

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	6
1 ОСНОВНЫЕ АЛГОРИТМИЧЕСКИЕ СТРУКТУРЫ.....	7
Практическая часть	10
2 БАЗОВЫЕ ТИПЫ ЯЗЫКА C++.....	12
Практическая часть	16
3 ОПЕРАТОРЫ ВЕТВЛЕНИЯ ЯЗЫКА C++.....	17
Практическая часть	21
4 ОПЕРАТОРЫ ОРГАНИЗАЦИИ ЦИКЛОВ ЯЗЫКА C++	24
Практическая часть	28
5 УКАЗАТЕЛИ И ССЫЛКИ.....	30
Практическая часть	34
6 МАССИВЫ	36
Практическая часть	40
7 ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ ДАННЫХ. СТРУКТУРЫ.....	43
Практическая часть	44
8 ФУНКЦИИ	49
Практическая часть	55
9 ЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ.....	56
Практическая часть	64
10 ВВЕДЕНИЕ В ЯЗЫК ПРОГРАММИРОВАНИЯ PYTHON.....	66
Практическая часть	69
11 ОПЕРАТОР ВЕТВЛЕНИЯ В ЯЗЫКЕ PYTHON	70
Практическая часть	73
12 ОПЕРАТОРЫ ОРГАНИЗАЦИИ ЦИКЛОВ ЯЗЫКА PYTHON	74
Практическая часть	77
13 СТРОКИ	79
Практическая часть	82
14 МАССИВЫ. СПИСКИ.....	83

Практическая часть	87
15 ФАЙЛЫ	89
Практическая часть	90
16 ПРОЦЕДУРЫ. ФУНКЦИИ	92
Практическая часть	94
ЗАКЛЮЧЕНИЕ	96
ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ	97
Что такое map()?	97
Синтаксис	97
Пример использования.	97
Где применяется.	99
Плюсы и минусы.	100

ВВЕДЕНИЕ

Целью учебной практики является закрепление пройденного материала теоретического курса по дисциплинам ОПОП, получение навыков практического решения прикладных задач, получение первичных профессиональных умений и навыков, в том числе первичных умений и навыков научно-исследовательской деятельности. Для получения соответствующих навыков важно не только решение практических задач параллельно с получением и расширением теоретических знаний, но и умение решать практические задачи после освоения теоретических курсов. Такие задачи должны быть комплексными, т. е. для их решения нужно владеть знаниями нескольких дисциплин. Решение достаточного количества таких задач в ходе учебной практики позволит не только усовершенствовать навыки, полученные за период обучения, но и выявить слабые места, которым стоит уделить больше внимания в ходе дальнейшего обучения. Решение практических задач дает возможность понять необходимость вдумчивого и тщательного изучения теоретических основ профессии, что в свою очередь закладывает прочный фундамент для дальнейшего обучения.

1 ОСНОВНЫЕ АЛГОРИТМИЧЕСКИЕ СТРУКТУРЫ

Алгоритм – четкое предписание исполнителю выполнить определенную последовательность действий, направленных на достижение определённой цели.

От любой другой последовательности действий алгоритм отличают его свойства:

1. Дискретность – разбиение алгоритма на последовательность отдельных законченных действий, шагов. Каждый такой шаг должен быть закончен до выполнения следующего.

2. Точность – однозначность указаний.

Состояние объектов среды исполнителя однозначно определено на каждом шаге. На каждом шаге однозначно определён шаг, который нужно выполнить следующим. Таким образом, при применении алгоритма к одному набору входных данных на выходе каждый раз будет получен один и тот же результат.

3. Понятность – алгоритм должен быть изложен на языке, понятном для исполнителя; таким образом, каждый шаг алгоритма будет трактован однозначно, т. е. состоять из команд, входящих в систему команд исполнителя.

4. Конечность (результативность) – обязательное получение результата за конечное число шагов.

Работа алгоритма должна быть завершена за конечное число шагов в любом случае, даже если решение не найдено. Теоретические аспекты бесконечных алгоритмов в рамках учебной практики не рассматриваются.

5. Массовость – применение алгоритма к решению всего класса однотипных задач.

Способы представления алгоритма

Выделяются следующие формы записи алгоритмов:

– графическая запись (блок-схемы);

- на естественном языке (словесная запись, псевдокод);
- код на языке программирования;
- в виде математической формулы.

Алгоритм в виде блок-схемы представляет собой последовательность связанных между собой функциональных блоков, соответствующих шагам алгоритма. Блоки соединены между собой линиями, определяющими действие, которое должно быть выполнено следующим.

Представление алгоритма в виде блок-схемы строго формализовано. Инструкции к представлению алгоритма таким образом содержатся в ГОСТ 19.701-90. ЕСПД. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения.

Требования к форме и размерам блоков приведены в ГОСТ 19.003-80. Схемы алгоритмов и программ. Обозначения условные и графические.

При изображении блоков размер стороны a выбирается из ряда 15, 10, 20 мм, допускается увеличение значения параметра a на число, кратное 5. Сторона b соотносится со стороной a таким образом, что $b = 1,5a$.

Основные блоки представлены в прил. 3.

Любой алгоритм может быть представлен с использованием трех основных алгоритмических структур: следования, ветвления и цикла. Алгоритмы, содержащие несколько алгоритмических структур, называют комбинированными.

Следование – алгоритмическая структура, в которой все команды выполняются последовательно, одна за другой. Алгоритмы, в которых используется только алгоритмическая конструкция «следование», называются линейными.

Ветвление – алгоритмическая структура, в которой в зависимости от значения логического выражения будет выполнено либо одно, либо другое действие.

Существует две формы ветвления: полная и неполная. В полной форме ветвление содержит два действия (последовательности команд), одно из этих

действий будет выполнено при значении условия «истина», а второе – при значении условия «ложь». В неполной форме ветвление содержит только одно действие или последовательность команд, которые будут выполнены при значении условия «истина». В прил. 4 приведены структурные схемы полной и неполной форм ветвления.

Алгоритмы, в основе которых лежит алгоритмическая конструкция «ветвление», называют разветвляющимися.

Повторение (цикл) – алгоритмическая конструкция, с помощью которой определенная последовательность действий выполнится необходимое число раз. Алгоритмы, основой которых служит конструкция «повторение», называют циклическими, или циклом. Под телом цикла понимают действия, многократно повторяющиеся в процессе выполнения цикла.

Выделяют два типа циклов (по взаимному расположению тела цикла и условиям продолжения):

- 1) цикл с постусловием;
- 2) цикл с предусловием.

При использовании конструкции «цикл с предусловием» проверка условия происходит до выполнения действий тела цикла. Возможна ситуация, в которой тело цикла не выполнится ни разу.

При использовании конструкции «цикл с постусловием» тело цикла будет выполнено как минимум один раз, так как проверка условия происходит после выполнения тела цикла и в зависимости от результата этой проверки будет осуществлён выход из цикла или переход на следующую итерацию.

Для успешной организации алгоритмической конструкции «повторение» следует до входа в цикл задать начальные значения переменных, используемых в цикле. В теле цикла необходимо предусмотреть изменение переменных, анализируемых в условии продолжения цикла.

При решении конкретных задач алгоритм может содержать более одной конструкции «повторение». В зависимости от их взаимного расположения говорят о вложенных или последовательных циклах.

