённые в
стандарте оптимизации copy/move elision
// и избегать создания ненужных объектов
// https://en.cppreference.com/w/cpp/language/copy_elision
// https://en.wikipedia.org/wiki/Copy_elision

#include <iostream>
#include <cmath>

class A
{
public:
  // Конструктор
  // Назначает объектам уникальный идентификатор с помощью статической
переменной counter,
  // выделяет динамическую память под массив размера SIZE
  // и инициализирует его элементы значением id_
  // (просто чтобы видеть, что в массиве есть какие-то данные)
  A() :
    a_(new int[SIZE])
  {
    static int counter = 0;
    id_ = ++counter;
    for (int i = 0; i < SIZE; i++)
```

```
  {
    a_[i] = id_;
  }
  std::cout << "Constructor: " << id_ << '\n';
}

// Деструктор
~A()
{
  delete[] a_;
  std::cout << "Destructor: " << id_ << '\n';
}

// Конструктор копирования
// Копирует элементы массива в новую область памяти,
// новый объект получает идентификатор на 100 больший, чем исходный
A(const A& src) :
  id_(src.id_ + 100),
  a_(new int[SIZE])
{
  for (int i = 0; i < SIZE; i++)
  {
    a_[i] = src.a_[i];
  }
  std::cout << "Copy constructor: " << src.id_ << " to " << id_ << '\n';
}

// Конструктор перемещения
// Переносит массив из исходного объекта в новый,
// новый объект получает идентификатор на 1000 больший, чем исходный
A(A&& src) noexcept :
  id_(src.id_ + 1000),
  a_(src.a_)
{
  src.a_ = nullptr;
  std::cout << "Move constructor: " << src.id_ << " to " << id_ << '\n';
}

// Оператор копирующего присваивания
A& operator=(const A& src)
{
  if (this != &src)
  {
    int* temp = a_;
    a_ = new int[SIZE];
    delete[] temp;
    for (int i = 0; i < SIZE; i++)
    {
      a_[i] = src.a_[i];
    }
```

```
      std::cout << "Copy assignment: " << src.id_ << " to " << id_ << '\n';
    }
    return *this;
  }

  // Оператор перемещающего присваивания
  A& operator=(A&& src) noexcept
  {
    if (this != &src)
    {
      delete[] a_;
      a_ = src.a_;
      src.a_ = nullptr;
      std::cout << "Move assignment: " << src.id_ << " to " << id_ << '\n';
    }
    return *this;
  }

  int getId() const
  {
    return id_;
  }

private:
  static const int SIZE = 10; // размер массива - статическая константа

  int id_;
  int* a_;
};

// Функции, принимающие и возвращающие объект типа A различным образом
// (можете дописать собственные)

A test(A x) {
  return x;
}

A& test2(A& x) {
  return x;
}

A test3()
{
  A a;
  return a;
}

A test4()
{
  return A();
```

```
}

int main()
{
  A a1;

  A a2 = test(a1);

  //std::cout << test(a1).getId() << '\n';

  //A a3 = a1;
  //A a4;
  //a4 = a1;

  // ... попробуйте свои варианты создания и использования объектов

  return 0;
}
```

// Стек

```
#include <iostream>

// Стек - структура данных, работающая по принципу LIFO (Last In, First Out),
// т.е. элементы извлекаются из стека в порядке, обратном порядку их
добавления.
// Ниже реализован стек целых чисел на базе связного списка
class StackInt
{
public:
    // конструктор
    StackInt() :
        head_(nullptr) // nullptr - пустой указатель (не указывает ни на какой
адрес)
        // устаревший аналог - NULL
    {}

    // деструктор освобождает память, выделенную под список (вызывается
неявно при уничтожении объекта);
    // кода объект содержит в себе какой-либо внешний ресурс (например,
динамически выделенную память),
    // необходимо(!) реализовать (или запретить) конструктор копирования,
оператор присвивания и деструктор,
    // в C++11 желательно также определить конструктор перемещения и оператор
перемещающего присваивания
    ~StackInt()
    {
        while (head_)    // то же, что while (head_ != nullptr)
        {
            Node* temp = head_;
            head_ = head_->next;
            // обязательно освобождаем память, выделенную под каждый элемент
            delete temp;
        }
    }

    void push(int);    // добавление элемента в стек
    int pop();           // извлечение элемента из стека

