КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра системного анализа и информационных технологий

М.П. ДЕНИСОВ

ЛАБОРАТОРНЫЕ РАБОТЫ ПО ОСНОВАМ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++. ЧАСТЬ 1

Учебно-методическое пособие

Казань – 2021

УДК 004.822
ББК 32.973

Принято на заседании кафедры системного анализа и информационных технологий Казанского (Приволжского) федерального университета
Протокол № 4 от 09 декабря 2020 года

Рецензент:
старший преподаватель кафедры САиИТ института ИВМиИТ КФУ **Р.Р. Тагиров**

Денисов М.П.
Лабораторные работы по основам программирования на языке с++. Часть 1 / М.П. Денисов. – Казань: Казан. ун-т, 2021. – 64 с.

Учебно-методическое пособие предназначено для студентов первого курса, первого семестра, обучающихся в бакалавриате по направлению «Информационная безопасность». Предполагается использование пособия для самостоятельной дополнительной работы студентов по проблемным для них темам в рамках дисциплины «Основы программирования».

© Денисов М.П., 2021
© Казанский университет, 2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 4

ЛАБОРАТОРНАЯ РАБОТА 1. Типовая структура программы. Вывод текста в консоль. 5

ЛАБОРАТОРНАЯ РАБОТА 2. Последовательность действий. Переменные. Типы данных. Ввод с клавиатуры. 14

ЛАБОРАТОРНАЯ РАБОТА 3. Выражения. Выполнение в зависимости от условий. 28

ЛАБОРАТОРНАЯ РАБОТА 4. Многократное выполнение. 46

СПИСОК ЛИТЕРАТУРЫ 63

ВВЕДЕНИЕ

В рамках курса «основы программирования» студент осваивает типовые элементы алгоритмов, типовые алгоритмы, основные конструкции определенного языка программирования, получает начальные навыки решения прикладных задач.

Все студенты имеют разный уровень подготовки и некоторым для понимания определенных тем требуется их более подробное описание.

Данное пособие предназначено для самостоятельной работы по таким темам в дополнение к основной работе в рамках курса и в дополнение к индивидуальным ответам преподавателя на вопросы по таким темам.

Пособие состоит из нескольких частей. Каждая часть состоит из нескольких лабораторных работ. Каждая работа состоит из теоретической и практической частей.

В рамках теоретической части подробно разбирается решение одной или несколько задач, формирование алгоритма для их решения и реализация программы. При необходимости приводится подробное объяснение необходимых конструкций языка программирования. Также при необходимости приводится процесс выполнения полученной программы или ее фрагментов.

В рамках практической части предлагается одна или несколько задач для закрепления материала.

Предполагается, что студент будет прорабатывать только те работы и их части, которые связаны с проблемными для него темами.

Также возможно расхождение с традиционной терминологией, если это требовалось для упрощения объяснений.

В рамках данной части пособия приводится 4 лабораторные работы.

В работе 1 «Типовая структура программы. Вывод текста в консоль.» подробно описывается типовая структура простой программы на языке с++ и ее эле-

менты. Также разбирается поток вывода в консоль cout и основные проблемы, которые могут возникнуть при работе с ним.

В работе 2 «Последовательность действий. Переменные. Типы данных. Ввод с клавиатуры» разбирается процесс создания алгоритма для решения задачи, требующей только последовательного выполнения действий. Также рассматриваются механизмы языка c++, позволяющие хранить промежуточную информацию. Рассматривается поток ввода cin, принципы его работы и основные проблемы, возникающие при его использовании.

В работе 3 «Выражения. Выполнение в зависимости от условий.» разбирается процесс создания алгоритма для решения задачи, требующей выполнение различных действий при различных условиях. Также разбирается работа выражений в языке c++ (операторы, зависимость результата от типа, возвращаемые значения, приоритетность выполнения и т.д.) и проблемные моменты, с которыми студент может столкнуться на данном этапе.

В работе 4 «Многократное выполнение.» разбирается процесс создания алгоритма для решения задачи, требующей многократного повторения выполнений одинаковых действий. Разбираются циклы с предусловием, с постусловием, «for» и основные проблемы, которые могут возникнуть при их применении. Также разбираются короткие записи арифметических операций и оператор «,».

ЛАБОРАТОРНАЯ РАБОТА 1. ТИПОВАЯ СТРУКТУРА ПРОГРАММЫ. ВЫВОД ТЕКСТА В КОНСОЛЬ.

Язык с++ [1,2] является компилируемым языком. Это означает, что программа, написанная на этом языке, должна быть преобразована специальными средствами в исполняемый файл (.exe), который в свою очередь содержит инструкции, понятные операционной системе, для которой эта программа была написана. Такие средства называются компиляторами (например, gcc).

Для удобства разработки текстовый редактор кода, компилятор, средства отладки и т.д. могут быть объединены в единое программное обеспечение, которое называют интегрированной средой разработки (IDE). Удобство заключается в автоматизации большинства настроек компилятора, подсказках во время написания кода, графическом интерфейсе, возможностях пошагового выполнения программы и т.д. Примерами таких сред являются: Visual Studio [2], Dev C++.

На данный момент, существуют также и web варианты. Они, как правило проигрывают в возможностях организации структуры программы, подсказках, отладке и сложной сборки, а также позволяют запускать только консольные приложения. Однако, для простых задач они позволяют производить преобразование кода и запуск получившейся программы «в один клик» и не требуют установки и настройки программного обеспечения на свой компьютер.

В рамках данного пособия будет использоваться web компилятор https://www.onlinegdb.com/online_c++_compiler (рис. 1). Для этого необходимо поместить код в соответствующее поле ввода (рис. 1 а) и нажать кнопку «Run» (рис. 1 б). После этого внизу окна появится консоль (рис. 1 в), где будет отображаться вывод программы, а также осуществляться пользовательский ввод. Текст, выведенный в консоли зеленым цветом, является отладочным и не будет выводиться, если преобразовать код в конечную программу (сформировать .exe файл). Если

программа будет выполняться слишком долго, то выполнение можно остановить принудительно с использованием кнопки «Stop» (рис. 1 г).

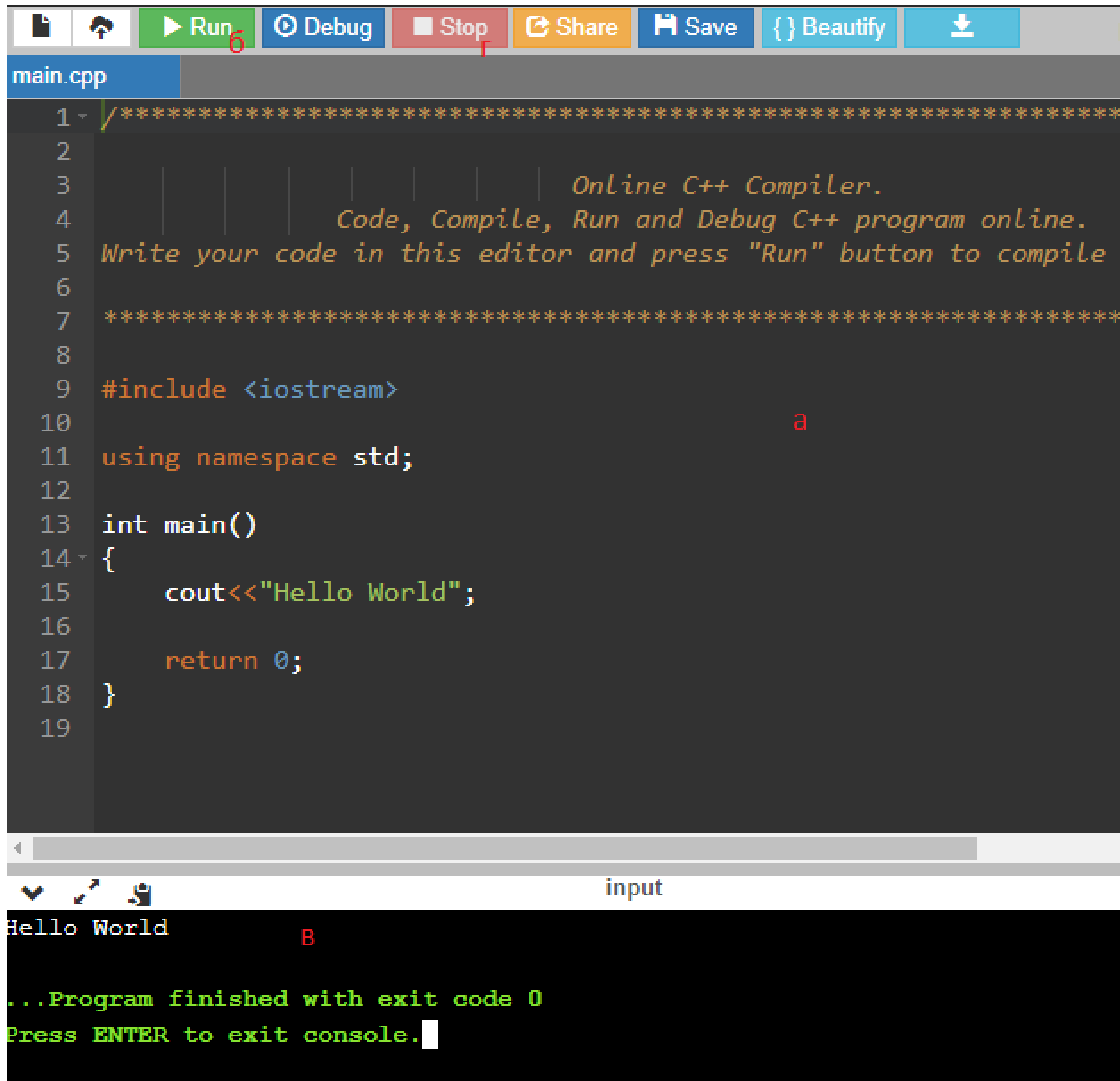

Рис. 1 Онлайн компилятор OnlineGDB (а – поле для ввода кода программы; б – кнопка сборки и запуска программы; в – консоль; г – кнопка завершения работы программы)

Рассмотрим структуру программы на языке с++ на типичном примере «Hello World». Задача: вывести в консоль заранее заданный текст («Hello world»).

```
#include <iostream>
```

```
using namespace std;
int main()
{
   cout<<"Hello World";
   return 0;
}
```

Основа каждой программы – это набор упорядоченных команд, которые необходимо выполнить для решения поставленной задачи. В случае описываемой программы необходима была только одна команда: вывести на экран строку «Hello World». И на языке с++ она записывается: cout << "Hello World";.

Остальные строки программы являются стандартными для большинства задач данного пособия и будут составлять каркас будущих программ.

Каждая команда должна оканчиваться специальным знаком «;». Именно по этому знаку происходит разделение кода на команды (переносы строк между командами не обязательны, но желательны для поддержания «читаемости» кода). Любое количество команд может быть объединено в одну логическую единицу («составную команду»). Для этого необходимо обернуть их в фигурные скобки «{}».

Такой составной команде можно дать наименование и вызывать ее выполнение из другой части кода. Такой механизм называется «функции» и подробнее будет рассмотрен позднее. Однако именно такая функция вызывается, если запустить выполнение программы. Вызывается функция с определенным именем: main.

Полностью ее сигнатура (имя и дополнительные описания; все, кроме тела – «составной команды») описывается следующим образом: int main(). Где main – это само имя, а int – тип возвращаемых данных – целое число (в результате работы данной функции должно быть определено некоторое значение, являющееся целым

числом). В случае функции main результатом является код ошибки. При этом число 0 означает, что программа закончила работу верно (ошибок не было), а любое другое число означает номер ошибки, которая произошла.

В круглых скобках после наименования функции перечисляются аргументы – параметры, которые необходимы для работы функции. Например, для вычисления произведения 2 чисел нужно знать эти числа. Тогда они и будут аргументами. Подробнее этот механизм будет описан позднее, а на данный момент главная функция не требует аргументов и потому содержимое скобок пусто.