Если один цикл является частью тела другого цикла, то первый цикл называют вложенным, второй – внешним.

Перед решением задачи и написанием кода на языке программирования необходимо представить графическое решение в виде блок-схемы.

Практическая часть

Задание: даны длины ребер a , b , c прямоугольного параллелепипеда. Найдите его объем и площадь поверхности.

Листинг приложения:

```
void TaskIndivid13_block1() {
    int a, b, c, _v, _s;
    cout << "Введите длину ребра a: ";
    cin >> a;
    cout << "Введите длину ребра b: ";
    cin >> b;
    cout << "Введите длину ребра c: ";
    cin >> c;
    _s = 2 * (a * b + b * c + a * c);
    _v = a * b * c;
    cout << "Площадь поверхности = " << _s << endl << "Объем параллелепипеда = " <<
_v << endl;
}

int main(){
    setlocale(LC_ALL, "rus");
    Task13_block1();
}
```

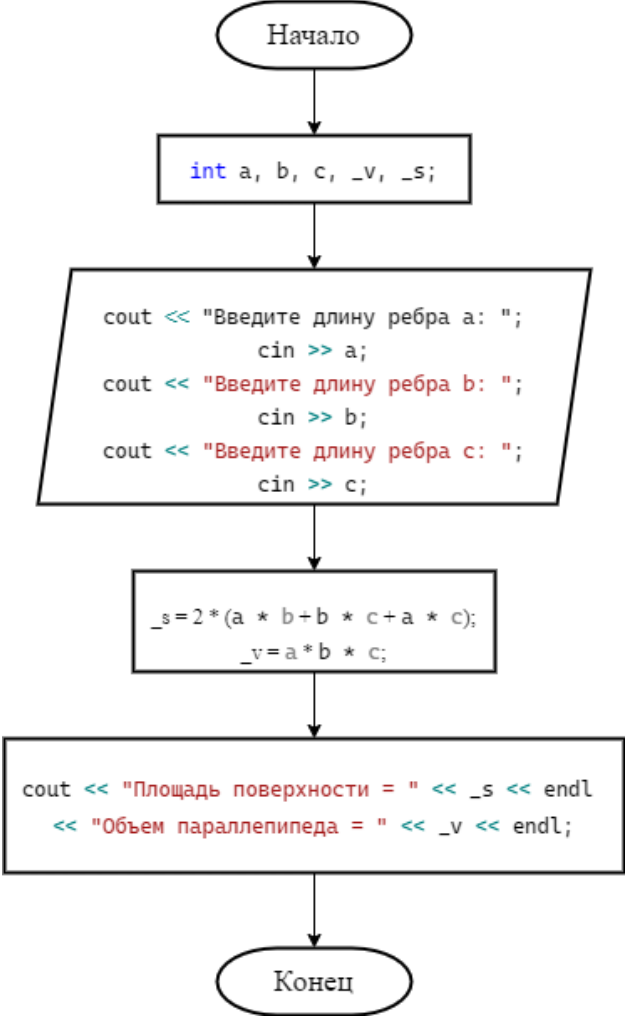
Результат выполнения кода:	Блок-схема
<pre> Введите длину ребра a: 4 Введите длину ребра b: 5 Введите длину ребра c: 7 Площадь поверхности = 166 Объем параллелепипеда = 140 </pre>	 <pre> graph TD Start([Начало]) --> Decl[<code>int a, b, c, _v, _s;</code>] Decl --> Input[/<code>cout << "Введите длину ребра a: "; cin >> a; cout << "Введите длину ребра b: "; cin >> b; cout << "Введите длину ребра c: "; cin >> c;</code>/] Input --> Calc[<code>_s=2*(a * b+b * c+a * c); _v=a*b * c;</code>] Calc --> Output[/<code>cout << "Площадь поверхности = " << _s << endl << "Объем параллелепипеда = " << _v << endl;</code>/] Output --> End([Конец]) </pre>

Таблица 1 – Выполнение кода и блок-схема 1 темы

2 БАЗОВЫЕ ТИПЫ ЯЗЫКА C++

Перед объявлением переменной или константы необходимо определить, какого она будет типа. Для этого нужно понимать цели, для которых она будет использована, какие значения она может принимать.

Тип данных определяет количество выделяемой памяти под переменную или константу, правила хранения данных этого типа, допустимые операции для данных этого типа.

Количество памяти, выделяемое под переменную, и правила хранения данных определяют диапазон значений данных конкретного типа данных.

Типы данных языка C++ можно разделить на две группы: базовые и пользовательские.

К базовым типам языка относят: `void`, `int`, `char`, `float`, `double`, `bool` (прил. 5).

К целым типам `int`, `short`, `long`, `char` применимы спецификаторы `signed` (знаковый), `unsigned` (беззнаковый).

Использование спецификаторов `short` и `long` перед типом данных `int` ведёт к уменьшению или увеличению количества выделяемой памяти.

Целые беззнаковые числа хранятся в прямом двоичном коде. Диапазон таких данных – от нуля до числа, двоичное представление которого состоит из ряда единиц, длина ряда – количество бит, отводимых под переменную соответствующего типа.

Таким образом, диапазон беззнаковых целых типов данных будет определён следующим образом: $0 - 2^n - 1$, где n – число разрядов, отводимых под число:

0	0	0	0	0	0	0	0	0 – минимальное значение
1	1	1	1	1	1	1	1	255 – максимальное значение

Знаковые целые числа хранятся в дополнительном коде. Старший бит хранит знак: единица – минус, ноль – плюс. Незначащие разряды, следующие за знаковым, заполняются нулями. Например, если под переменную отведено восемь разрядов, то двоичное число 1001 в памяти будет представлено так:

0	0	0	0	1	0	0	1
---	---	---	---	---	---	---	---

Дополнительный код отрицательного числа m равен $2^k - |m|$, где k – количество разрядов в ячейке. Для получения дополнительного кода отрицательного числа необходимо: – модуль отрицательного числа представить в прямом коде; – инвертировать значение всех бит числа, т. е. все нули заменить на единицы, а единицы – на нули; – к полученному обратному коду прибавить единицу. Получим 8-разрядный дополнительный код числа – 48:

00110000 – $|-48| = 48$ – в прямом коде;

11001111 – $|-48|$ – в обратном коде;

11010000 – -48 – в дополнительном коде.