    // вывод всего стека на экран
    friend std::ostream& operator<<(std::ostream& stream, const StackInt&
stack);

private:
    // структура, описывающая элемент стека
    struct Node
    {
        int data;                // данные
        Node* next;    // указатель на следующий элемент
```

```
    };

    // указатель на вершину стека
    Node* head_;

    // для упрощения примера копирование и перемещение объектов запрещены
    StackInt(const StackInt&) = delete;
    StackInt(StackInt&&) = delete;
    StackInt& operator=(const StackInt&) = delete;
    StackInt& operator=(StackInt&&) = delete;
};

// метод, помещающий значение в стек
void StackInt::push(int val)
{
    Node* newNode = new Node;
    newNode->data = val;
    newNode->next = head_;
    head_ = newNode;
}

// метод, извлекающий значение из стека;
// извлекаемое значение возвращается функцией, элемент при этом удаляется из
стека
int StackInt::pop()
{
    if (!head_)
    {
        throw std::logic_error("Stack is empty");
    }
    int res = head_->data;
    Node* temp = head_;
    head_ = head_->next;
    delete temp;
    return res;
}

// вывод всего стека на экран
std::ostream& operator<<(std::ostream& stream, const StackInt& stack)
{
    StackInt::Node* temp = stack.head_;
    while (temp)
    {
        stream << temp->data << "  ";
        temp = temp->next;
    }
    return stream;
}

int main()
```

```
{
    StackInt stack;

    // пытаемся извлечь значение из пустого стека
    try
    {
        stack.pop();   // будет сгенерировано исключение
    }
    catch (const std::logic_error& e)
    {
        std::cerr << e.what() << '\n';
    }

    // помещаем несколько значений в стек
    stack.push(5);
    stack.push(2);
    stack.push(57);
    stack.push(10);
    stack.push(7);
    stack.push(8);
    stack.push(22);

    // выведем стек на экран
    std::cout << "1: ";
    std::cout << stack;
    std::cout << '\n';

    // проверяем метод pop, должно вернуться число, помещённое в стек
последним
    std::cout << "extract " << stack.pop() << '\n';
    std::cout << "2: ";
    std::cout << stack;
    std::cout << '\n';

    // и ещё раз
    std::cout << "extract " << stack.pop() << '\n';
    std::cout << "3: ";
    std::cout << stack;
    std::cout << '\n';

    // снова добавим в стек несколько чисел
    stack.push(89);
    stack.push(15);
    std::cout << "4: ";
    std::cout << stack;
    std::cout << '\n';

    return 0;
}
```

/

// Очередь

```
#include <iostream>

// Очередь - структура данных, работающая по принципу FIFO (First In, First
Out),
// т.е. элементы извлекаются из очереди в том же порядке, в котором
добавлялись.
// Ниже реализована очередь целых чисел на базе связного списка
class QueueInt
{
public:
    QueueInt() :
        head_(nullptr),
        tail_(nullptr)
    {}

    ~QueueInt()
    {
        while (head_)
        {
            Node* temp = head_;
            head_ = head_->next;
            delete temp;
        }
    }

    QueueInt(const QueueInt&);                  // конструктор копирования
    QueueInt(QueueInt&&) noexcept;              // конструктор перемещения
    QueueInt& operator=(const QueueInt&);       // перегруженный оператор
присваивания
    QueueInt& operator=(QueueInt&&) noexcept;   // перегруженный оператор
перемещающего присваивания

    void swap(QueueInt&) noexcept;              // обмен двух очередей
значениями (используется в операторе присваивания)

    void put(int);                              // добавить элемент в
очередь
    int take();                                 // получить элемент из
очереди

    friend std::ostream& operator<<(std::ostream& stream, const QueueInt&
queue);

private:
    struct Node           // элемент очереди
    {
```

```
        int data;              // данные
        Node* next;          // указатель на следующий элемент
    };

    Node* head_;           // указатель на начало очереди
    Node* tail_;           // указатель на конец очереди
};

// конструктор копирования
// кода объект класса содержит в себе какой-либо внешний ресурс (например,
динамически выделенную память),
// необходимо(!) реализовать (или запретить) конструктор копирования,
оператор присвивания и деструктор
// в C++11 желательно также определить конструктор перемещения и оператор
перемещающего присваивания
QueueInt::QueueInt(const QueueInt& queue) :
    head_(nullptr),
    tail_(nullptr)
{
    // собираем копию очереди внутри временного объекта, чтобы не допустить
утечки памяти
    // в случае возникновения исключения при выделени памяти под очередной
элемент
    QueueInt temp;
    Node* src = queue.head_;
    while (src)
    {
        temp.put(src->data);
        src = src->next;
    }
    swap(temp);
}