После описания сигнатуры следует тело функции, описанное как «составная команда», т.е. как набор команд (или одна команда) в фигурных скобках.

Т.к. при описании сигнатуры заявлен тип возвращаемых данных – целое число, то необходимо указать что является результатом работы функции. В данном случае результат – номер ошибки, считаем, что программа всегда завершается успешно, т.е. результат – 0. Результат указывается с использованием команды «return 0;». Где вместо 0 в общем случае может быть любое выражение. Эта команда также заканчивает выполнение функции. А в случае главной функции заканчивает выполнение программы. Все что идет после этой команды исполнено не будет.

Разберем подробнее строку кода, отвечающую за вывод в консоль: «cout << "Hello world";».

Представим, что есть некоторое виртуальное представление сущности «консоль». Также есть 2 очереди: первая – элементы, которые ожидают вывода в консоль; вторая – элементы, которые были введены пользователем в консоль с клавиатуры и ожидают обработки.

Первая очередь именуется – cout, вторая – cin. В данной работе работа будет вестись только с cout, cin будет рассмотрен позднее. Далее обе очереди для упрощения автор часто будет называть «консоль».

«<<» – в случае cout является оператором вывода (2 знака меньше подряд).

После этого оператора могут идти любые элементы, которые позволено выводить в консоль (для которых определено как именно происходит вывод в консоль). В нашем случае это строка (любая последовательность символов, записанная в двойных кавычках). Для упрощения можно запоминать следующим образом: «консоль (cout) вывести(<<) строка("Hello world")».

«<<» является бинарным оператором (работает с 2 аргументами). Любая операция имеет результат (например, для операции «+» и выражения «2 + 2» результат будет целым числом 4). Это позволяет записывать сложные выражения (например, «(2 + 2) + 2»). Операция «<<» работает точно также, а в качестве результата всегда возвращает саму сущность «cout». Это означает, что можно записывать такие составные выражения как: «((cout << "Hello") << " ") << "World";». При этом так как все операторы одинаковые и имеют одинаковый приоритет, то как и в случае с «(2+2)+2 = 2+2+2» скобки можно опустить и записать «cout << "Hello" << " " << "World";».

Описанные выше команды преобразуются в процессе компиляции в понятные операционной системе инструкции для выполнения. Но, в операционной системе не описана сущность «cout» и оператор «<<» для различных элементов, доступных в языке «c++». Также, мы не описываем эту сущность сами. Тем не менее после компиляции программа работает. Это связано с тем, что все это уже описано разработчиками языка «c++» и хранится в отдельных файлах, которые можно использовать в своих программах (библиотеках).

Для использования библиотеки необходимо ее подключить. Это делается с использование команды «#include <iostream>». Где вместо «iostream» может быть имя другой библиотеки. Стоит заметить, что эта команда является особенной. Во-первых, она находится за пределами функции main и любых других функций, во-вторых, она не оканчивается «;». Это связано с тем, что данная команда является директивой препроцессора.

Все команды, которые мы рассматривали ранее так или иначе выполнялись в процессе выполнения программы. Т.е. это были команды самой программы. Команды, начинающиеся с «#» являются директивами препроцессора (командами для компилятора). Эти команды выполняются на одном из этапов сборки программы. И в конечной программе их уже нет. Так директива «#include» помещает на место, где она записана, код из указанного файла. Так можно разбивать свою программу на множество файлов, если код становится слишком объемным. При этом если имя библиотеки записано в треугольные скобки («#include <iostream>»), то файл ищется в стандартных файлах языка с++, а если в двойных кавычках (#include "myfile.cpp"), то ищется в папке с файлом, где эта директива указана.

Разбиение программы на файлы в рамках данного пособия производиться не будет. Стандартные библиотеки, которые будут использоваться в ближайших работах:

iostream – библиотека для работы с очередями (потоками) ввода/вывода (input output stream; включает cin, cout).

math.h – библиотека для работы с математическими функциями (sqrt – квадратный корень, abs – модуль)

В рамках приведенной типовой структуры программы осталось описать только строку «using namespace std;». Эта строка необходима для сокращенного написания имен стандартных сущностей и функций (таких как cin, cout).

В языке с++ предусмотрена возможность объявлять собственные сущности (в частности, переменные, которые будут описаны в следующих работах). При этом, имена для сущностей, которые выбрал разработчик, могут быть уже заняты стандартными сущностями. Для того, чтобы такого не происходило, в языке предусмотрена возможность разбиения сущностей на группы (пространства имен). Каждая группа также именуется и полное имя сущности записывается как «имя_группы::имя_сущности» (разделитель между именем группы и именем сущности два двоеточия). Сущности могут быть объявлены и без группы.

Большинство стандартных сущностей находятся в пространстве имен «std». Например, полные имена для cin, cout – std::cin, std::cout. Конструкция «using namespace имя_пространства;» позволяет указать системе, что если используется сущность и ее объявление не найдено в основном файле, то следует искать ее также в указанном пространстве имен. В приведенном примере объявление «cout» сначала будет искаться в основном файле, а затем в пространстве имен «std».

Стоит отметить, что «using namespace std;» позволяет значительно «облегчить» код и улучшить его читаемость. Поэтому эта конструкция будет постоянно использоваться в данном пособии. Но, при увеличении объема программы и количества используемых внешних библиотек, бывает целесообразно использовать полные имена сущностей, чтобы избегать непредсказуемого поведения при одинаковых именах сущностей в разных библиотеках.

При описании в коде выводимого текста открывающаяся и закрывающаяся двойные кавычки должны находиться на одной строке. Для вывода переноса строки и подобного используются специальные символы. Каждый такой символ начинается с обратного слеша «\». Примеры специальных символов: «\n» – перенос строки, «\t» – табуляция. Для вывода самого обратного слеша его необходимо записать дважды «\\». Также для обозначения переноса строки можно использовать специальную сущность из пространства имен «std» – «endl».

Например, для вывода «Hello» и «World» на разных строках код может выглядеть следующим образом:

```
cout << "Hello\nWorld";
cout << "Hello" << endl << "World";
```

Также стоит отметить конструкции для пояснительных комментариев в коде. Это необходимо как для себя самого, чтобы не забывать в дальнейшем что делает тот или иной фрагмент кода, так и для других разработчиков, которые могут этот код в дальнейшем использовать. Всего таких конструкций две.

«//текст комментария» – строчный комментарий. Все что записано после двойного слеша «//» и до конца строки игнорируется и удаляется на стадии компиляции. Данный тип комментариев чаще всего используется для пояснений действий, производимых в конкретной строке:

```
cout << "Hello World" << endl; //выводим на экран строку
```

«/*текст комментария*/» – блочный комментарий. Все что записано после слеш и звездочки «/*» и до звездочки и слеша «*/» игнорируется и удаляется на стадии компиляции. Данный тип комментариев чаще всего используется для пояснения логики работы фрагментов программы или всей программы:

```
/* Данная программа
выводит на экран заранее заданный текст
В данном случае это строка «Hello World»
*/
int main() {
….
}
```

Задания

Написать программу, которая выведет в консоль ситуацию из игры в крестики нолики следующего вида:

```
X | O | X
O | X | X
O | X | O
```

При этом сделать разные варианты реализации:

1. С использованием конструкции using namespace std;
2. Без использования конструкции using namespace std;
3. С использованием endl
4. Без использования endl

5. Чтобы в коде весь вывод был записан в одну строку (одной командой)
6. Чтобы в коде также читалось указанное изображение. Т.е. чтобы крестики и нолики, находящиеся друг под другом на изображении, также находились друг под другом в нужной позиции и в коде.

Ввод и вывод в консоль здесь и далее (на протяжении всего пособия) осуществлять только латинскими символами (английский алфавит). Русский в рамках этого пособия использоваться не будет в связи с необходимостью изучения дополнительных механизмов.

ЛАБОРАТОРНАЯ РАБОТА 2. ПОСЛЕДОВАТЕЛЬНОСТЬ ДЕЙСТВИЙ. ПЕРЕМЕННЫЕ. ТИПЫ ДАННЫХ. ВВОД С КЛАВИАТУРЫ.

В рамках данной работы будет решена задача расчета объема прямоугольного параллелепипеда (многогранник с шестью гранями, каждая из которых прямоугольник). Измерения (длины трех ребер, принадлежащих одной вершине) будут запрошены у пользователя и введены с клавиатуры. Результат будет выведен в консоль.

Для реализации программы, решающей задачу, необходимо:

1. Выделить все сущности, которые необходимы для решения задачи.
2. Составить неформальное описание последовательности операций, которых достаточно для решения задачи. При этом возможно повторение шага 1.
3. Формализовать полученные на этапах 1-2 сущности и операции на языке c++ в соответствии со структурой программы, рассмотренной в работе 1.

Результат второго этапа будет называться алгоритмом. Стоит также отметить существование теоремы, в соответствии с которой любой алгоритм может быть представлен только при помощи 3 структур управления: последовательное выполнение операций, выполнение операций в зависимости от условий, многократное повторение выполнения операций. [3]

Будем рассматривать эти структуры управления постепенно. В данной работе будет использоваться только последовательное выполнение операций.

Выделим сущности, которые точно понадобятся для решения задачи: a,b,c – измерения, V – объем.

Далее составим последовательность операций, необходимых для решения задачи и представим их в графическом виде (в виде блок-схемы), где каждая операция будет представлена в виде блока (плоской геометрической фигуры определенного вида), а порядок выполнения в виде стрелок, соединяющих эти блоки.

Применяемые в данной работе виды блоков приведены на рис. 2 (в дальнейшем список будет пополняться).

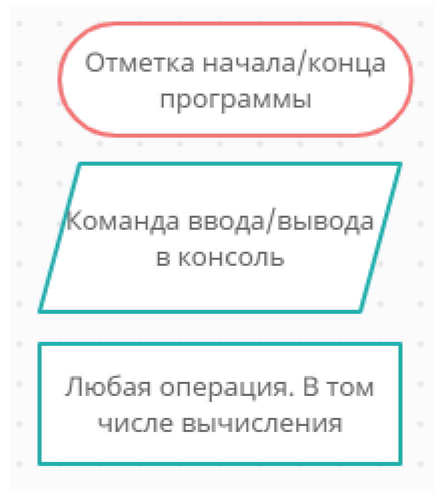

Рис. 2 Элементы блок-схемы, которые будут использоваться в текущей работе

Операции, требуемые для получения решения данной задачи достаточно просты (блок схема для этого варианта приведена на рис. 3а):

Ввести a, b, c

Вычислить V = a * b * c

Вывести V на экран

Необходимо также выводить пояснения при вводе данных, чтобы пользователь понимал, какие данные запрашивает программа. Этого можно добиться если вывести в консоль строку с пояснением до осуществления ввода. Можно выводить пояснение перед вводом каждого параметра (рис. 3б) или единожды перед вводом всех трёх параметров (рис. 3в).