Диапазон целых знаковых чисел – от -2^{n-1} до $2^{n-1} - 1$, где n – число разрядов, отводимых под переменную. Для хранения вещественных типов данных в C++ определены типы `float`, `double`, `long double`, отличающиеся количеством выделяемой памяти. При хранении этих типов данных часть разрядов отводится для записи порядка числа, остальные разряды – для записи мантиссы. Логические переменные могут принимать значения `true` и `false`. Тип таких переменных – `bool`. Тип данных `void` не имеет значений, он нужен в ряде ситуаций: например, когда правила языка требуют указания типа данных, но необходимости в таком типе нет (если отсутствует возвращаемое значение функции). Переменную типа `void` объявить нельзя. При работе с символами используют типы `char` и `wchar_t`. Для хранения символов набора из 256 символов ASCII применяют тип `char`. Для работы с символами, код которых занимает более одного байта, используют тип `wchar_t`. Для объявления

переменной необходимо указать её тип и имя, завершив объявление точкой с запятой:

```
<имя типа> <имя переменной>;  
int i;
```

Можно совместить объявление переменной с инициализацией:

```
<имя типа имя переменной> = <значение>;  
int i = 25;
```

При необходимости объявить несколько однотипных переменных их разделяют запятой:

```
int i, j;
```

При объявлении константы перед её именем необходимо указать ключевое слово `const`. Значение константы нужно задать при объявлении:

```
<имя типа> const <имя константы> = <значение>;  
int const N=7;
```

Правила именования в языке C++

Любой идентификатор (имя) должен начинаться с латинской буквы. Кроме того, имя может содержать цифры и символ подчёркивания. Язык программирования C++ регистрочувствительный. Это означает, что `int z;` и `int Z;` – объявление двух разных переменных. Имена переменных должны нести смысловую нагрузку, т. е. из названия переменной должно следовать, для каких целей она используется.

Место описания идентификатора задаёт его область действия. Переменная, описанная внутри блока, будет локальной, т. е. видна только в этом блоке от точки объявления и ниже. Блоком называют часть кода, ограниченную фигурными скобками, например, тело цикла. Переменную, объявленную вне блока, называют глобальной. Обращение к такой переменной возможно в модуле, в котором она объявлена, от точки объявления и ниже. Если имя локальной и глобальной переменных совпадают, то локальная перекрывает глобальную, т. е. обращение внутри блока будет вестись к локальной переменной.

При использовании в выражениях разнотипных операндов применяют явное или неявное приведение типов. При неявном приведении типа в операции присваивания тип правого операнда приводится к типу левого. При приведении вещественного типа к целочисленному дробная часть отбрасывается. При приведении целого типа к вещественному добавляется нулевая дробная часть. Например:

```
float r = 9.57;  
int i = r; //i=7  
i=w; //r=7.00
```

При использовании явного приведения типа перед операндом необходимо указать тип данных, к которому приводят исходный тип операнда. При этом важно понимать, что приведение типа осуществляется только в указанной точке, сама переменная, её значение и способ хранения данных при этом не изменяются. Явное приведение возможно двумя способами. Если есть уверенность, что приведение пройдет без потери данных, используют подход языка C. Перед идентификатором, тип которого приводится, указывают нужный тип в круглых скобках. Например:

```
float w = 4.35;  
int r = (int)w;
```

Если пользователь уверен в том, что переполнение не произойдет, то применяется оператор `static_cast`:

```
static_cast <тип> (<выражение>)  
int a = 54;  
char ch = i; // неявное преобразование
```

Такой подход приведет к предупреждающему сообщению во время компиляции. Для того чтобы избежать этого, лучше сделать так:

```
int a = 54;  
char ch = static_cast<char>(a);
```

Оператор `%` возвращает остаток от деления, оператор `%=` присваивает левому операнду остаток от деления левого операнда на правый. Соответственно, в результате выполнения выражения `m %= n`; целочисленная

переменная `m` примет значение частного текущего значения этой переменной и `n`.

Практическая часть

Задание: напишите программу для перевода сантиметров в дюймы. Для справки: в одном дюйме 2,54 сантиметра.

Листинг приложения:

```
void TaskIndivid13_block2() {
    double sant, duim;
    cout << "Введите значение см: ";
    cin >> sant;
    duim = sant * 2.54;
    cout << "В " << sant << " см: " << duim << " дюйма." << endl;
}

int main(){
    setlocale(LC_ALL, "rus");
    TaskIndivid13_block2();
}
```

Результат выполнения кода:	Блок-схема
<div>Задание № 1 (блок 4)</div> <div>Введите значение см: 43</div> <div>В 43 см: 109.22 дюйма.</div>	<pre> graph TD Start([Начало]) --> Init[double sant, duim;] Init --> Input[/cout << "Введите значение см: "; cin >> sant;/] Input --> Calc[duim = sant * 2.54;] Calc --> Output[/cout << "В " << sant << " см: " << duim << " дюйма." << endl;/] Output --> End([Конец]) </pre>

Таблица 2 – Результаты выполнения кода и блок-схема 2 темы

3 ОПЕРАТОРЫ ВЕТВЛЕНИЯ ЯЗЫКА C++

Условный оператор if

Оператор if позволяет реализовать алгоритмическую конструкцию «ветвление» на языке программирования C++. Оператор имеет полную и сокращённую формы. Последняя используется при отсутствии действий на ветке «иначе», т. е. при невыполнении условия.

Синтаксис оператора:

if (выражение) оператор1; [else оператор 2;]

Выполнение оператора начинается с вычисления выражения. Выражение может быть любым: арифметическим, логическим или сложным. В любом случае результат вычисления выражения будет интерпретирован как логическое значение. При этом ноль – «ложь», любое отличное от нуля значение – «истина».

Если результат вычисления выражения – «истина», то выполняется оператор1. В противном случае выполняется оператор2 – при использовании полной формы оператора if, или ничего не выполняется, если используется неполная форма оператора ветвления и оператор2 отсутствует. Оператором1 и оператором2 могут быть любые операторы языка программирования C++. Если на месте одного из операторов нужно выполнить несколько действий, используют составной оператор. Для этого эти действия заключают в фигурные скобки. Таким образом не нарушаются правила использования оператора if. Рассмотрим несколько примеров.

Полная форма оператора if:

```
int c = 7, d = 9, max = 0;
```

```
if (c > d) max = c; else max = d;
```

Неполная форма оператора if:

```
if (c > d) max = c;
```

Использование составного оператора:

```
if (c < d && (c > d || c == 0)) d++; else { d*= c; c++;}
```

Так как оператор1 и оператор2 могут быть любыми, значит, на их месте может быть использован оператор if. Такое ветвление называют вложенным, например:

```
int c = 6, d = 7, f = 9, max = 0;
if (c > d) {if (c > f) max = c; else max = f;}
else if (d > f) max = d; else max=f;
cout << "max="<<max;
```

Оператор множественного ветвления switch

В ситуациях, когда выражение является целочисленным и возможны более чем два варианта ответа, целесообразнее использовать оператор множественного ветвления switch.

Правило использования оператора следующее:

```
switch (<целочисленное выражение>)
{
    case константное выражениеN-1: операторN-1;
    [default: <оператор N>]
}
```

После вычисления выражения его значения последовательно сравнивают со значениями констант. Все константы, используемые в операторе, должны быть уникальными, т. е. не должно быть повторений. Сравнение происходит до первого совпадения. Если совпадение найдено, то последовательно выполняют все операторы, без проверки на совпадение, включая оператор, указанный после ключевого слова default. Если необходимо выполнить только действия, относящиеся к определённой константе, следует использовать составной оператор. Последним действием этого оператора должен быть оператор break, передающий управление оператору, следующему за оператором switch.

Если значение выражения не совпало ни с одним из значений указанных констант, будет выполнен оператор, указанный после ключевого слова default.

Ключевое слово `default` может отсутствовать: в этом случае никакие действия не будут выполнены, если значение выражения не совпало ни с одной константой.