// конструктор перемещения
QueueInt::QueueInt(QueueInt&& queue) noexcept :
    head_(nullptr),
    tail_(nullptr)
{
    swap(queue);
}

// оператор копирующего присваивания
QueueInt& QueueInt::operator=(const QueueInt& queue)
{
    // выполняем присваивание только если адреса объектов отличаются
    if (this != &queue)
    {
        // создаём копию присваиваемой очереди
        QueueInt tempQueue(queue);
        // меняем содержимое временной копии и контекстного объекта
```

```
        swap(tempQueue);
    }
    // возвращаем ссылку на контекстный объект
    return *this;

    // временный объект tempQueue будет уничтожен при завершении метода
    // и его деструктор освободит память, занятую элементами изначальной
очереди;
    // такой приём написания оператора присваивания называется copy-and-swap
}

// оператор перемещающего присваивания
QueueInt& QueueInt::operator=(QueueInt&& queue) noexcept
{
    if (this != &queue)
    {
        // используем написанный ранее конструктор перемещения,
        // превращая queue в R-value с помощью функции move
        QueueInt tempQueue(std::move(queue));
        swap(tempQueue);
    }
    return *this;
}

// обмен значениями аргумента и контекстного объекта (просто меняем
указатели)
void QueueInt::swap(QueueInt& queue) noexcept
{
    std::swap(head_, queue.head_);
    std::swap(tail_, queue.tail_);
}

// добавление элемента в очередь
void QueueInt::put(int d)
{
    if (!head_)
    {
        head_ = tail_ = new Node;
    }
    else
    {
        tail_->next = new Node;
        tail_ = tail_->next;
    }
    tail_->data = d;
    tail_->next = nullptr;
}

// извлечение элемента из очереди
int QueueInt::take()
```

```
{
    if (!head_)
    {
        throw std::logic_error("Queue is empty");
    }
    int res = head_->data;
    Node* temp = head_;
    if (head_ == tail_)
    {
        head_ = tail_ = nullptr;
    }
    else
    {
        head_ = head_->next;
    }
    delete temp;
    return res;
}

// вывод очереди на экран
std::ostream& operator<<(std::ostream& stream, const QueueInt& queue)
{
    QueueInt::Node* temp = queue.head_;
    while (temp)
    {
        std::cout << temp->data << '\n';
        temp = temp->next;
    }
    return stream;
}

int main()
{
    QueueInt queue;

    // добавляем в очередь элементы
    queue.put(8);
    queue.put(15);
    queue.put(16);
    queue.put(100);
    queue.put(-8);

    // выводим
    std::cout << "Start queue:\n";
    std::cout << queue;
    std::cout << '\n';

    // проверяем метод take, должно вернуться число, помещённое в очередь
первым
    std::cout << "extract " << queue.take() << '\n';
```

```
    // ещё раз выводим
    std::cout << "Queue after first taking:\n";
    std::cout << queue;
    std::cout << '\n';

    QueueInt queue2 = queue;   // здесь вызывается конструктор копирования
(эквивалентно QueueInt queue2(queue);)

    // извлекаем ещё один элемент из исходнй очереди
    std::cout << "extract " << queue.take() << '\n';

    // и выводим, чтобы показать, что очереди разные
    std::cout << "After second taking\n";
    std::cout << "queue:\n";
    std::cout << queue;
    std::cout << "\nqueue2:\n";
    std::cout << queue;
    std::cout << '\n';

    queue2 = queue;    // а вот здесь уже вызывается оператор присваивания

    std::cout << "queue2 after assignment:\n";
    std::cout << queue;
    std::cout << '\n';

    // попробуйте самостоятельно написать примеры применения конструктора
перемещения
    // и оператора перемещающего присваивания

    // извлекаем оставшиеся элементы и один "лишний"
    try
    {
        std::cout << "extract " << queue.take() << '\n';
        std::cout << "extract " << queue.take() << '\n';
        std::cout << "extract " << queue.take() << '\n';
        std::cout << "extract " << queue.take() << '\n';    // здесь будет
исключение
    }
    catch (const std::logic_error& e)
    {
        std::cerr << e.what() << '\n';
    }

    return 0;
}
```

//Чтение и запись в файл

```
// Программа читает последовательность чисел из файла "infile.txt" (должен находиться в дирректории проекта)
// и записывает в два раза большие числа в файл "outfile.txt" (будет создан автоматически).
// Во входном файле числа разделяются символами-разделителями (пробел, табуляция, перенос строки).