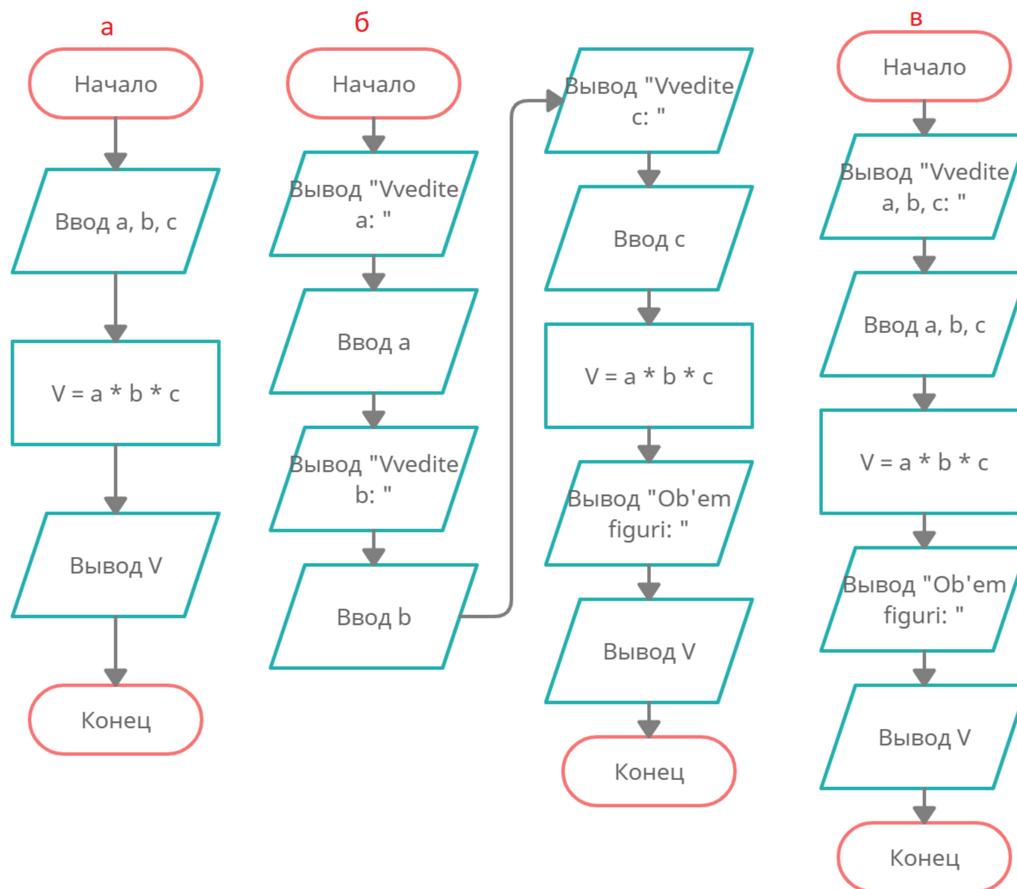

Рис. 3 Блок схема алгоритма решения задачи о расчете площади прямоугольного параллелепипеда

Далее необходимо выполнить третий этап и составить программу на языке с++. В прошлой лабораторной работе была разобрана структура типовой программы на языке с++. Повторим ее ниже. Реализованный алгоритм будет записан вместо троеточия.

```
//подключаем библиотеку для работы с вводом/выводом консоли
//также эта библиотека подключит многие стандартные сущности
//языка с++, т.к. зависит от них
#include <iostream>
/*эта строка позволит писать вместо std::cout просто cout
объявление всех используемых сущностей будут искаться
```

```
сначала в общей области, а затем в пространстве имен std*/
using namespace std;

//основная функция (точка входа)
int main()
{
   … //здесь будет реализация нашего алгоритма

   //если дошли до конца, то программа закончилась успешно
   //возвращаем код ошибки 0 (успех)
   return 0;
}
```

В соответствии с алгоритмом сначала вводятся измерения a, b, c, затем вычисляется V. Значит значения измерений где-то должны сохраняться минимум до того времени пока они не понадобятся. Аналогично значение V должно сохраняться до момента пока не будет произведен вывод на экран. Также удобно обращаться к данным значениям непосредственно по их именам. Для подобных действий в языке c++ есть механизм переменных. Это такие сущности, которые имеют тип, наименование и значение. По имени сущности можно получать доступ к значению и изменять его. Тип необходимо указывать для того, чтобы компилятор «понимал» каким образом работать с переменной.

Каждая переменная в конечном итоге представляется в оперативной памяти в двоичном виде (в виде 0 и 1) и в таком же виде обрабатывается. Так, например, число 5 в двоичном виде – это 101 ($1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 5$). Каждый 0 и 1 – это бит, 8 бит – это байт, 1024 байта – килобайт и т.д. Например, целое число со знаком (может быть отрицательным и положительным), часто занимает 4 байта (т.е. последовательность из 32 0 и 1). При этом первый элемент необходим для хране-

ния знака. Подробное описание хранения различных типов данных в памяти выходит за пределы данного пособия, подробнее со системами счисления можно ознакомиться, например в [4], о представлении различных типов данных в двоичном виде на примере вещественных чисел можно ознакомиться, например в [5].

Данное отступление было необходимо, чтобы побудить читателя подробнее ознакомиться с указанными механизмами. Это бывает важно для планирования программы и понимания ее работы. Как минимум, от выбранного типа данных зависит: минимальное и максимальное число, которое сможет хранить переменная; объем оперативной памяти, которая понадобится для хранения значения.

Также, для объяснения важности знания этих механизмов, рассмотрим следующий пример: мы создаем переменную, которая хранит беззнаковое целое число и занимает 1 байт. Соответственно значение переменной будет представлено в виде восьми нулей и единиц. Это значит, что она сможет хранить числа от 0 (00000000) до 255 (11111111). Пусть переменная имеет имя k. Что же будет, если исполнить для такой переменной следующие действия?

Сохранить в переменной значение 255. Сохранить в переменной ее старое значение (255), увеличенное на 2 (т.е. 257). Вывести на экран значение, хранящееся в переменной.

```
k = 255; k = k + 2; cout << k;
```

257 в переменной храниться не может, т.к. 8 бит не могут представить данное число. При этом ошибки также не будет.

255 (11111111) + 2 (00000010) = 257 (100000001)

100000001 содержит 9 элементов. Под переменную выделено 8 байт. Соответственно первый элемент будет отброшен, а оставшиеся 8 сохранены: 00000001 (1). И на экран будет выведено 1. Это явление называется переполнением и может возникать достаточно часто.

Теперь рассмотрим синтаксис (формальный язык с помощью которого записываются различные конструкции в коде программы) объявления переменных в с++: «тип_данных имя_переменной;». Например, «int a;»

Можно объявить сразу несколько переменных одного типа. Для этого нужно перечислить имена переменных через запятую: «int a, b, c;»

Некоторым переменным можно сразу присвоить определенные значения: «int x = 0, y = 0, z;». Вместо конкретных значений могут быть использованы любые выражения.

Стоит отметить, что объявление переменной также является командой и должно идти до любого использования этой переменной.

Объявление может быть внутри главной (или любой другой) функции – локальная переменная или за ее пределами (аналогично команде using namespace std;) – глобальная переменная. Мы не будем использовать глобальные переменные в ближайших работах. При увеличении размера программы, глобальные переменные могут создавать проблемы и непредсказуемое поведение, потому применять их следует только при необходимости.

Также любая переменная может быть объявлена константой, если указать до типа ключевое слово «const»: «const int t = 0;». Константа может получить значение только один раз. В дальнейшем по ее имени можно получать значение, но любая попытка изменения значения вызовет ошибку.

Типы переменных, используемые в данном пособии, приведены в таблице 1.

Таблица 1

Некоторые типы данных в языке с++

Название типа	Размер (байт)	Диапазон
int	4	-2147483648 … 2147483647
unsigned int	4	0 … 4294967295
long	4	-2147483648 … 2147483647
short	2	-32768 … 32767

unsigned short	2	0 … 65535
bool	1	0 или 1 («нет» или «да»; false или true)
char	1	-128...127
unsigned char	1	0…255
float	4	$-1.175*10^{38}…3.402*10^{38}$
double	8	$-2.225*10^{308}…1.797*10^{308}$

При этом необходимо понимать, что размер и диапазон в редких случаях могут отличаться в зависимости от используемого компилятора и целевой операционной системы. Например, char может изменяться в диапазоне 0…255, int может занимать 2 байта.

Если в процессе выполнения необходимо узнать точное количество байт необходимо вызывать sizeof(тип). Например, sizeof(int), sizeof(double).

Для определения границ целочисленных типов можно использовать константы из библиотеки «limits.h», значения которых понятно по их названиям: INT_MIN, INT_MAX, CHAR_MIN, CHAR_MAX. Для указания максимальных значений беззнаковых типов (которые начинаются с unsigned) используются UINT_MAX, UCHAR_MAX (минимальные значения, очевидно, 0).

Для определения границ типов для представления чисел с плавающей точкой (дробные) можно использовать константы из «float.h»: FLT_MIN, FLT_MAX, DBL_MIN, DBL_MAX.

Также, при работе с числами с плавающей точкой, в связи с особенностями их представления [5], можно увидеть записи, похожие на 1.34e-23. Такая запись означает, что число равно $-1.34*10^{23}$. Также стоит помнить, что такие числа до определенного значения хранятся как точные значения, а после них как приближенные. Это значит, что после этого значения даже предположительно одинаковые числа могут быть не равны. Например, если объявить переменные float t1 = 1, t2 = 1. Затем t1 умножить на 1.452123 и 10 раз на 2.456, а t2 10 раз на 2.456 и по-

том на 1.452123, то оба числа будут представлены как 11595.4, но при сравнении не будут равны. Аналогично для double со значением 1.61315e+78.

Отдельно можно выделить тип char, который является целым числом, но может обрабатываться и как число и как символ. Подробнее данный тип будет рассмотрен позднее.

Тип bool может принимать всего 2 значения, потому с его помощью удобно вычислять логические выражения, которые будут рассмотрены позднее. Также для удобства для него заведено 2 константы: true (да; истина) = 0, false (нет; ложь). Для лучшей читаемости кода для данного типа лучше пользоваться именно ими, а не непосредственно 0 и 1.

Отдельно стоит указать тип «string». Переменные данного типа могут хранить строки. Данный тип сильно отличается от ранее приведенных. Также он находится в пространстве имен «std», т.е. полное имя «std::string». Подробнее данный тип будет рассмотрен намного позднее. Однако, процесс ввода значения в переменные данного типа и вывод их в консоль не сильно отличается от других типов. Поэтому для подобных действий тип можно использовать уже в этой работе.

Зная вышеописанное можно начать формализовывать алгоритм, приведенный на рис. 3б:

```
int a, b, c, V;
cout << "Vvedite a: ";
```

Таким образом были объявлены необходимые сущности и запрошено первое значение у пользователя. Теперь необходимо получить значение, введенное с клавиатуры. Ранее уже было сказано, что для этого используется поток (очередь) cin.

Аналогично оператору вывода «<<» для «cout», для «cin» есть оператор ввода «>>» (два знака больше), слева от которого должен стоять поток («cin»), а справа имя переменной, в которую будет записано вводимое значение. Например, «cin

>> a;». Аналогично тому, как оператор «<<» возвращает «cout», оператор «>>» возвращает «cin». Соответственно можно описывать ввод нескольких значений за 1 команду: «cin >> a >> b >> c;».

Также стоит описать как происходит ввод из консоли в соответствии с данным оператором. При первом вызове команды вида «cin >> имя_переменной» выполнение программы переходит в режим ожидания, а консоль в режим ввода (пользователь может печатать в ней текст). Выход из этого режима происходит после нажатия на клавиатуре «enter». Всё что ввел пользователь (строка) помещается в поток ввода («cin»). Далее срабатывает оператор «>>» (назовем этот набор действий cin1):

1. Если в начале строки есть пробелы, они все удаляются;

2. Все символы с начала строки до пробела удаляются из строки (оставшиеся символы в виде строки также остаются в потоке ввода) и сохраняются в отдельную строку (назовем ее temp);

3. Набор символов из строки temp приводятся к типу переменной, которая стоит справа от оператора «>>» (пусть это будет «a»);

4. Полученное значение сохраняется в переменную «a».

При повторном вызове команды вида «cin >> имя_переменной»:

1. Действие 1 из набора действий cin1

2. В зависимости от условий:

2.1. Если строка в потоке ввода не пуста, то действия 2-4 из набора действий cin1 (не происходит переход в режим ожидания и ввода).

2.2. Если строка в потоке ввода пуста, то полное повторение действий как при первом вызове команды вида «cin …» (происходит переход в режим ожидания и ввода).

Для понимания работы команды с цепочкой ввода вида «cin >> a >> b;», можно привести ее к отдельным командам «(cin >> a) >>b» -> «cin >> a; cin >>b;».

Исходная команда работает полностью аналогично получившимся отдельным командам.