Рассмотрим простой и наглядный пример:

```
int i = 1; switch (++i)
{
case 1:printf("1\n");
case 2:printf("1+1=2\n");
case 3:printf("2+1=3\n");
case 4:printf("3+1=4\n"); default:printf("No!\n");
}
```

Результат выполнения этого фрагмента кода будет следующим:

1 + 1 = 2

2 + 1 = 3

3 + 1 = 4

No!

Если в рассмотренном выше примере после каждого оператора использовать оператор `break`, то результат будет другим:

```
int i = 1; switch (++i)
{
case 1: {printf("1\n"); break; }
case 2: {printf("1+1=2\n"); break; }
case 3: {printf("2+1=3\n"); break; }
case 4: {printf("3+1=4\n"); break; }
default:printf("No!\n");
}
```

В результате выполнения этого фрагмента на экране появится следующая строка: 1 + 1 = 2.

Если значение переменной `i` задать отрицательным, то на экране появится строка «No!».

Тернарная операция

Тернарная операция имеет три операнда. Эту операцию используют, когда необходимо не только реализовать ветвление, но и вернуть некоторое значение в точку вызова. Синтаксис операции следующий:

`(<операнд1>)?<операнд2>:<операнд3>;`

Операнд1 – выражение, результат которого интерпретируется логически. Если результат вычисления выражения – «истина», будет выполнен операнд2, иначе – операнд3.

Операнд2 и операнд3 представляют собой операторы. Результат выполнения соответствующего оператора – результат тернарной операции, это значение будет возвращено в точку вызова.

Рассмотрим примеры применения тернарной операции.

```
float y = 3.2; float x = -2.9;
```

```
float z = (y < x) ? y++ : x++; cout << "z =" << z << endl; cout << "x=" << x << endl;
```

```
cout << "y=" << y;
```

В результате выполнения этого фрагмента кода на экране появятся следующие строки:

```
z = -2.9
```

```
x = -1.9
```

```
y = 3.2
```

Рассмотрим менее очевидный пример:

```
int m=1, i, j, k; i = 6; j = 9;
```

```
k=i<j?i++:++j?m=i+j, printf("i=%i\n", i) : i;
```

```
cout<<"i="<<i<<endl;
```

```
cout<<"j="<<j<<endl;
```

```
cout<<"m="<<m<<endl;
```

```
cout<<"k="<<k<<endl;
```

В результате выполнения на экран будут выведены следующие значения:

```
i = 7
```

```
j = 9
m = 1
k = 6
```

Так как первый операнд тернарной операции имеет значение «истина», выполнится второй операнд $i++$, возвращаемое значение – 7 – будет сохранено в переменной k . Третий операнд выполнен не будет, поэтому значения переменных j и m не будут изменены.

Практическая часть

Задание: ученик вводит в программу два вещественных числа: сначала свой балл на экзамене, а затем – балл, который он получил в полугодии. Разбалловка в его школе следующая: А – 90,1 – 100 В – 74,4 – 90 С – 60,1 – 74,3 FХ – <60,1 Общая оценка считается как сумма баллов за экзамен и за полугодие. Вычислите общий балл и выведите на экран сначала его, а затем – ту оценку, которую получил ученик.

Листинг приложения:

```
void TaskIndivid13_block3() {
    double x, y, total;
    char check;
    cout << "Балл за экзамен: ";
    cin >> x;
    cout << "Балл за полугодие: ";
    cin >> y;
    total = (x + y) / 2;
    if (total >= 90.1 && total <= 100) {
        check = 'A';
        cout << "Общий балл: " << total << endl << "Его оценка: " << check <<
endl;
    }
    else if (total >= 74.4 && total <= 90) {
        check = 'B';
        cout << "Общий балл: " << total << endl << "Его оценка: " << check <<
endl;
    }
    else if (total >= 60.1 && total <= 74.3) {
        check = 'C';
        cout << "Общий балл: " << total << endl << "Его оценка: " << check <<
endl;
    }
    else {
        check = 'F';
    }
}
```

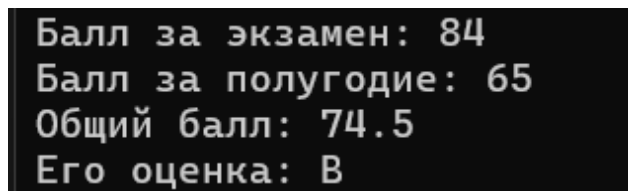
```

        cout << "Общий балл: " << total << endl << "Его оценка: " << check <<
endl;
    }
}

int main(){
    setlocale(LC_ALL, "rus");
    TaskIndivid13_block3();
}

```

Результат выполнения кода:



```

Балл за экзамен: 84
Балл за полугодие: 65
Общий балл: 74.5
Его оценка: В