#include <iostream>
#include <fstream>

int main()
{
    std::ifstream in("infile.txt");
    std::ofstream out("outfile.txt");
    if (!in.is_open() || !out.is_open())
    {
        std::cerr << "File can not be opened\n";
        return 1;
    }
    int n = 0;
    // здесь считывание производится прямо в условии цикла,
    // затем поток in приводится к типу bool (false - если при считывании были ошибки)
    while (in >> n)
    {
        out << n * 2 << '\n';
    }
    in.close();
    out.close();

    return 0;
}
```

// Простой пример работы с виртуальными методами

```
#include <iostream>

// базовый класс животного
class Animal
{
public:
  // классы, содержащие виртуальные методы, должны также иметь виртуальный деструктор
  // для корректного разрушения объектов производных классов через указатель базового класса
  virtual ~Animal() = default;

  // метод, печатающий на экран звук, который издает животное
  // попробуйте убрать слово virtual и посмотрите, как изменится выолнение программы
  virtual void getSound() const
  {
    std::cout << "Unknown sound\n";
  }
};

class Bull : public Animal
{
public:
  // метод getSound переопределяется в производных классах
  void getSound() const
  {
    std::cout << "Mmooooo\n";
  }
};

class Dog : public Animal
{
public:
  void getSound() const
  {
    std::cout << "Haw-Haw\n";
  }
};

int main()
{
  // массив указателей на объекты базового(!) класса
  const int ANIMAL_COUNT = 3;
  Animal* animals[ANIMAL_COUNT]{ nullptr };

  try
  {
```

```
    // массив с динамически созданными объектами производных классов
    animals[0] = new Bull;
    animals[1] = new Dog;
    animals[2] = new Bull;

    for (int i = 0; i < ANIMAL_COUNT; ++i)
    {
      animals[i]->getSound();
    }
  }
  catch (const std::exception& e)
  {
    std::cout << "Exception: " << e.what() << '\n';
  }

  for (int i = 0; i < ANIMAL_COUNT; ++i)
  {
    delete animals[i];
  }

  return 0;
}
```

// Пример абстрактного класса

```
#include <iostream>
#include <string>

// абстрактный класс, содержащий имя преподавателя и количество его статей
// объекты этого класса в явном виде не могут быть созданы
class Teacher
{
public:
  Teacher(const std::string& name) :   // конструктор вызовем в конструкторе
потомка
    name_(name),
    publicationCount_(0)
  {}

  // классы, содержащие виртуальные методы, должны также иметь виртуальный
деструктор
  // для корректного разрушения объектов производных классов через указатель
базового класса
  virtual ~Teacher() = default;

  const std::string& getName() const
  {
    return name_;
  }

  int getPublicationCount() const
  {
    return publicationCount_;
  }

  void setPublicationCount(int count)
  {
    publicationCount_ = count;
  }

  // расчёт повышающего коэффициента к зарплате;
  // чистая виртуальная функция, именно она делает класс абстрактным
  virtual double getCoefficient() const = 0;

protected:
  std::string name_;
  int publicationCount_;               // количество статей
};

// Ассистент
class Assistant : public Teacher
{
public:
```

```
  Assistant(const std::string& name) :
    Teacher(name)                       // в конструкторе вызываем конструктор
базового класса
  {}

  // на зарплатный коэффициент ассистента влияет только количество статей;
  // ключевое слово override (C++11) показывает, что метод переопределён
  // и не дает ошибиться, например, при изменении сигнатуры метода в базовом
классе
  double getCoefficient() const override
  {
    const double NORMAL = 1.0;
    const double HIGH = 1.3;

    return publicationCount_ < PUBLICATION_BARRIER ?
      NORMAL :
      HIGH;
  }

private:
  static const int PUBLICATION_BARRIER = 3;
};