Доделаем реализацию варианта с рис. 3б

```
int a, b, c, V;
cout << "Vvedite a: ";
cin >> a;
cout << "Vvedite b: ";
cin >> b;
cout << "Vvedite c: ";
cin >> c;
//данное выражение идентично его записи
//в неформальном виде, потому на данном этапе
//не разбирается отдельно
V = a * b * c;
cout << "Ob'em figuri: " << V;
```

Теперь рассмотрим 2 варианта совместного поведения пользователя и программы (Таблица 2, Таблица 3), которые продемонстрируют работу потоков ввода/вывода.

Таблица 2

Вариант выполнения программы расчета объема фигуры 1

Команда	Пояснение	Состояние консоли
cout << “Vvedite a: “		Vvedite a:
cin >> a	Поток ввода пуст. Переход в режим ожидания ввода.	Vvedite a: 2 |

	Пользователь вводит 1 значение и нажимает «enter» Символ «|» в состоянии консоли указывает текущее положение курсора в консоли и не видим при реальном выполнении	
cout << “Vvedite b: ”		Vvedite a: 2 Vvedite b: |
cin >> b	Поток ввода пуст, полностью повторяется операция ввода, как и в первый раз. Пользователь также вводит одно значение	Vvedite a: 2 Vvedite b: 2 |
cout << “Vvedite c: ”		Vvedite a: 2 Vvedite b: 2 Vvedite c: |
cin >> c	Поток ввода снова пуст. Операции, как и раньше. Но, пользователь вводит несколько значений и жмет «enter». В переменную «c» помещается только значение, полученное из первых символов до пробела (2). Также пропускаются ведущие пробелы. Остаток строки остается в потоке ввода (« 4 5»)	Vvedite a: 2 Vvedite b: 2 Vvedite c: 2 4 5 |
V = a * b * c;	V = a * b * c = 2 * 2 * 2 = 8	Vvedite a: 2

cout << “Ob’em figu- ri: “ << V;		Vvedite b: 2 Vvedite c: 2 4 5 Ob’em figuri: 8|

Таблица 3

Вариант выполнения программы расчета объема фигуры 2

Команда	Пояснение	Состояние консоли
cout << “Vvedite a: “		Vvedite a:
cin >> a	Поток ввода пуст. Переход в режим ожидания ввода. Пользователь вводит много значений и нажимает «enter». Из потока ввода в переменную «a» записывается только «2». При этом в потоке ввода остается « 3 5 9»	Vvedite a: 2 3 5 9 |
cout << “Vvedite b: ”		Vvedite a: 2 3 5 9 Vvedite b: |
cin >> b	Поток ввода не пуст. Переход в режим ожидания не происходит. Берется следующее значение из потока ввода («3»). В потоке ввода остается « 5 9»	Vvedite a: 2 3 5 9 Vvedite b: |
cout << “Vvedite c: ”		Vvedite a: 2 3 5 9 Vvedite b: Vvedite c: |
cin >> c	Поток ввода не пуст. Переход в ре-	Vvedite a: 2 3 5 9

	жим ожидания не происходит. Берется следующее значение из потока ввода («5»). В потоке ввода остается «9»	Vvedite b: Vvedite c: |
V = a * b * c; cout << “Ob’em figuri: “ << V;	V = a * b * c = 2 * 3 * 5 = 30	Vvedite a: 2 3 5 9 Vvedite b: Vvedite c: Ob’em figuri: 30|

Как можно видеть, наш вариант реализации подходит для первого варианта (Таблица 2). Т.е. мы рассчитываем, что пользователь будет нажимать «enter» после каждого введенного значения. При этом второй вариант (Таблица 3) позволяет понять, как можно реализовать алгоритм, указанный на рис. 3в. Там мы будем рассчитывать, что пользователь введет 3 значения через пробел и только потом нажмет «enter». При этом обе реализации будут верно считать результат в любом случае. Влияния на порядок выполнения команд не будет. Влияние будет оказано только на внешний вид текста в консоли. Это связано с тем, что ввод осуществляется не непосредственно из консоли, а из дополнительной сущности – потока ввода.

Приведем полную реализацию алгоритма с рис. 3в:

```
#include <iostream>
using namespace std;
int main()
{
   int a, b, c, V;
   cout << "Vvedite a, b, c: ";
   cin >> a >> b >> c;
```

```
    V = a * b * c;
    cout << "Ob'em figuri: " << V;
}
```

Текст в консоли (при условии приведенного варианта ввода) должен выглядеть следующим образом:

```
Vvedite a, b, c: 2 2 2
Ob'em figuri: 8
```

Задания

1. Написать программу расчета длины отрезка по 2 точкам (x, y). Необходимо 2 раза опросить пользователя (сначала запросить координаты первой точки, потом второй). Выражение для расчета длины отрезка на данном этапе приводится в задании, т.к. выражения будут подробнее рассматриваться в следующих работах: sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1)). Где x1, y1, x2, y2 – координаты 1 и 2 точки соответственно. sqrt – функция взятия квадратного корня числа из библиотеки «math.h»

2. Написать программу анкетирования пользователя. Необходимо запросить следующие данные и именно в таком порядке: фамилия; имя; отчество; одним запросом: день, месяц, год рождения.

Далее необходимо вывести информацию в соответствии с приведенным шаблоном:

```
FIO: фамилия имя отчество
Data: год_рождения.месяц.день
```

ЛАБОРАТОРНАЯ РАБОТА 3. ВЫРАЖЕНИЯ. ВЫПОЛНЕНИЕ В ЗАВИСИМОСТИ ОТ УСЛОВИЙ.

В рамках данной работы будет решена следующая задача:

1. Необходимо определить попадает ли точка J в квадрат EFGH и при этом не попадает в квадрат ABCD (рис. 4). Если попадает, то вывести на экран «DA», если нет, то вывести «NET»;

2. По условиям всегда EA = FB = GC = HD; AI=BI=CI=DI; EF=FG=GH=HE; AB=BC=CD=DA;

3. Стороны обоих квадратов параллельны осям координат;

4. С клавиатуры вводятся: x, y – координаты точки J; fx, fy – координаты точки I; sa – размер стороны внутреннего квадрата – AB; se – размер стороны внешнего квадрата – EF.

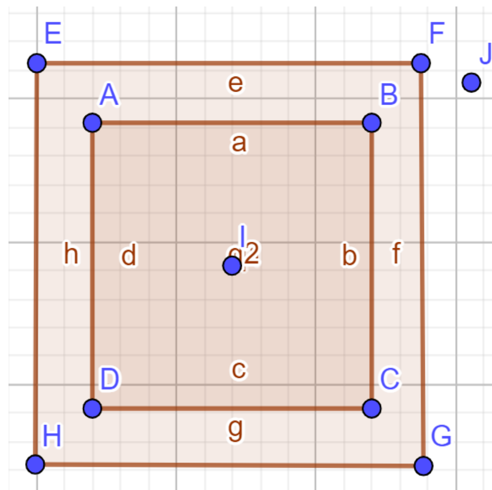

Рис. 4 Графическое представление задачи проверки попадания точки в фигуру

Как и в прошлой работе, зная условия задачи необходимо определить сущности, составить алгоритм и реализовать его.

Уже по условиям задачи необходимо ввести с клавиатуры: x, y, fx, fy, sa, se. Значит эти сущности точно будут необходимы. Для улучшения «читаемости» программы назовем их в коде понятнее: pointX, pointY (координаты точки, вхождение которой необходимо проверить); figureX, figureY (координаты центра сложной

фигуры, вхождение точки в которую проверяем); sideInner (сторона внутреннего квадрата); sideOuther (сторона внешнего квадрата). С такими названиями будет проще разбираться как работает уже готовая программа.

Остальные сущности не представляется возможным определить до составления концепции алгоритма.

Рассмотрим несколько концепций:

1. Проверять попадает ли точка в квадрат EFGH, затем проверять, что точка не попадает в квадрат ABCD;

2. Проверять попадает ли точка в одну и трапеций EADH, EABF, FGCB, HDCG;

3. Проверять попадает ли точка в один из прямоугольников ELQH, LABM, MFGP, DQPC (рис. 5).

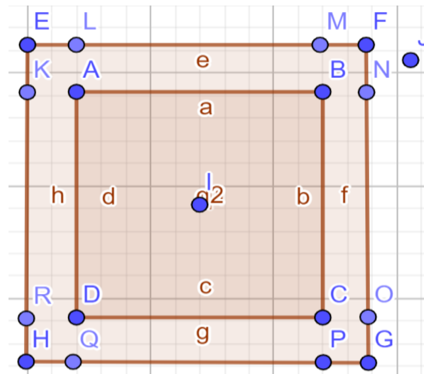

Рис. 5 Графическое отображение концепции попадания точки в прямоугольники

Для выбора концепции рассмотрим, как именно можно осуществить проверку в каждом из случаев.

1. Попадание в один из квадратов и выход за пределы другого

1.1. Квадрат EFGH образован сторонами EF, HG (эти стороны являются отрезками и лежат на прямых y = yEF, y = yHG), EH, FG (лежат на прямых x = xEH, x = xFG). Для того, чтобы точка (pointX, pointY) лежала внутри этого квадрата, необходимо, чтобы pointX был между xEH и xFG, а pointY между yHG и yEF. Где yEF – это y координата точки E и точки F; xEH – x координата точек E и H; остальное по аналогии. В этом случае x координата точки будет между отрезками

EH и FG, а y координата между HG и EF, что и означает нахождение точки внутри квадрата.

1.2. Квадрат ABCD образован сторонами AB, DC (на прямых y = yAB, y = yDC), AD, BC (x = xAD, x = xBC). Чтобы точка (pointX, pointY) выходила за пределы квадрата, необходимо чтобы pointY был больше yAB или меньше yDC или pointX был больше xBC или меньше xAD. Это будет означать, что хоть одна из координат точки не лежит между соответствующими отрезками и точка находится вне квадрата.

2. Попадание в трапеции рассчитывается сложнее 1 варианта, потому на данном этапе эта концепция может быть отброшена

3. Попадание в каждый из прямоугольников проверяется аналогично пункту 1.1. Соответственно необходимо будет описать 4 проверки вместо 2 в первой концепции. На данный момент также можно отбросить эту концепцию.

Таким образом, была выбрана концепция 1.

Рассмотрим, как можно получить yEF, yHG, xEH, xFG, yAB, yDC, xAD, xBC.

yEF это y координата точки I увеличенная на половину стороны квадрата EFGH. Т.е. yEF = figureY + (sideOuther / 2). yHG аналогично зависит от точки I в меньшую сторону: yHG = figureY – (sideOuther / 2).

xEH, xFG считаются по аналогии по x координате. xEH = figureX – (sideOuther / 2); xFG = figureX + (sideOuther / 2).

Аналогично: yAB = figureY + (sideInner / 2); yDC = figureY – (sideInner / 2); xAD = figureX – (sideInner / 2); xBC = figureX + (sideInner / 2).

Эти сущности также могут быть введены, но могут и вычисляться каждый раз в рамках проверок. На данном этапе, для упрощения условий, сущности будут введены и будут рассчитываться отдельно.

Для данного варианта решения можно ввести 2 дополнительные сущности: inOuther (находимся ли внутри внешнего квадрата – «да» или «нет»), outOfInner (находимся ли за пределами внутреннего квадрата – «да» или «нет»).

Также необходим дополнительный элемент для графического отображения алгоритма (блок-схемы) – ветвление. Т.е. необходим такой элемент, в котором можно задать некоторый вопрос и в зависимости от ответа на него продолжить выполнение программы по соответствующему пути. Проще всего выбирать вопросы, которые требуют ответов – «да» или «нет». Именно такой элемент и будем применять (рис. 6).