```

Рисунок 1 – Выполнение кода 3 темы

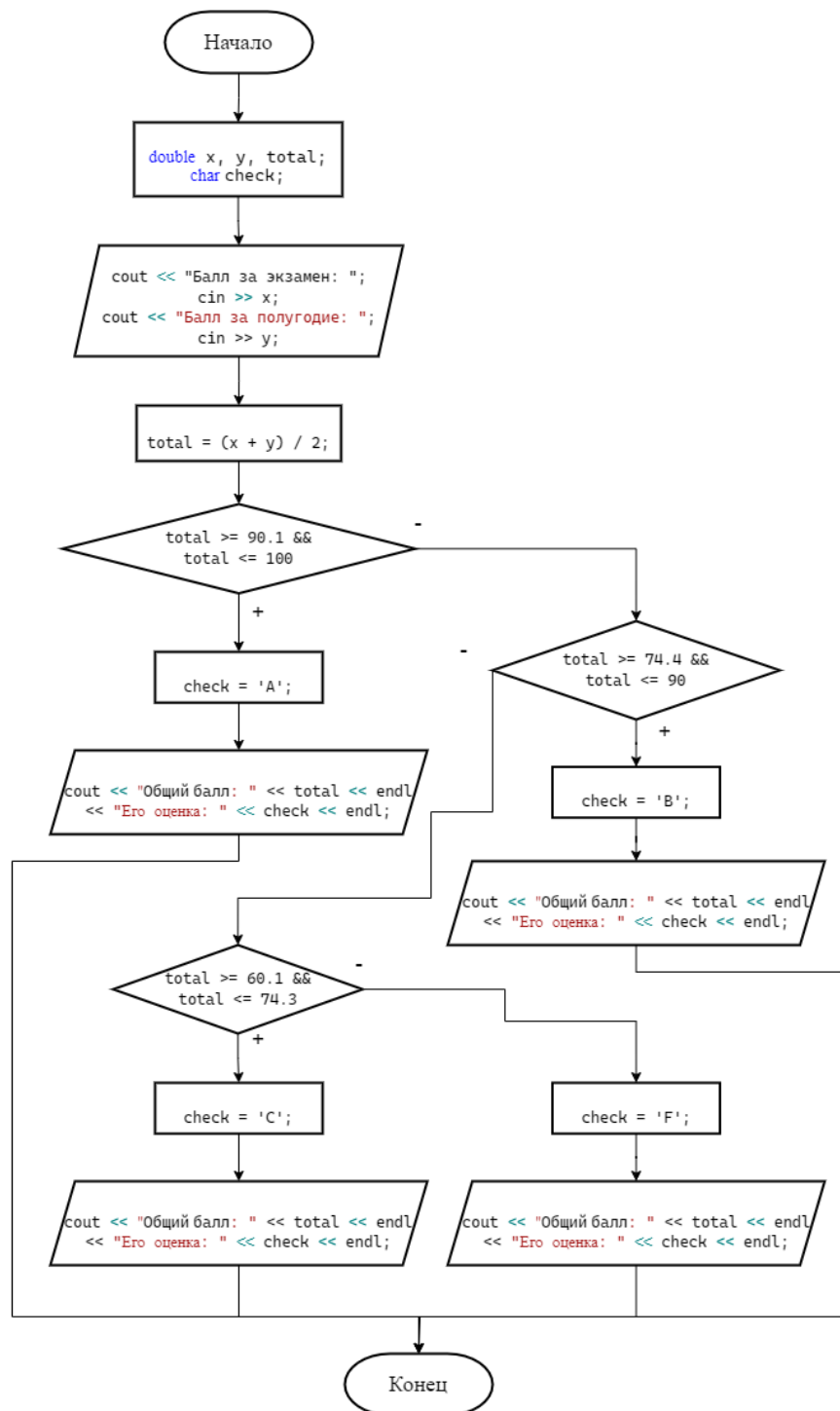


Рисунок 2 – Блок-схема задания 3 темы

4 ОПЕРАТОРЫ ОРГАНИЗАЦИИ ЦИКЛОВ ЯЗЫКА C++

Алгоритмическую конструкцию «повторение» реализуют при помощи операторов организации цикла.

В языке программирования C++ существует три оператора организации циклов: `while`, `do while` и `for`.

Независимо от того, каким оператором реализован цикл, необходимо соблюсти три этапа работы с ним: задание начальных значений и/или параметров цикла, выполнение тела цикла, проверка условия продолжения цикла.

Телом цикла называют многократно повторяющиеся действия.

Итерацией называют однократное выполнение тела цикла.

Условие продолжения цикла – выражение, в зависимости от значения которого будет выполнена следующая итерация и осуществлён выход из цикла.

В зависимости от взаимного расположения условия продолжения цикла и тела цикла различают цикл с предусловием (условие расположено перед телом цикла) и цикл с постусловием (условие расположено после тела цикла).

Структурные схемы цикла с предусловием и постусловием представлены в прил. 4.

Если из условия задачи ясно, что тело цикла необходимо выполнить как минимум один раз, используют цикл с постусловием; если же возможна ситуация, в которой тело цикла не выполнится ни разу, используют цикл с предусловием.

Параметрами цикла называют переменные, используемые в условии продолжения цикла, счётчиком цикла – целочисленные параметры цикла, изменяющиеся с определённым шагом.

Для того чтобы цикл работал корректно, в его теле обязательно должно быть предусмотрено изменение параметров цикла. Начальное значение всех

обрабатываемых в цикле переменных, включая параметры цикла, должно быть задано до тела цикла.

Цикл с предусловием `while`

Оператор `while` позволяет реализовать цикл с предусловием.

Синтаксис оператора:

`while (выражение) оператор;`

Если результат вычисления выражения «истина», будет выполнено тело цикла. Если результат вычисления выражения «ложь», то управление будет передано оператору, следующему за телом цикла. В случаях, когда тело цикла должно содержать более одного оператора, используют фигурные скобки, объединяющие последовательность операторов в один оператор, называемый составным. Таким образом, требование к тому, чтобы после выражения стоял оператор «тело цикла», не нарушается.

Пример использования оператора `while` для вычисления произведения пяти введенных с клавиатуры чисел:

```
int p = 1;
int const n = 5; int a = 0, i = 1; while (i <= n)
{
    printf("a=");
    scanf_s("%i", &a); p = p * a; i++;
}
printf("p=%i\n", p);
```

Цикл с постусловием `do...while()`

Оператор `do...while` позволяет реализовать цикл с постусловием. Использовать этот оператор следует в ситуациях, когда из условия задачи понятно, что тело цикла необходимо выполнить как минимум один раз.

Синтаксис оператора:

`do <оператор> while (выражение);`

После выполнения оператора «тело цикла» вычисляется выражение. В зависимости от результата вычисления выражения будет осуществлён переход на следующую итерацию цикла, если результат

«истина», или выход из цикла, если результат – «ложь».

В качестве примера вычислим произведение пяти вводимых с клавиатуры элементов с помощью оператора do...while:

```
int const n = 5;
int p = 1, a = 0, i = 1; do {printf("a=");
scanf_s ("%i", &a); p = p * a; i++;
} while (i < n); printf("p=%i\n",p);
```

Цикл с параметром for

Оператор for позволяет реализовать цикл с предусловием и условием продолжения. Синтаксис оператора:

for (выражение1; выражение2; выражение3;) оператор; Выражение1 называют инициализатором. Инициализатор содержит объявления и инициализацию переменных – параметров цикла, выполняется один раз до начала выполнения цикла. Инициализатор может содержать несколько операторов, разделённых запятой.

Выражение2 – условие продолжения цикла. Если результат вычисления выражения – «истина», то выполняется тело цикла, в противном случае управление передаётся оператору, следующему за оператором for.

Выражение3 – итератор, обычно содержит оператор, изменяющий значение параметра цикла. Итератор будет выполнен после всех операторов тела цикла. Далее снова осуществляется проверка условия – выражения2.

Если тело цикла содержит более одного оператора, то его заключают в фигурные скобки.

Вычислим произведение пяти введенных с клавиатуры чисел с использованием оператора for:

```
int const n = 5;
int p = 1, a = 0, i = 1;
for (int i = 0; i < n; i++)
{printf("a="); scanf_s ("%i", &a); p*=a; }
printf("p=%i\n", p);
```

Любое из выражений, указанных в круглых скобках после ключевого слова `for`, может быть пропущено. При этом возможна ситуация, в которой все три выражения отсутствуют. Какое бы из выражений ни было опущено, наличие точки с запятой, завершающей это выражение, обязательно.

Если отсутствует первое выражение, инициализация параметра цикла должна быть выполнена до ключевого слова `for`. Если отсутствует второе выражение, выход из цикла необходимо предусмотреть в теле цикла. Обычно для этого используют оператор `break`, передающий управление оператору, следующему сразу за циклом. Если отсутствует третье выражение, необходимо включить итератор в тело цикла.

Внесём изменения в рассмотренный выше пример, опустив все три выражения:

```
int p = 1;
int const n = 5; int a = 0;
int i = 1; for(;;)
{ if (i > n) break; printf("a=");
  scanf_s("%i", &a); p = p * a; i++;
}
printf("s=%i\n", s);
```

При необходимости осуществить досрочный выход из цикла используют оператор `break`. После того как оператор `break` будет встречен в теле цикла, управление будет передано оператору, следующему за циклом. Таким образом, операторы, стоящие в теле цикла после ключевого слова `break`, выполнены не будут.

Если в любом из рассмотренных операторов цикла необходимо выполнить досрочный выход, то используют операторы `break` и `return`.