// Профессор
class Professor : public Teacher
{
public:
  Professor(const std::string& name) :
    Teacher(name),
    postgraduateCount_(0)
  {}

  int getPostgraduateCount() const
  {
    return postgraduateCount_;
  }

  void setPostgraduateCount(int count)
  {
    postgraduateCount_ = count;
  }

  // профессор может получить повышенный зарплатный коэффициент,
  // если у него достаточное количество аспирантов
  double getCoefficient() const override
  {
    const double NORMAL = 1.0;
    const double HIGH = 1.3;
    const double HIGHEST = 1.6;
```

```
    if (publicationCount_ < PUBLICATION_BARRIER)
    {
      return NORMAL;
    }
    else if (postgraduateCount_ < POSTGRADUATE_BARRIER)
    {
      return HIGH;
    }
    else
    {
      return HIGHEST;
    }
  }

private:
  // необходимое количество статей для профессора выше, чем для ассистента
  static const int PUBLICATION_BARRIER = 5;
  static const int POSTGRADUATE_BARRIER = 2;

  int postgraduateCount_;              // количество аспирантов
};

// функция выводит информацию о преподавателе,
// в качестве параметра принимается ссылка на объект базового(!) типа
void printInfo(std::ostream& stream, const Teacher& teacher)
{
  stream << teacher.getName() << " is ";
  // с помощью оператора dynamic_cast можно определить, кем на самом деле
является преподаватель
  if (dynamic_cast<const Professor*>(&teacher) != nullptr)
  {
    stream << "a professor ";
  }
  else if (dynamic_cast<const Assistant*>(&teacher) != nullptr)
  {
    stream << "an assistant ";
  }
  stream << "with salary coefficient " << teacher.getCoefficient() << '\n';
}

int main()
{
  // попробуйте создать объект базового класса - возникнет ошибка, т.к. класс
абстрактный

  Professor professor1("Boris Ivanov");
  professor1.setPublicationCount(6);

  Professor professor2("Alexander Stepanov");
  professor2.setPublicationCount(7);
```

```
  professor2.setPostgraduateCount(3);

  Assistant assistant1("Ivan Petrov");
  assistant1.setPublicationCount(2);

  printInfo(std::cout, professor1);
  printInfo(std::cout, professor2);
  printInfo(std::cout, assistant1);

  return 0;
}
```

// Шаблон коллекции

```
#include <iostream>

// классы исключений
class StackEmptyException : public std::exception
{
public:
  const char* what() const noexcept override
  {
    return "Stack is already empty";
  }
};

class StackOverflowException : public std::exception
{
public:
  const char* what() const noexcept override
  {
    return "Stack overflow";
  }
};

// Шаблон абстрактного класса Stack, способный работать со значениями любых типов.
// Слова template и class часть языка, T - имя типа (может быть названо как угодно)
template <class T>
class Stack
{
public:
  Stack() = default;
  virtual ~Stack() = default;              // виртуальный деструктор по умолчанию

  // метод push принимает значение типа T
  virtual void push(const T& data) = 0;
  // метод pop удаляет значение с вершины стека;
  virtual void pop() = 0;
  // метод top возвращает значение с вершины стека, но не удаляет его;
  virtual const T& top() const = 0;

  // попробуйте реализовать методы копирования/перемещения самостоятельно
  Stack(const Stack&) = delete;
  Stack(Stack&&) = delete;
  Stack& operator=(const Stack&) = delete;
  Stack& operator=(Stack&&) = delete;
};

// Реализция стека на основе связного списка
```

```
template <class T>
class ListStack : public Stack<T>
{
public:
  ListStack() :
    head_(nullptr)
  {}
  ~ListStack() override;

  void push(const T& data) override;
  void pop() override;
  const T& top() const override;

private:
  struct Node
  {
    T data;          // вместо конкретного типа, data имеет шаблонный тип T
    Node* next;

    Node(T data, Node* next = nullptr) :
      data(data),
      next(next)
    {}
  };

  Node* head_;
};

template <class T>
ListStack<T>::~ListStack()
{
  Node* temp = head_;
  while (temp)
  {
    head_ = head_->next;
    delete temp;
    temp = head_;
  }
}

template <class T>
void ListStack<T>::push(const T& data)
{
  Node* newNode = new Node(data, head_);
  head_ = newNode;
}

template <class T>
void ListStack<T>::pop()
{
```

```
  if (head_ == nullptr)
  {
    throw StackEmptyException();
  }
  Node* temp = head_;
  head_ = head_->next;
  delete temp;
}

template <class T>
const T& ListStack<T>::top() const
{
  if (head_ == nullptr)
  {
    throw StackEmptyException();
  }
  return head_->data;
}