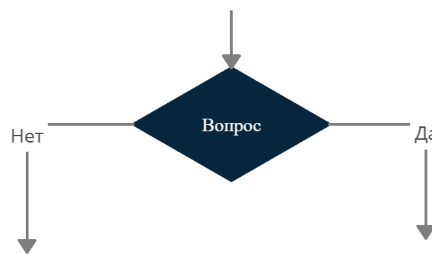

Рис. 6 Элемент блок-схемы – ветвление

Уточним итоговую концепцию в соответствии с новой информацией:

Ввести данные с клавиатуры

Рассчитать inOuther, outOfInner в соответствии с пунктом 1.1, 1.2. первой концепции.

Задать вопрос: Выполняется ли одновременно: inOuther – «да»; outOfInner – «да»?

ЕСЛИ ответ «да», ТО вывести на экран «DA»

ЕСЛИ ответ «нет», ТО вывести на экран «NET»

Составим подробный алгоритм (рис. 7).

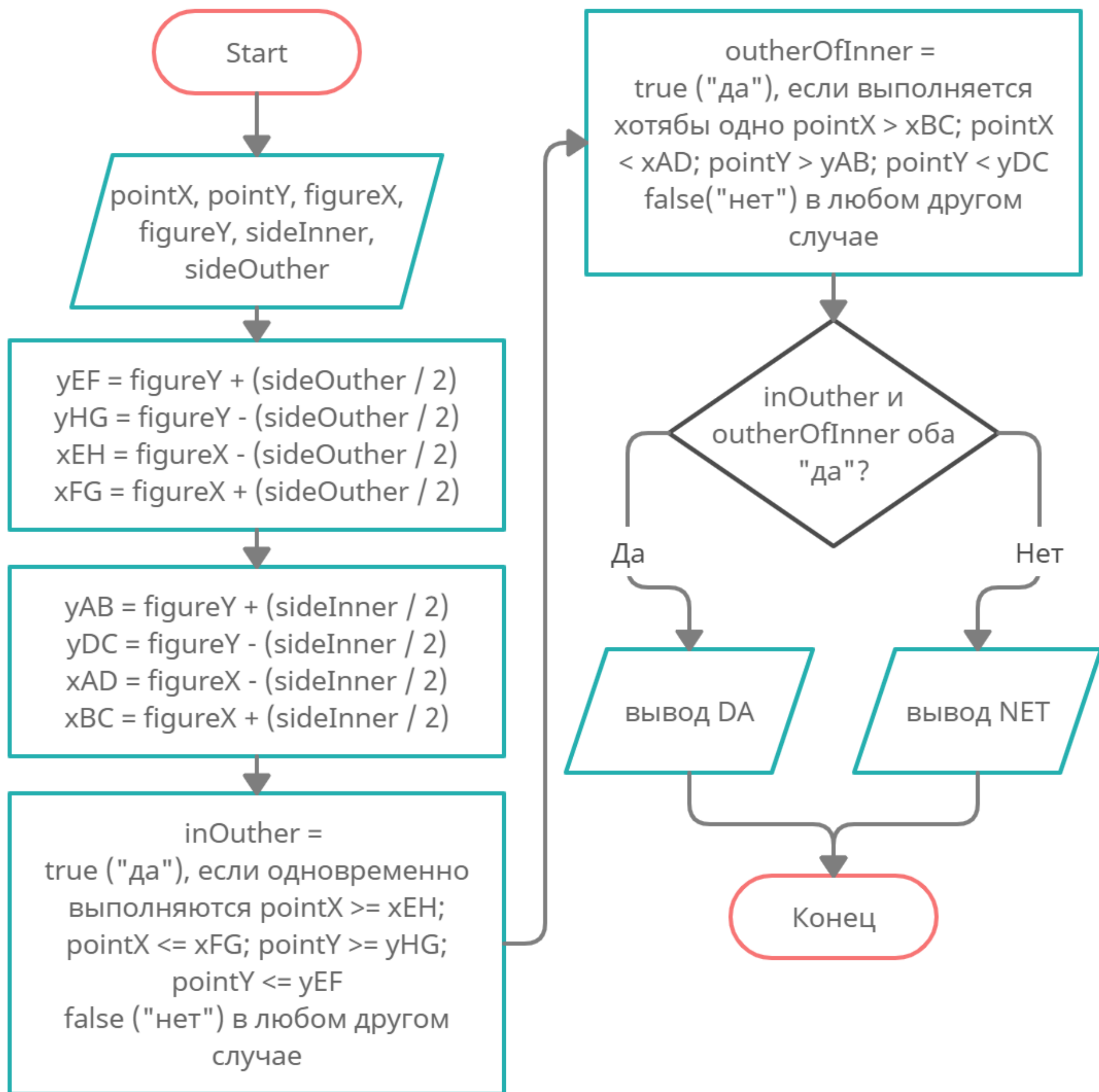

Рис. 7 Алгоритм проверки нахождения точки между двумя квадратами

Далее необходимо этот алгоритм реализовать. Для этого необходимо знать синтаксис записи выражений для вычислений и синтаксис для выполнения команд в зависимости от условий.

Большинство арифметических операций записываются так, как мы и привыкли их видеть. Например, 2 + 2 - 3, так и записывается.

Доступны следующие арифметические операции: a + b (сложение); a – b (вычитание); a * b (умножение); a / b (деление); a % b (остаток от деления).

В качестве операндов могут стоять: конкретные значения, переменные, а также другие выражения. Например: «a * (b + 2 – (k / 4));». Любое выражение возвращает результат (для 2 + 2 это 4).

Стоит помнить о приоритетности операций. Сначала выполняются более приоритетные операции. Операции с одинаковым приоритетом выполняются слева направо или справа налево, в зависимости от конкретных групп. Среди описанных операций выделяются следующие группы по приоритетам: {*, /, %} приоритетнее {+, -}. В рамках каждой из этих 2 групп приоритеты одинаковые, операции выполняются слева направо. Это значит, что в случае 1 + 2 + 3 сначала происходит вычисление 1 + 2, затем к полученному результату прибавляется 3. Т.е. (1 + 2) + 3. А в случае 1 + 2 * 3. Сначала выполняется 2 * 3, а затем к 1 прибавляется полученный результат.

Правило приоритетов распространяется абсолютно на все операции, в том числе и на логические, которые будут рассмотрены далее. Для указания собственного порядка действий можно использовать круглые скобки, как это делалось на уроках математики.

Также результат операции сильно зависит от типа операндов: «std::cout << (3 / 2);» выводит на экран «1», а «std::cout << (3.0 / 2);» выводит «1.5». Это связано с тем, что в первом случае оба операнда целые числа (аналогично работает, если в качестве операндов будут 2 переменные типа int) и происходит целочисленное деление (дробная часть результата отбрасывается). Во втором случае один из операндов число с плавающей точкой (для переменных это типы double, float), потому происходит деление с учетом дробной части.

Оператор % работает только если оба операнда целые числа и возвращает остаток от деления: «std::cout << (5 % 2);» выводит а экран «1». Подробнее: 2

(частное, результат целочисленного деления 5 / 2) * 2 (делитель) + 1 (остаток, результат 5 % 2) = 5 (делимое).

Следующим нам понадобится оператор «=» – присвоение. Данный оператор сохраняет значение правого операнда в левый операнд. Очевидно, что левым операндом может быть только переменная. Также этот оператор имеет самый низкий приоритет среди всех операторов, обсуждаемых в данной работе. И в своей группе операции выполняются справа налево. В качестве результата операции возвращается сохраняемое значение.

Например, разберем:

a = 2 + (c = b = 4 * 4 + 4 – 4);

Сначала выполняются операции в скобках. Среди них сначала выполняется наиболее приоритетная операция «*»

a = 2 + (c = b = 16 + 4 – 4);

Далее слева направо выполняются следующие по приоритету операции «+, -»:

a = 2 + (c = b = 20 – 4);

a = 2 + (c = b = 16)

Далее остаются группы операций, которые выполняются справа налево «=»:

сохраняем 16 в b, результат 16

a = 2 + (c = 16)

сохраняем 16 в c, результат 16

a = 2 + 16

Далее снова выполняются операции по приоритету. Это «+» и выполняется слева направо:

a = 18

Далее следующая по приоритету «=»:

сохраняем 18 в a, результат 18

Вычисления закончены.

Подробнее про приоритеты встроенных операторов в c++ и порядке их выполнения можно посмотреть в [6]. Там же можно увидеть, что здесь перечислены не все арифметические операторы. Часть из пропущенных операторов будут рассмотрены в следующих работах.

Далее необходимо рассмотреть операции сравнения: a > b, a < b, a <= b, a >= b (больше, меньше, меньше или равно, больше или равно); a == b (проверка на равенство через двойное равно); a != b (проверка на неравенство).

Каждая такая операция возвращает значение типа bool (true/1 или false/0). Т.е. для «1 > 0» результат будет true, а для «0 > 1» - false.

Следующая группа операторов – логические: a || b (двойная вертикальная черта; a ИЛИ b); a && b (двойной амперсанд; a И b); !a (восклицательный знак; унарный оператор отрицания; НЕ a). Логические операции читатель мог ранее встречать на курсах логики или дискретной математики. Краткая справка по ним будет приведена далее, а подробнее можно ознакомиться, например, в [7].

Каждый операнд такой операции должен быть типа bool. Т.к. для каждого операнда возможно всего 2 значения, то можно описать все возможные результаты данных операций при определенных операндах.

a && b: true && true => true; true && false => false; false && true => false; false && false => false (т.е. оператор И возвращает «да» только в том случае, если оба его операнда «да», а если хоть один «нет», то результат «нет».)

a || b: true || true => true; true || false => true; false || true => true; false || false => false (оператор ИЛИ напротив, возвращает «нет» только если оба операнда «нет», а в остальных случаях возвращает «да»)

!a: !true => false; !false => true (оператор НЕ возвращает значение, противоположное значению операнда).

Также опишем в одном месте приоритетность всех рассмотренных операций. Далее каждая строчка описывает группу операторов одной приоритетности и

порядок их рассмотрения. Каждая следующая строка содержит операторы с меньшим приоритетом:

! – справа налево

*, /, % - слева направо

+, - - слева направо

>, <, >=, <= - слева направо

==, != - слева направо

&& - слева направо

|| - слева направо

= - справа налево

Получив данную информацию, можно формализовать отдельные действия из алгоритма. yEF, yHG, xEH, xFG, yAB, yDC, xAD, xBC считаются полностью аналогично записи в блок схеме.

Рассмотрим inOuther. На все 4 вопроса «pointX >= xEH»? «pointX <= xFG»? «pointY >= yHG»? «pointY <= yEF»? ответ должен быть «да» и только в этом случае мы можем сказать, что точка находится во внешнем квадрате и присвоить значение «да» inOuther. Т.е. все эти выражения должны быть соединены через «И»:

inOuther = ((((pointX >= xEH) && (pointX <= xFG)) && (pointY >= yHG)) && (pointY <= yEF));

В записи выше приоритеты операций полностью заданы скобками. Однако, даже если скобки убрать, то результат не изменится.

inOuther = pointX >= xEH && pointX <= xFG && pointY >= yHG && pointY <= yEF;

Рассмотрим для примера pointX = 0, pointY = 0, xEH = -1, xFG = 1, yHG = -1, eEF = 1

Самая высокая по приоритету группа операций здесь это >=, <=, потому они будут выполнены первыми слева направо

inOuther = … && … && … && … => inOuther = true && … && … && … => inOuther = true && true && … && … => inOuther = true && true && true && … => inOuther = true && true && true && true

Далее слева направо будет выполнены &&.

inOuther = (((true && true) && true) && true) => inOuther = true

Т.к. в каждой из операций операндами будут true, то конечный результат тоже будет true.

Далее пройдет присвоение значения переменной.

Рассмотрим другой пример: pointX = 0, pointY = 2, xEH = -1, xFG = 1, yHG = -1, eEF = 1

inOuther = 0 >= -1 && 0 <= 1 && 2 >= -1 && 2 <= 1 => inOuther = true && true && true && false => inOuther = true && true && false => inOuther = true && false => inOuther = false

Далее рассмотрим outOfInner. Она «да» если любое из утверждений «да»: «pointX > xBC», «pointX < xAD», «pointY > yAB», «pointY < yDC». Такую зависимость можно задать, используя оператор «ИЛИ»:

outOfInner = pointX > xBC || pointX < xAD || pointY > yAB || pointY < yDC

Здесь также не обязательны скобки.