Разберём пример с использованием в теле цикла оператора `break`. Будем вычислять сумму пяти вводимых с клавиатуры чисел. Если на *i*-м шаге сумма превысит некоторое значение *N*, то будет выполнен досрочный выход из цикла:

```
int s = 0;
```



```

int const n = 5; int const N = 15; int a= 0;
int i = 1; while (i<=n)
{
printf("a=");
scanf_s("%i",&a); s = s + a; i++;
if (s>N) break;
}
printf("s=%i\n", s);

```

Использование оператора return приводит к досрочному выходу из функции, в теле которой он был использован.

Использование в теле цикла оператора continue позволяет перейти на следующую итерацию, минуя оставшиеся операторы тела цикла:

```

for (int count = 0; count <= 30; count++)
{if ((count % 5) != 0) continue;
std::cout << count << std::endl; /* Это дей- ствие будет пропущено, если
число не делится нацело на 5*/}

```

В результате выполнения этого цикла на экран будет выведен столбец чисел от 0 до 30, делящихся нацело на 5.

Практическая часть

Задание: пользователь вводит целое неотрицательное число N. Выведите все нечетные числа от 1 до N включительно в одну строку. Оператор ветвления в задаче не использовать.

Листинг приложения:

```

void TaskIndivid13_block4() {
int N;
cout << "Введите целое неотрицательное число: ";
cin >> N;
for (int i = 1; i <= N; i += 2) {
cout << i << "\t";
}
cout << endl;
}

```

```
int main(){
    setlocale(LC_ALL, "rus");
    TaskIndivid13_block4();
}
```

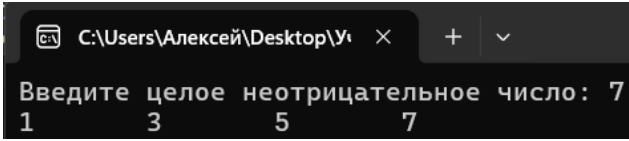
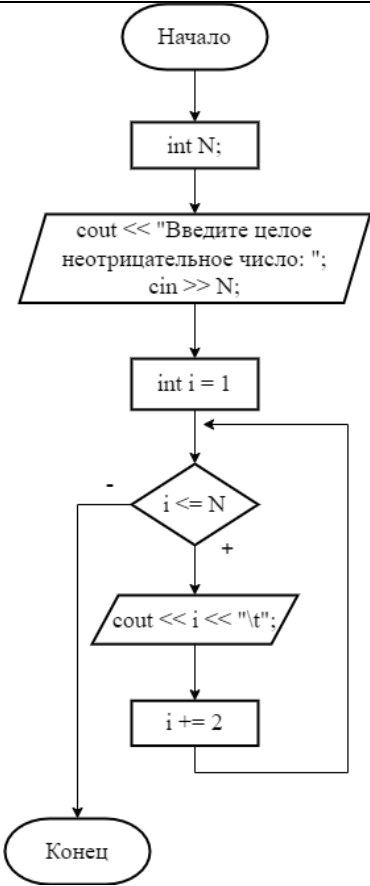
Результат выполнения кода:	Блок-схема
	 <pre> graph TD Start([Начало]) --> N[int N;] N --> Input[/cout << "Введите целое неотрицательное число: "; cin >> N;/] Input --> I1[int i = 1] I1 --> Cond{i <= N} Cond -- "+" --> Output[/cout << i << "\n"/] Output --> Inc[i += 2] Inc --> Cond Cond -- "-" --> End([Конец]) </pre>

Таблица 3 – Выполнение кода и блок-схема 4 темы

5 УКАЗАТЕЛИ И ССЫЛКИ

Указатель – производный тип данных, хранящий адрес какой-либо ячейки памяти. Нельзя объявить просто переменную – указатель. Указатель всегда связан с определённым типом данных, типом указуемого.

Синтаксис объявления указателя:

<тип данных> *<имя указателя>;

Пример: `int *p`; `p` – указатель на целое число.

Для инициализации указателя в правой части оператора присваивания нужно указать адрес конкретной области данных.

Если указатель настраивают на объявленные ранее переменную или константу, то перед соответствующим идентификатором ставят символ `&` – операция взятия адреса:

`int q = 7; int* p; p = &q;`

Объявление указателя может быть совмещено с инициализацией.

Пример:

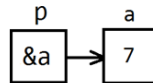
`int a = 7; int* p = &a;`

Результат работы приведённого примера можно схематично представить, как показано на рис. 1.

Стрелка на рис. 1 означает, что указатель `p` содержит адрес переменной `a`, т. е. указывает на переменную `a`.

Через указатель возможна работа как с самим указателем, так и со значением указуемого.

Для работы с указуемым необходимо совершить переход по адресу. Для этого используют операцию разадресации (разыменовывание). Для разадресации перед уже объявленным и проинициализированным указателем ставят символ `*`.



Пример: `int a =7; int* p=&a;`

`*p=5;`

В приведённом примере в указуемое – переменную `a` – записано значение 5.

Указатель может быть константным. Значение константного указателя нельзя изменить, т. е. его нельзя перенастроить, а изменить значение указуемого (если указуемое не константа) можно. Для объявления константного указателя перед его именем указывают ключевое слово `const`.

Пример:

`int b=10;`

`int * const p =&b;`

При необходимости объявить указатель на константу ключевое слово `const` следует указать до или после типа указуемого, но строго до символа `*`.

Пример:

`const int b=10; int const *p =&b; const int c = 30; p = &c;`

`printf ("*p=%d", *p); // *p=30`

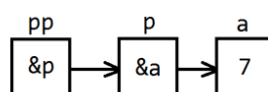
В рассмотренном примере `p` – указатель на целочисленную константу, значение `p` можно изменить. В примере указатель `p` перенастроен на другую константу того же типа – `c`.

Так как указатель – это тип данных, значит, он может также быть типом указуемого, т. е. возможно объявление указателя на указатель.

Пример:

`int a = 7; int* p = &a; int** pp = &p;`

`**pp = 10;`



В рассмотренном примере переменная `pp` – указатель на указатель на целое – настроена на указатель на целое. После выполнения приведённого выше фрагмента кода значение переменной `a` будет равно 10.

Операции над указателями

На указателях определены следующие операции: сравнения, вычитания, сложения с константой, инкремента и декремента. Для получения наиболее наглядного результата эти операции удобно продемонстрировать, настраивая указатели на элементы массива.

Под массивом понимают поименованную последовательность элементов, расположенных в памяти подряд. Ниже приведен пример объявления и инициализации массива из пяти целых чисел:

```
int mas[5] = {1,2,3,4,5};
```

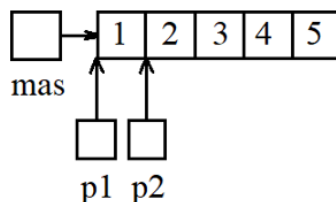
В языке C++ нумерация элементов массива начинается с нуля.

Объявим два указателя на целое `p1` и `p2` и настроим их на элементы массива `mas`. Настроим `p1` на нулевой элемент массива, `p2` – на первый элемент массива:

```
int *p1 = &mas[0]; // указатель настроен на нулевой элемент
int *p2 = &mas[1]; // указатель настроен на первый элемент
```

В языке программирования C++ имя массива рассматривается как константный виртуальный указатель; с учётом сказанного настройку указателей правильнее выполнять следующим образом:

```
int *p1 = mas; int *p2 = mas+1;
```



Если, используя операции сравнения, сравнить два указателя, то будут сравниваться значения указателей, т. е. адреса указуемых.