// Реализция стека на основе массива
template <class T>
class ArrayStack : public Stack<T>
{
public:
  ArrayStack(std::size_t maxSize) :
    values_(new T[maxSize]),
    maxSize_(maxSize),
    size_(0)
  {}
  ~ArrayStack() override
  {
    delete[] values_;
  }

  void push(const T& data) override;
  void pop() override;
  const T& top() const override;

private:
  T* values_;             // указатель на массив с данными
  std::size_t maxSize_;   // количество выделенной памяти
  std::size_t size_;      // текущий размер
};

template <class T>
void ArrayStack<T>::push(const T& data)
{
  if (size_ == maxSize_)
  {
    throw StackOverflowException();
```

```
  }
  values_[size_] = data;
  ++size_;
}

template <class T>
void ArrayStack<T>::pop()
{
  if (size_ == 0)
  {
    throw StackEmptyException();
  }
  --size_;
}

template <class T>
const T& ArrayStack<T>::top() const
{
  if (size_ == 0)
  {
    throw StackEmptyException();
  }
  return values_[size_ - 1];
}

// Эта структура нужна только для примера,
// далее несколько её экземпляров будет помещено в стек
struct Example
{
  int i;
  double d;
  Example(int i, double d) :
    i(i),
    d(d)
  {}
};

int main()
{
  Stack<double>* doubleStack = new ListStack<double>;     //стек, хранящий
значения типа double
  try
  {

    doubleStack->push(5.45);
    doubleStack->push(10.782);
    std::cout << doubleStack->top() << '\n';
    doubleStack->pop();
    std::cout << doubleStack->top() << '\n';
    doubleStack->pop();
```

```
    std::cout << doubleStack->top() << '\n';
    doubleStack->pop();
  }
  catch (const std::exception& e)
  {
    std::cerr << e.what() << '\n';
  }

  std::cout << '\n';

  delete doubleStack;

  Stack<char>* charStack = new ArrayStack<char>(3);       //стек, хранящий
значения типа char
  try
  {
    charStack->push('v');
    charStack->push('f');
    charStack->push('r');
    charStack->push('d');
  }
  catch (const std::exception& e)
  {
    std::cerr << e.what() << '\n';
  }

  try
  {
    std::cout << charStack->top() << '\n';
    charStack->pop();
    std::cout << charStack->top() << '\n';
    charStack->pop();
    std::cout << charStack->top() << '\n';
    charStack->pop();
    std::cout << charStack->top() << '\n';
    charStack->pop();
  }
  catch (const std::exception& e)
  {
    std::cerr << e.what() << '\n';
  }

  std::cout << '\n';

  delete charStack;

  try
  {
    ListStack<Example> exampleStack;       // стек, хранящий экземпляры
структуры Example
```

```
    exampleStack.push(Example(5, 5.5));
    exampleStack.push(Example(7, 7.7));

    std::cout << exampleStack.top().i << '\n';
    exampleStack.pop();

    Example temp = exampleStack.top();
    std::cout << temp.d << '\n';
  }
  catch (const std::exception& e)
  {
    std::cerr << e.what() << '\n';
  }

  return 0;
}

// Шаблоны позволяют использовать обобщённое программирование, реализуя
алгоритмы и структуры данных без привязки
// к конкретным типам данных. Кроме шаблонов классов, можно также создавать и
шаблоны отдельных функций.

// Можно использовать и несколько шаблонных типов в одном шаблоне:

// template <class T, class A, class B, ...>

// В момент компиляции шаблоны заменяются на конкретные реализации классов и
функций.
// Причём, будет создано столько реализаций класса, сколько раз шаблон был
использован с различными типами.

// В состав языка C++ входит, так называемая, стандартная библиотека шаблонов
(STL)
// STL содержит реализации наиболее часто используемых контейнеров
// (например vector<T> - динамический массив, list<T> - список, map<T1, T2> -
ассоциативный массив),
// а так же итераторы для доступа к элементам контейнеров и алгоритмы для
обработки данных
// (например, sort(), count(), find() и т.д.)
```