Следующее логическое выражение используется для определения дальнейшего варианта выполнения программы: если inOuther, outOfInner оба «да». Значит также можно использовать оператор «И»: «inOuther && outOfInner».

При этом промежуточные сущности можно было не вводить и сразу определить выражение:

pointX >= xEH && pointX <= xFG && pointY >= yHG && pointY <= yEF && (pointX > xBC || pointX < xAD || pointY > yAB || pointY < yDC).

Здесь скобки обязательны, т.к. && имеет больший приоритет и без скобок результат будет совсем другой, а именно:

(pointX >= xEH && pointX <= xFG && pointY >= yHG && pointY <= yEF && pointX > xBC) || pointX < xAD || pointY > yAB || pointY < yDC

Также можно было не вводить и xEH, xFG и т.д., а сразу использовать выражения для их вычислений. Здесь нужно соблюдать баланс между минимизацией количества переменных и сложностью выражений для поддержания «читаемости» кода и уменьшения необходимых ресурсов компьютера.

Также стоит отметить одну особенность вычисления логических выражений. Рассмотрим выражение (a) && (b), где a, b – некоторые сложные выражения. В данном случае должен сначала быть вычислен результат a, затем b и только потом a && b. Но, в связи с тем, что истинности данного выражения оба операнда должны быть истинными, если результат a будет ложен, то b вычисляться не будет и результат всего выражения сразу будет false. Аналогично для a || b. Если a истинно, то b вычисляться не будет и общий результат сразу будет true.

Т.к. выражения могут быть крайне сложными, в том числе использующие функции доступа к различным данным или запрашивающие пользовательский ввод, то знание этих особенностей позволяет спланировать выражения так, чтобы минимизировать число дорогих действий и количество ошибок.

Остается рассмотреть, как обеспечить ветвление процесса выполнения программы. Это реализуется с использованием конструкции «if … else …»:

```
if(условие)
    команда или составная команда 1
else
    команда или составная команда 2
```

Где в качестве «условия» может выступать любое выражение, которое возвращает значение типа bool. Если условие истинно (результат выражения «true»), то выполняется команда1, а если ложно (false), то команда 2.

Часть else может отсутствовать:

```
if(условие)
    команда или составная команда 1
```

В этом случае если условие истинно, то выполняется команда 1, а если ложно, то вся конструкция игнорируется и продолжается выполнение команд, которые идут после данной конструкции.

Любая из команд может быть составной. Это значит, что если по условию нужно выполнить несколько действий, то достаточно взять их в фигурные скобки:

```
if (условие) {
   команда1; … команда n;
} else {
   команда1; … команда m;
}
```

Где каждая команда может быть аналогично простой или составной.

Сама конструкция «if … else … » также считается одной составной командой. Это означает, что внутри одной конструкции ветвления может быть использована другая конструкция ветвления. И вложенность не ограничена. Оборачивать саму конструкцию в фигурные скобки при этом не обязательно.

Также стоит определить типы сущностей.

yEF, yHG и т.д. должны быть числом с плавающей точкой, т.к. при их вычислениях используется деление на 2 и дробная часть отбрасываться не должна.

Предполагается, что координаты, а значит и длины сторон задаются целыми числами. Однако, sideInner и sideOuther будут задаваться также числом с плаваю-

щей точкой. Это связано с тем, что они используются в формулах вида «sideInner / 2» и если sideInner будет int, то оператор «/» будет работать как целочисленное деление, что также отбросит дробную часть.

pointX, pointY, figureX, figureY могут быть объявлены как целым числом, так и числом с плавающей точкой.

inOuther, outOfInner очевидно bool, т.к. принимают значения «да» или «нет». Они также могут быть объявлены и как int и использовать только значения 0 и 1. Однако, это, как минимум, будет не оптимально с точки зрения использования оперативной памяти (int занимает больше, чем bool).

Используя все вышеописанное, можно оформить полную версию кода программы:

```
#include <iostream>
using namespace std;
int main()
{
   double yEF, yHG, xEH, xFG, yAB, yDC, xAD, xBC;
   double sideInner, sideOuther;
   int pointX, pointY, figureX, figureY;
   bool inOuther, outOfInner;

   //пояснения через cout не приводились в блок схеме для облегчения схемы
   cout << "Vvedite koordinati centra figuri cherez probel i nagmite enter: ";
   cin >> figureX >> figureY;
   cout << "Vvedite storonu vnutrenngo i vheshenego kvardratov cherez probel i 
nagmite enter: ";
   cin >> sideInner >> sideOuther;
```

```
    cout << "Vvedite koordinati proveraemoi tochki cherez probel i nagmite enter:
";
    cin >> pointX >> pointY;

    yEF = figureY + (sideOuther / 2);
    yHG = figureY - (sideOuther / 2);
    xEH = figureX - (sideOuther / 2);
    xFG = figureX + (sideOuther / 2);

    yAB = figureY + (sideInner / 2);
    yDC = figureY - (sideInner / 2);
    xAD = figureX - (sideInner / 2);
    xBC = figureX + (sideInner / 2);

    inOuther = pointX >= xEH && pointX <= xFG && pointY >= yHG &&
pointY <= yEF;
    outOfInner = pointX > xBC || pointX < xAD || pointY > yAB || pointY < yDC;

    if(inOuther && outOfInner) {
      cout << "DA" << endl;
    } else {
      cout << "NET" << endl;
    }

    return 0;
  }
```

Пример работы:

Vvedite koordinati centra figuri cherez probel i nagmite enter: 0 0

Vvedite storonu vnutrenngo i vheshenego kvardratov cherez probel i nagmite enter: 4 6

Vvedite koordinati proveraemoi tochki cherez probel i nagmite enter: 0 0

NET

Для закрепления материала далее будут приведены фрагменты программы, которые могут заменить текущую конструкцию «if … else». При этом результат меняться не будет.

```
//скобки не обязательны, т.к. по 1 действию
if(inOuther && outOfInner)
   cout << "DA" << endl;
else
   cout << "NET" << endl;
//end

//здесь скобки обязательны в части if, т.к. 2 действия
//и не обязательны в части else, т.к. 1
if(inOuther && outOfInner) {
   cout << "DA";
   cout << endl;
} else
   cout << "NET" << endl;
//end

//else можно и не писать
if(inOuther && outOfInner) {
```

```
    cout << "DA" << endl;
}
//но, чтобы удовлетворять условиям задачи
//все же нужно выводить NET
//для этого определим другой if, который срабатывает
//если истинно отрицание от основного условия,
//т.е. если оно не выполняется
//скобки обязательны, т.к. отрицание имеет больший приоритет,
//чем &&
if(!(inOuther && outOfInner)) {
    cout << "NET" << endl;
}
//end
```

Также стоит отметить, что вместо понятия outOfInner можно было использовать понятие inInner, которое бы указывало попадает ли точка во внутренний квадрат. Тогда, при объявлении одноименной переменной, условие «inOuther && outOfInner» превратилось бы в «inOuther && !inInner». Легко понять, что это одно и тоже, если раскрыть !inInner и увидеть, что он идентичен outOfInner.

Также стоит еще раз обратить внимание читателя на тот факт, что операция сравнения – это «==», а «=» - это операция присвоения. Причем операция присвоения также возвращает значение. И, если она будет использована в рамках условия if, то ошибки не будет. Если результат будет отличен от 0, то он будет приведен к true, а если 0, то к false. Это достаточно частая ошибка, которую не всегда просто сразу увидеть.

Например:

```
int a, b; a = 4; b = 5;
```

```
//присваиваем a значение b (5)
//теперь в a «5», в b «5»
//результат операции «5» приводится к true
//выполняются действия внутри if
//выводится DA
if(a = b) cout << “DA”;

int a, b; a = 0; b = 0;
//присваиваем a значение b (0)
//теперь в a «0», в b «0»
//результат операции «0» приводится к false
//не выполняются действия внутри if
//НЕ выводится DA
if(a = b) cout << “DA”;
```

Задания

1. Определить в какую четверть попадает точка. Координаты точки вводятся с клавиатуры. Если точка в начале координат, то необходимо вывести на экран «0». Если точка находится на оси координат между четвертями, то нужно вывести номера обеих четвертей через пробел. Если точка находится в одной из четвертей, то необходимо вывести номер четверти. Номера четвертей приведены на рис. 8.

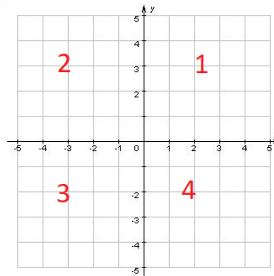

Рис. 8 Номера четвертей для задания определения четверти, в которую попадает точка

2. Определить, существует ли точка, которая лежит во всех трёх заданных окружностях одновременно. Вводятся координаты точки, координаты центров трёх окружностей, радиусы трёх окружностей. Примеры приведены на рис. 9.

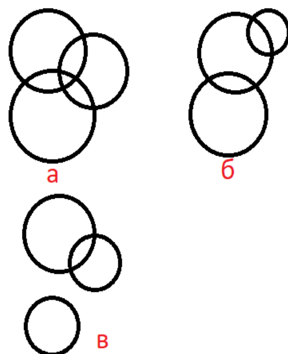

Рис. 9 Примеры для задачи определения существования точки, которая лежит во всех трёх заданных окружностях

ЛАБОРАТОРНАЯ РАБОТА 4. МНОГОКРАТНОЕ ВЫПОЛНЕНИЕ.

В рамках данной работы будет решена следующая задача:

1. Необходимо вычислить сумму всех четных чисел, которые больше или равны 1 и меньше или равны n.

2. С клавиатуры вводится только n.

3. На экран выводится строка вида «Summa elementov: 12»

Начнем планирование программы по стандартному плану. Выделим сущности, которые точно необходимы: n – максимальное значение проверяемого элемента; s – сумма четных элементов.

Далее разберем условия задачи для составления первоначальной концепции.

Необходимо вычислить сумму элементов s. Сумму всегда можно представить как $s = a_1 + a_2 + a_3 + \ldots + a_n$.

1. Каждый a_i в соответствии с задачей должен быть четным. Т.е. должен на цело делиться на 2. Т.е. остаток от деления его на 2 должен быть 0 (a_i % 2 == 0)

2. Каждый a_i должен удовлетворять: $1 <= a_i <= n$.

3. Все числа от 1 до n, удовлетворяющие условиям должны присутствовать в сумме.

Рассмотрим, как бы мы решали данную задачу самостоятельно без написания программы и как можно это реализовать. Первым этапом является формирование списка всех чисел от 1 до n.

Пусть n будет 5.

Рассмотрим все числа от 1 до n:

1, 2, 3, 4, 5

Рассмотрим, как бы мы вывели их на экран с использованием команд языка «c++»:

```
cout << 1 << ", " << 2 << ", " << 3 << ", " << 4 << ", "<< 5;
```

Однако, если в данном случае мы захотим рассмотреть числа от 2 до 7, то придется полностью изменять программу, пусть и состоящую из одной команды.

Попробуем разобраться как именно мы составили данный список. Проще всего сосчитать от 1 до 5. Т.е. начинаем перечисление с 1, далее рассматриваем следующее число, потом следующее от следующего и т.д.

Значит для формирования следующего числа нам нужно всегда знать предыдущее, и оно должно где-то храниться. Пусть для этого будет использоваться сущность с именем «i».

В данном случае, следующее число – это предыдущее, увеличенное на 1. Т.е. для текущего числа, записанного в «i», следующим будет «i + 1».

Т.е. для формирования списка можно записать в «i» начальное значение, а далее каждый раз прибавлять к этому числу 1 и полученный результат снова сохранять в i.