Пример:

```
if (p1 < p2) printf("p1\n"); else printf("p2\n");
```

В данном случае ($p1 < p2$) – «истина», так как указатели настроены на последовательно расположенные элементы, адрес указуемого $p2$ старше.

Операция присвоения одного указателя другому. `int* p3 = p2;` // указатель $p3$ будет настроен туда же, куда настроен указатель $p2$.

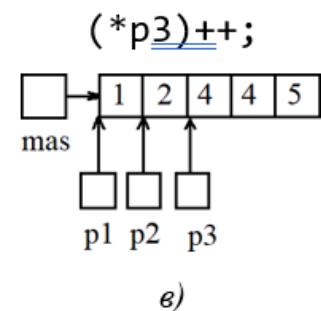
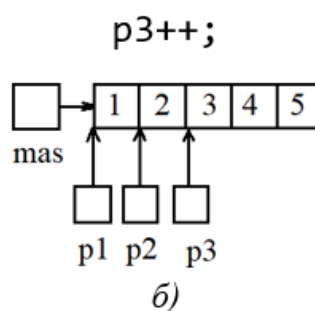
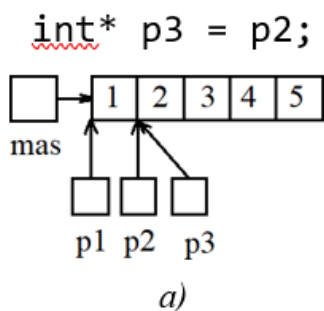
Унарные инкремент и декремент, сложение указателя с целочисленной константой. Операции постфиксного и префиксного инкремента и декремента изменяют значение указателя на единицу. За единицу принимается размер типа указуемого. Таким образом, указатель будет перенастроен:

`p3++;` // значение $p3$ увеличится на один размер типа указуемого.
`(*p3)++;` // увеличение на единицу значения указуемого. `p3+=4;` // указатель будет перенастроен, его значение увеличит-

ся на четыре единицы, под единицей в данном случае понимают размер типа указуемого.

Вычитание указателей. Возвращаемое значение операции вычитания указателей – целое число. Это число – количество ячеек указуемого типа между указуемыми:

`int d = p3 - p1;`



`int* p3 = p2;`

`p3++;`

`(*p3)++;`

Схематичное представление операций на указателях: а – настройка указателя $p3$ на ячейку `mas[2]`; б – сдвиг $p3$ на одну ячейку размера указуемого; в – изменение значения указуемого

Ссылки

Ссылка, как и указатель, содержит адрес переменной, с которой связана. Инициализация ссылки обязательна при объявлении:

<тип данных> & <имя ссылки> = <переменная>;

Наибольшее применение ссылки находят при передаче параметров в функции. При использовании ссылки операции производятся с переменной, с которой она связана. Ссылку нельзя перенастроить.

Пример:

```
float F=9.5; float H=7.43;
```

```
int &ref=F; // объявление ссылки ref и связь её с переменной.
```

```
ref = H; /*переменной F присвоено значение переменной H, ссылка ref остаётся связанной с переменной F*/
```

```
ref++; // изменение значения переменной, связанной с ref
```

Практическая часть

Задание: объявите и проинициализируйте целочисленную переменную. Объявите указатель на целое и настройте его на эту переменную. Обратившись к адресуемому через указатель, увеличьте значение первого втрое и выведите его на экран.

```
void task5_4() {  
    int a;  
    cout << "Введите число: ";  
    cin >> a;  
    int* p = &a;  
    *p *= 3;  
    cout << "Значение умноженное на 3: " << *p << endl;  
}  
  
int main()  
{task5_4();}
```

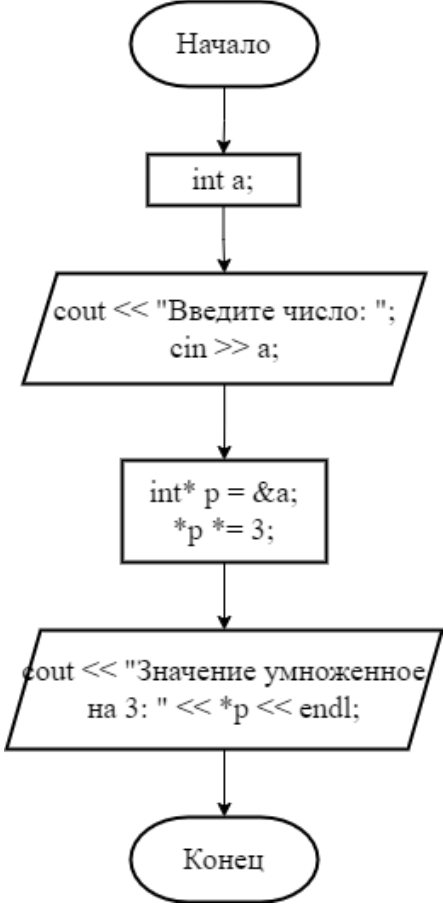
Результат выполнения кода:	Блок-схема
<div data-bbox="263 622 844 712" style="background-color: black; color: white; padding: 10px;"> Введите число: 5 Значение умноженное на 3: 15 </div>	 <pre> graph TD Start([Начало]) --> Decl[<code>int a;</code>] Decl --> Input[/<code>cout << "Введите число: "; cin >> a;</code>/] Input --> Calc[<code>int* p = &a; *p *= 3;</code>] Calc --> Output[/<code>cout << "Значение умноженное на 3: " << *p << endl;</code>/] Output --> End([Конец]) </pre>

Таблица 4 – Выполнение кода и блок-схема 5 темы

6 МАССИВЫ

Массивом в программировании называют поименованную последовательность однотипных элементов, расположенную в памяти подряд.

Для объявления массива нужно указать тип данных его элементов, имя массива и количество элементов в квадратных скобках:

<тип элементов массива> <имя массива> [<количество элементов>];

Нумерация ячеек массива в языке программирования C++ начинается с нуля, следовательно, номер последнего элемента массива на единицу меньше количества элементов в массиве. Если в массиве N элементов, то номер последнего элемента $N - 1$.

Объявим массив из пяти целых чисел:

```
int mas [5];
```

Количество элементов массива при его объявлении должно быть обязательно задано целочисленной константой. Это связано с тем, что при выделении памяти под массив в момент компиляции необходимо иметь точную информацию о количестве его элементов и это количество не должно изменяться во время работы. При этом константа может быть как именованной, так и не именованной, как в приведённом выше примере.

Предпочтительнее вариант с именованной константой:

```
int const N=5; int mas [N];
```

Такой подход ведёт к минимуму изменений в коде программы при необходимости создания массива из другого количества элементов. Для этого будет достаточно изменить значение константы при объявлении.

Задать значение элементов массива можно тремя способами: при объявлении, в цикле и поэлементно. При объявлении это будет выглядеть следующим образом:

```
int const N=5;  
int mas [N]={1,2,3,4,5};
```

Если список значений содержит меньше элементов, чем их количество, указанное в квадратных скобках, то указанными значениями будут проинициализированы первые элементы массива. Оставшиеся элементы получат значение по умолчанию для соответствующего типа данных.