Попробуем снова вывести на экран эти числа:

```
i = 1;
cout << i; i = i + 1;
cout << i; i = i + 1;
cout << i; i = i + 1;
cout << i; i = i + 1;
cout << i;
```

Теперь для того, чтобы начинать не с 1, необходимо поменять только одно число в команде присвоения начального значения переменной «i».

Далее выполняются команды «cout << i; i = i + 1;» столько раз, сколько мы хотим вывести чисел. Последняя строка с командами, на данный момент не содержит «i = i + 1;». Однако, если эту команду включить и туда, то результат не изменится. Т.е. теперь, чтобы вывести не до 5, а до 7, необходимо будет добавить еще 2 аналогичные строки.

Однако, если n заранее не известно (вводится с клавиатуры), то снова возникает проблема с количеством таких строк.

Рассмотрим как мы сами решаем, когда остановится в перечислении чисел от 1 до 5. Мы начинаем перечисление с 1, каждый раз на основании предыдущего числа формируем следующее. Но, перед тем как называть новое число, мы проверяем не больше ли оно заданной границы. Т.е. перед тем как называть следующее число, мы проверяем не больше ли оно 5 («i <= 5»).

Запишем это в виде программы. Для этого объединим каждую «cout << i; i = i + 1;» в составную команду «{ cout << i; i = i + 1; }» и добавим конструкции для условного выполнения.

```
i = 1;
if(i <= n) {
  cout << i; i = i + 1;
  if(i <= n) {
    cout << i; i = i + 1;
    if(i <= n) {
      cout << i; i = i + 1;
      if(i <= n) {
        cout << i; i = i + 1;
        if(i <= n) {
          cout << i; i = i + 1;
        }
      }
    }
  }
}
```

Теперь, если n будет вводиться с клавиатуры и будет меньше или равен количеству сформированных конструкций, то программа будет работать верно. Но,

если n окажется больше, чем сформировано конструкций, то снова будет неверный результат. Кроме того, такая программа громоздкая и нечитаемая.

Значит нужна некоторая новая конструкция для повторного выполнения команд. Есть команда: «{cout << i; i = i + 1;}». Перед ее выполнением необходимо проверить, что ответ на вопрос «i <= 5» - «да». Если это так, то необходимо выполнить рассматриваемую команду и снова перейти на проверку, а в случае ее истинности, снова выполнить команду. И так до тех пор, пока условие не окажется ложным. Как только условие окажется ложным, перейти на дальнейшие действия программы.

Рассмотрим это в виде блок схемы (рис. 10 а).

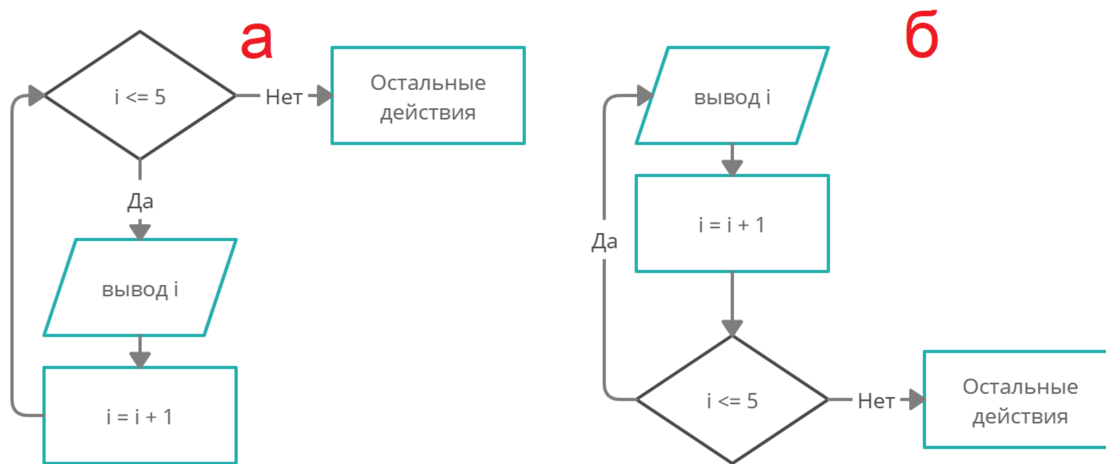

Рис. 10 Блок схема повторяемых действий

Для реализации такого фрагмента алгоритма (цикл с предусловием), в языке c++ есть специальная конструкция: «while».

```
while(i <= 5) {
  cout << i <<" ";
  i = i + 1;
}
//далее остальные действия
```

Т.е. синтаксис «while(условие) команда или составная команда». Пока условие истинно, выполняем команду.

Стоит отметить, что условие проверяется каждый раз перед выполнением, в том числе и в первый раз. Если оно изначально ложно, то команда не выполнится ни разу.

Также если команда не будет менять переменные, значения которых влияют на условие, то условие всегда будет оставаться неизменным. В этом случае, если выполнение хоть раз зайдет в тело цикла (условие истинно; выполнить команду), то выполнение будет продолжаться бесконечно (условие всегда будет оставаться истинным).

В нашем случае в теле цикла увеличивается переменная «i» и, когда она достигнет 6 (т.к. для 5 «5 <= 5» все еще будет истинно), повторное выполнение будет закончено. При этом, если изначально переменная будет иметь значение больше 5, то действия в теле цикла не будут выполнены ни разу.

Также можно сначала выводить число, а уже затем проверять условие и либо снова повторять действие или заканчивать повторения. Такое выполнение – это цикл с постусловием (рис. 10 б).

```
do {
  cout << i <<" ";
  i = i + 1;
} while(i <= 5);
//далее остальные действия
```

Т.е. синтаксис «do команда или составная команда while(условие);».

Заметим, что как правило после конструкций вида if, while и подобных, не ставилась «;». Это связано с тем, что после конструкций обычно шло действие, после которого и так ставилась «;» или составное действие, которое оборачива-

лось в «{ }» и в «;» не было необходимости. Здесь же составное действие «цикл с постусловием» оканчивается конструкцией проверки условия и потому требует «;» в конце.

С учетом вышеизложенной информации можно составить алгоритм для вывода всех чисел от 1 до n (рис. 11).

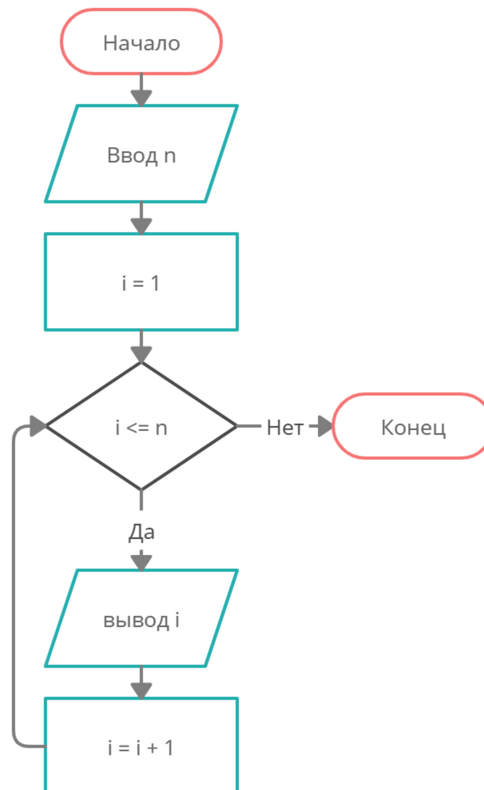

Рис. 11 Алгоритм вывода всех чисел от 1 до n.

Следующим шагом будет оставить среди формируемых чисел только четные. Т.е. для чисел: «1, 2, 3, 4, 5» должны остаться только «2, 4». Для начала тоже просто выведем их на экран.

Значит теперь необходимо выводить на экран только числа, подходящие под условие. Данное условие было сформировано ранее и, если текущее число хранится в переменной «i», будет выглядеть следующим образом – «i % 2 == 0». Дополним алгоритм (рис. 12). Затрагивается только блок вывода, он теперь выполняется только при соблюдении условий. Все остальное остается неизменным.

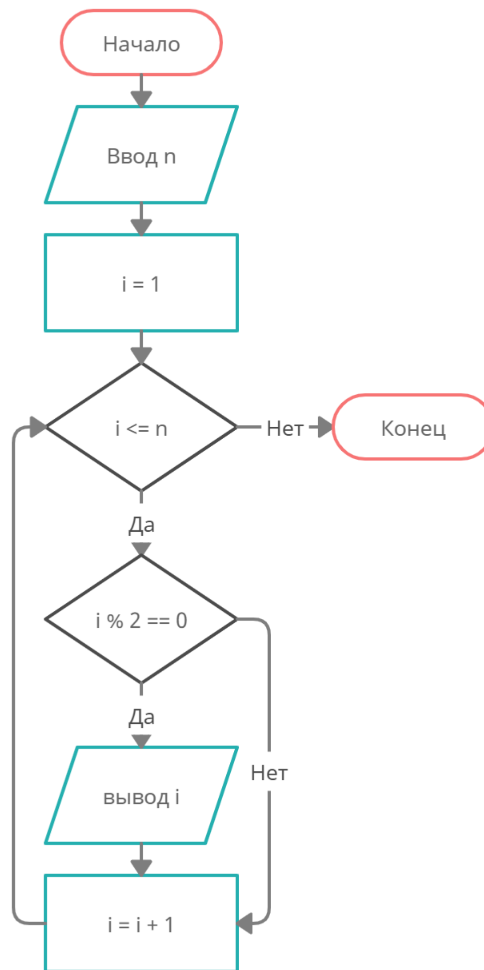

Рис. 12 Алгоритм вывода всех четных чисел от 1 до n

На данном этапе мы формируем список четных чисел от 1 до n. Теперь необходимо сосчитать сумму данных чисел. Ранее мы представляли сумму как $s = a_1 + a_2 + a_3 + \ldots + a_n$. Для хранения значения элемента a_i в рамках текущего алгоритма мы используем сущность «i».

Расчет суммы можно также разделить на повторяемые действия:

$s = a_1$

$s = s + a_2$

$s = s + a_3$

…

$s = s + a_i$

…

$s = s + a_n$

Т.е. по факту каждый раз нужно повторять действие «$s = s + a_i$», что в рамках алгоритма будет выглядеть как «s = s + i». Т.е. в первый раз при заходе в условие «i % 2 == 0», вместо вывода нужно сделать «s = i», а в последующие повторения «s = s + i».

Если делать таким образом, то появляется необходимость определять, какой заход является первым, какой последующим. Потому удобнее всего привести все действия к одному виду

$s = s + a_1$

…

$s = s + a_i$

…

$s = s + a_n$

Однако для того, чтобы добавлять что-то к s, там должно быть сохранено какое-то значение. В нашем случае там должно быть значение, при котором после первого действия s будет равно a_1. Т.е. «$s + a_1 = a_1$». Т.е. s изначально должно быть равно 0.

После выполнения всех итераций цикла (итерацией называется одно выполнение команды в рамках цикла) необходимо вывести получившееся s в консоль.

На основании вышесказанного дополним алгоритм (рис. 13).

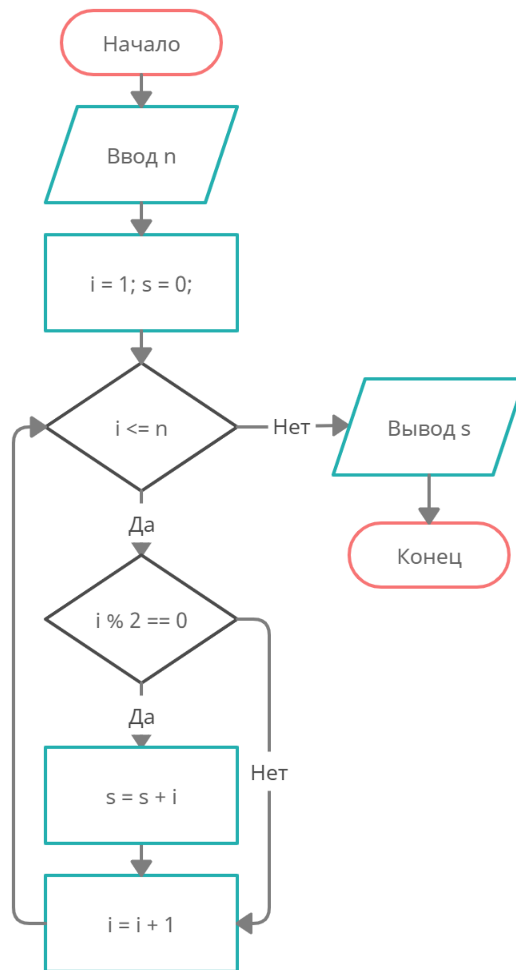

Рис. 13 Алгоритм расчета суммы всех четных чисел от 1 до n

Один из вариантов алгоритма сформирован. Рассмотрим еще раз последовательность чисел от 1 до 5: «1 2 3 4 5» и наш алгоритм. Можно увидеть, что мы будем рассматривать только числа от «i % 2 == 0», т.е. каждое второе число. Соответственно вместо проверки каждый раз, можно увеличивать «i» на 2 и определить с какого значения «i» начинать.

Для решения задачи мы проверяли все числа из этой последовательности на четность. Однако, можно было считать по-другому. Первое четное число в диапазоне от 1 до n – это 2. Вместо того, чтобы считать от 1 до 5 и проверять каждое следующее число на четность, можно попробовать считать от 2 до 5 и сразу рассматривать «следующее четное число». Следующее число – это число на 1 большее, чем предыдущее («i+1»). Если текущее число четное, то следующее число

точно нечетное, а число через одно точно четное. Соответственно «следующее четное число» - это «i + 2».

Рассмотрим все числа от 1 до 9 и все четные числа от 1 до 9

1, 2, 3, 4, 5, 6, 7, 8, 9

2, 4, 6, 8

На данном примере хорошо видно, что для перебора всех четных чисел достаточно начинать перебор с 2 и двигаться с шагом 2 (каждый раз сохранять в переменную текущее значение, увеличенное на 2).

Рассмотрим алгоритм, соответствующий данной концепции (рис. 14).

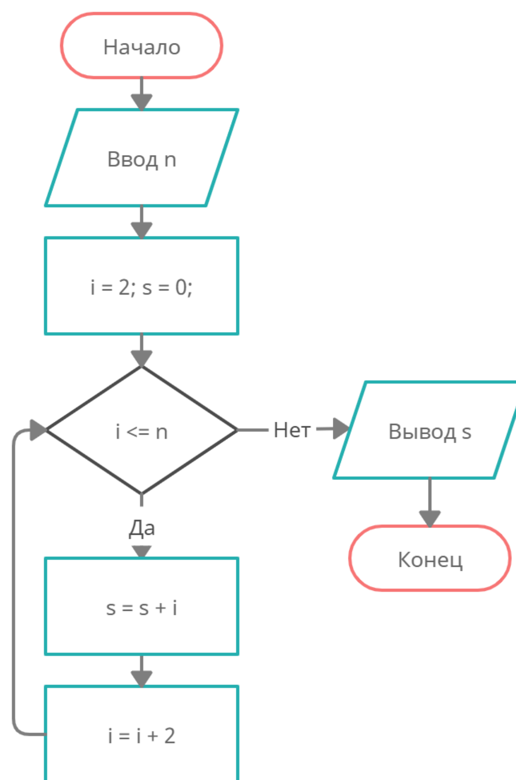

Рис. 14 Алгоритм расчета суммы всех четных чисел от 1 до n. Вариант 2

Второй вариант оказался проще и, в большинстве случаев, предпочтительнее. Однако, для закрепления материала, реализуем оба варианта. Типы всех переменных в данном случае int. Количество переменных небольшое, потому названия можно оставить в том виде, как они обсуждались в процессе формирования алгоритмов.

```
//вариант 1
#include <iostream>
using namespace std;
int main()
{
   int s //сумма всех четных чисел от 1 до n
      , i //текущее число
      , n; //максимальное число, до которого перебираем
   cin >> n;
   i = 1; s = 0;
   while(i <= n) {
      if(i % 2 == 0) {
         s = s + i;
      }
      i = i + 1;
   }
   cout << "Summa elementov: " << s;
   return 0;
}
//вариант 2
#include <iostream>
using namespace std;
int main()
{
   int s //сумма всех четных чисел от 1 до n
      , i //текущее число
      , n; //максимальное число, до которого перебираем
```

```
    cin >> n;
    i = 2; s = 0;
    while(i <= n) {
        s = s + i;
        i = i + 2;
    }
    cout << "Summa elementov: " << s;
    return 0;
}
```

Фрагмент алгоритма, когда необходимо присвоить переменной начальное значение, а затем выполнять цикл, где в конце каждой итерации значение переменной меняется, необходим достаточно часто. Поэтому для такой задачи была создана отдельная конструкция, позволяющая немного сократить объем. При этом ничего не меняется в рамках выполнения.

Это конструкция «for(начальные действия; условие; действия в конце цикла) команда или составная команда».

Она расшифровывается как:

```
{
  начальные действия;
  while(условие) {
      команда или составная команда;
      действия в конце цикла;
   }
}
```

Обрамляющие фигурные скобки важны. Переменные существуют до тех пор, пока не закончилось выполнение составной команды, в рамках которой они объявлены. Если мы объявляем переменную в начале функции main, то она будет доступна в рамках любого действия в функции main. Но, если бы объявили переменную в начале тела цикла, то эта переменная была бы доступна только в рамках этой итерации цикла, а затем удалялась бы.

Рассмотрим пример: необходимо вывести все числа от 1 до 10.

```
int i = 0;
while(i <= 10) { cout << i << " "; i = i + 1; }
cout << i; //для себя выведем последнее значение i (оно будет 11)
```

Запишем это в виде конструкции for:

```
int i;
for(i = 0; i <= 10; i = i + 1) {
  cout << i << " ";
}
cout << i;
```

Данный код будет работать полностью аналогично. При этом нас ничто не ограничивает в начальных действиях и объявление переменной также можно поместить туда (что чаще всего и делают).

```
for(int i = 0; i <= 10; i = i + 1) {
  cout << i << " "; //все работает хорошо
}
cout << i; //ошибка, переменная i не объявлена
```

Рассмотрим почему так произошло. Перепишем for обратно в while.

```
{
  int i = 0; //объявляем переменную
  while(i <= 10) {
    cout << i << " ";
    i = i + 1;
  }
  //составная команда, где объявлена i оканчивается, i уничтожается
}
cout << i; //ошибка, i не объявлено
```

Также рассмотрим несколько сокращенных записей арифметических операций:

i = i + 1; => i++; (сохраняет в i ее значение, увеличенное на 1)

i = i + a; => i += a; (сохраняет в i ее значение, увеличенное на a. Вместо a может быть как переменная, так и конкретное значение)

По аналогии существуют: «i--;», «i-=a;», «i*=a», «i/=a» и другие. Полный список не приводится, т.к. они не понадобятся в рамках этого пособия, он может быть найден в [6].

Данные операторы также могут быть использованы в рамках выражений и имеют приоритеты. Все это умышленно не приводится на данном этапе, как и такие вопросы как разница между «i++», «++i». Данный материал будет дан позднее. Обучаемым, которые сомневаются в полном понимании работы этих операций, предлагается на данном этапе использовать их только вне выражений.

Также рассмотрим еще один особенный оператор, который может понадобиться в конструкции «for» - “,”. Данный оператор имеет наименьший приоритет из всех операторов c++. Он разделяет выражения и выполняет их слева направо. В

качестве результата операции будет всегда результат правого выражения. Все остальные результаты отбрасываются. В нашем случае он полезен только в рамках «for» для перечисления нескольких начальных (или выполняемых в конце тела цикла) действий. Также стоит обратить внимание, что для условий этот оператор не нужен, все что стоит слева от него, в этом случае, не будет учитываться. Условие должно быть составлено только с использованием логических операторов.

Приведем отвлеченный пример работы оператора:

```
//будет вычислено левое выражение: 2 + 2, результат 4
    //будет вычислено правое выражение: 6 + 2, результат 8
    //результат 4 при этом забывается
    //8 сохраняется в а
    //
    //именно здесь круглые скобки крайне важны,
    //т.к. "=" имеет больший приоритет и без них
    //сначала должно осуществляться присвоение, а затем
    //вычисление правой части, что вызовет ошибку
    int a = (2 + 2, 6 + 2);
    //выводится 8
    cout << a;
```

Теперь преобразуем второй вариант реализации алгоритма с использованием «for».

```
//вариант 2
#include <iostream>
using namespace std;
int main()
{
```

```
    int s, n;
    cin >> n;
    for(int i = 2, s = 0; i <= n; i += 2) {
        s += i;
    }
    cout << "Summa elementov: " << s;
    return 0;
}
```

Как видно, все эти изменения «облегчают» код и делают его проще для восприятия. Эти конструкции очень похожи в разных языках программирования и, как правило, знакомы всем разработчикам.

Для окончательного закрепления материала произведем трассировку выполнения последней реализованной программы (пошаговое выполнение).

```
cin >> n; //вводим n = 5
for(int i = 2, s = 0; …; …) /*выполняются начальные действия.
В i сохраняется 2, в s 0
Результат всего выражения – 0, но он для нас не важен
*/
for(…; i <= n; …) //проверяется условие. 2 <= 5 – true, значит выполняем тело
s += i; //s = s + i = 0 + 2 = 2
for(…; …; i+=2) //после каждой итерации выполняем i+=2; i = i + 2 = 2 + 2 = 4
for(…; i <= n; …) //проверяется условие. 4 <= 5 – true, значит выполняем тело
s += i; //s = s + i = 2 + 4 = 6
```

for(…; …; i+=2) //после каждой итерации выполняем i+=2; i = i + 2 = 4 + 2 = 6

for(…; i <= n; …) //проверяется условие. 6 <= 5 – false, значит не заходим в тело цикла, повторное выполнение окончено

cout << "Summa elementov: " << s; //выводим на экран «Summa elementov: 6»

Также стоит заметить, что и «for» и «while» и «do … while» являются составными командами и могут использоваться внутри друг друга и в рамках других составных команд.

Задания

1. Вывести в консоль числа кратные 3 из арифметической прогрессии с разностью прогрессии d (т.е. $a_{i+1}=a_i+d$), большие или равные a и меньшие или равные b. a, b, d вводятся с клавиатуры

2. Для чисел от 1 до n посчитать сумму $s = 1^1 + 2^2 + 3^3 + 4^4 + \ldots + n^n$.

n вводится с клавиатуры. После реализации рассчитать s сначала для малых n (n = 3, 4). Далее рассчитать s для n = 1000. Ответить на вопрос почему получился такой результат?

СПИСОК ЛИТЕРАТУРЫ

1. cppreference.com [Электронный ресурс]. URL: https://en.cppreference.com/w/ (дата обращения: 01.12.2020).
2. Документация по Microsoft C/C++ | Microsoft Docs [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/cpp/?view=msvc-160 (дата обращения: 01.12.2020).
3. Bohm, Corrado; and Giuseppe Jacopini. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules (англ.) // Communications of the ACM : journal. — 1966. — May (vol. 9, no. 5). — P. 366—371
4. Системы счисления и их применение / С.Б. Гашков. – Москва: МЦНМО, 2004. – 52 с.
5. Представление вещественных чисел — Викиконспекты [Электронный ресурс]. URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%9F%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B2%D0%B5%D1%89%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D1%85_%D1%87%D0%B8%D1%81%D0%B5%D0%BB (дата обращения: 01.12.2020).
6. Встроенные операторы, приоритет и ассоциативность C++ [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/cpp/cpp/cpp-built-in-operators-precedence-and-associativity?view=msvc-160 (дата обращения: 01.12.2020).
7. Яблонский С.В. Введение в дискретную математику: Учеб. Пособие для вузов — 2е изд,, перераб. И доп. — М.: Наука. Гл. ред. Физ.-мат. Лит. — 384 с.