Пример:

```
int const N=5; int mas [N] = {1,2};
```

В случае, когда список инициализации содержит больше необходимого числа значений, на шаге компиляции возникает ошибка:

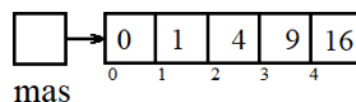
```
int mas1 [3] = {1,2,3,4,5,6,7,8,9}; // ошибка
```

Для обращения к элементам массива используют два подхода:

Через индексное выражение, например:

```
mas[3]=45;
```

Значение элемента с индексом 3 будет изменено на 45.



При инициализации элементов массива в цикле организуют перебор элементов – обычно от наименьшего индекса к наибольшему.

Объявим массив из пяти целых чисел и проинициализируем его в цикле значениями индексов соответствующих элементов:

```
*mas = 9;
```

Таким образом, нулевой ячейке массива будет присвоено значение 9.

Объявим массив из пяти целых чисел и проинициализируем его в цикле каждым из описанных способов обращения к элементам:

```
int mas1[5]; // объявление массива
```

```
for (int i = 0; i < 5; i++)
```

```
{mas1[i] = i*i; cout << mas1[i]<<" ";} // обращение к элементам через
```

индексное выражение.

```
int const M = 5;
```

```
int mas2[M]; // объявление второго массива
```

```
for (int i=0; i<M; i++)
```

```

{ // обращение к элементу массива через указатель
*(mas+i)=i*i; printf("%i ",mas[i]);
}

```

Размерность массива определяется количеством квадратных скобок после его имени:

```

<тип элементов имя массива> [<целочисленная константа1>][ <целочисленная константа2>];

```

Пример объявления двумерного массива:

```

int const M = 5; int const N = 3; int mas2[N][M];

```

Пример объявления трёхмерного массива:

```

int mas3[N][M][N];

```

Элементы многомерных массивов в памяти располагаются подряд построчно (рис. 6). При переходе к следующему элементу первым изменяется последний индекс:

```

mas[0][0]=2; // запись значения в ячейку с индексами 0, 0.

```

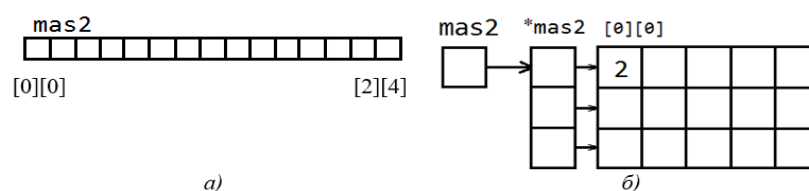
Для задания значений многомерного массива при объявлении значения указывают в фигурных скобках списком через запятую.

Пример:

```

int mas2[3][2] = {1,2,3,4,5,6 };

```



Пример вывода элементов двумерного массива в цикле:

```

for (int i = 0; i < 3; i++)
{
for (int j = 0; j < 2; j++) printf("%i ", mas2[i][j]); printf("\n");
}

```

Пример обращения к элементам двумерного массива:

```

mas2[1][1] = 2;
*(*(mas2+1)+2) = 3;

```

Динамическое распределение памяти

До настоящего момента речь шла о статических переменных. Так называют переменные, память под которые выделяется на этапе компиляции.

Переменные, память под хранение которых выделяется во время работы программы, называют динамическими.

Динамически распределяемую память называют кучей. Работа с динамическими переменными происходит через указатели.

Для выделения динамической памяти используют оператор new: new <тип данных> [<количество элементов>];

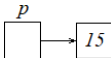
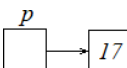

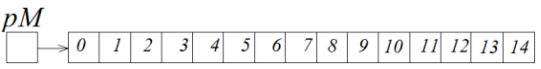
или так: new <тип данных> (<инициализатор>);

В точку вызова оператор new возвращает указатель на начало области захваченной памяти.

В табл. 5 приведены примеры работы с динамическими данными, память под которые захвачена с помощью оператора new, и схематичное пояснение произведённых действий.

Таблица 5

Работа с динамически захваченной памятью

Фрагмент кода	Схематичное представление
<code>int* p = new int(15);</code>	
Объявлен указатель на целочисленную переменную, выделена динамическая память под переменную типа <code>int</code> , в выделенную ячейку записано значение 15. Адрес начала захваченной области памяти записан в указатель <code>p</code> .	
<code>*p += 2;</code>	
Значение указываемого увеличено на 2/	
<code>int* pM = new int[15];</code>	
Объявлен указатель на целочисленную переменную, выделена динамическая память под массив из 15 целых чисел. Адрес его записан в указатель <code>pM</code> .	
<code>for (int i = 0; i < 15; i++) {pM[i] = i;}</code>	
Инициализация в цикле элементов динамического массива.	

Важно помнить, что динамическая память, захваченная при помощи new, должна быть освобождена. Для этого используют delete:

```
delete[] <указатель>;  
delete p;  
delete []pM;
```

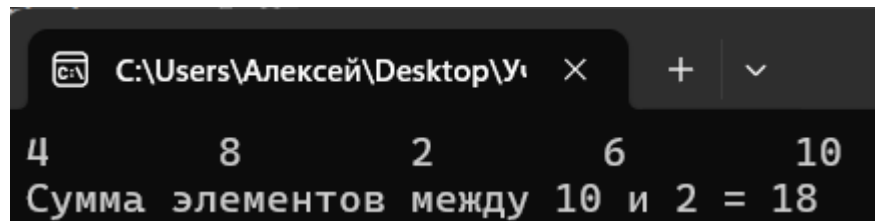
Практическая часть

Задание: Найдите сумму элементов массива, расположенных между максимальным и минимальным элементами (включительно).

Листинг приложения:

```
void TaskIndivid13_block6() {  
    const int n = 5;  
    int arr[n] = { 4, 8, 2, 6, 10 };  
    int max = -100500, min = 100500, maxIndex, minIndex, sum = 0;  
    for (int i = 0; i < n; i++) {  
        if (arr[i] > max) {  
            max = arr[i];  
            maxIndex = i;  
        }  
        if (arr[i] < min) {  
            min = arr[i];  
            minIndex = i;  
        }  
    }  
    for (int i = 0; i < n; i++) {  
        cout << arr[i] << "\t";  
    }  
    int _max = minIndex < maxIndex ? minIndex : maxIndex;  
    int _min = minIndex < maxIndex ? maxIndex : minIndex;  
    for (int i = _max; i <= _min; i++) {  
        sum += arr[i];  
    }  
    cout << "\nСумма элементов между " << max << " и " << min << " = " << sum <<  
endl;  
}  
  
int main(){  
    setlocale(LC_ALL, "rus");  
    TaskIndivid13_block6();  
}
```

Результат выполнения кода:



The screenshot shows a code editor window with a dark theme. The title bar at the top reads "C:\Users\Алексей\Desktop\Уч" followed by a close button (X) and window control buttons (+ and v). The code area contains two lines of text: the first line lists the numbers "4", "8", "2", "6", and "10" spaced out; the second line reads "Сумма элементов между 10 и 2 = 18".

```
4      8      2      6      10
Сумма элементов между 10 и 2 = 18
```

Рисунок 3 – Выполнение кода 6 темы

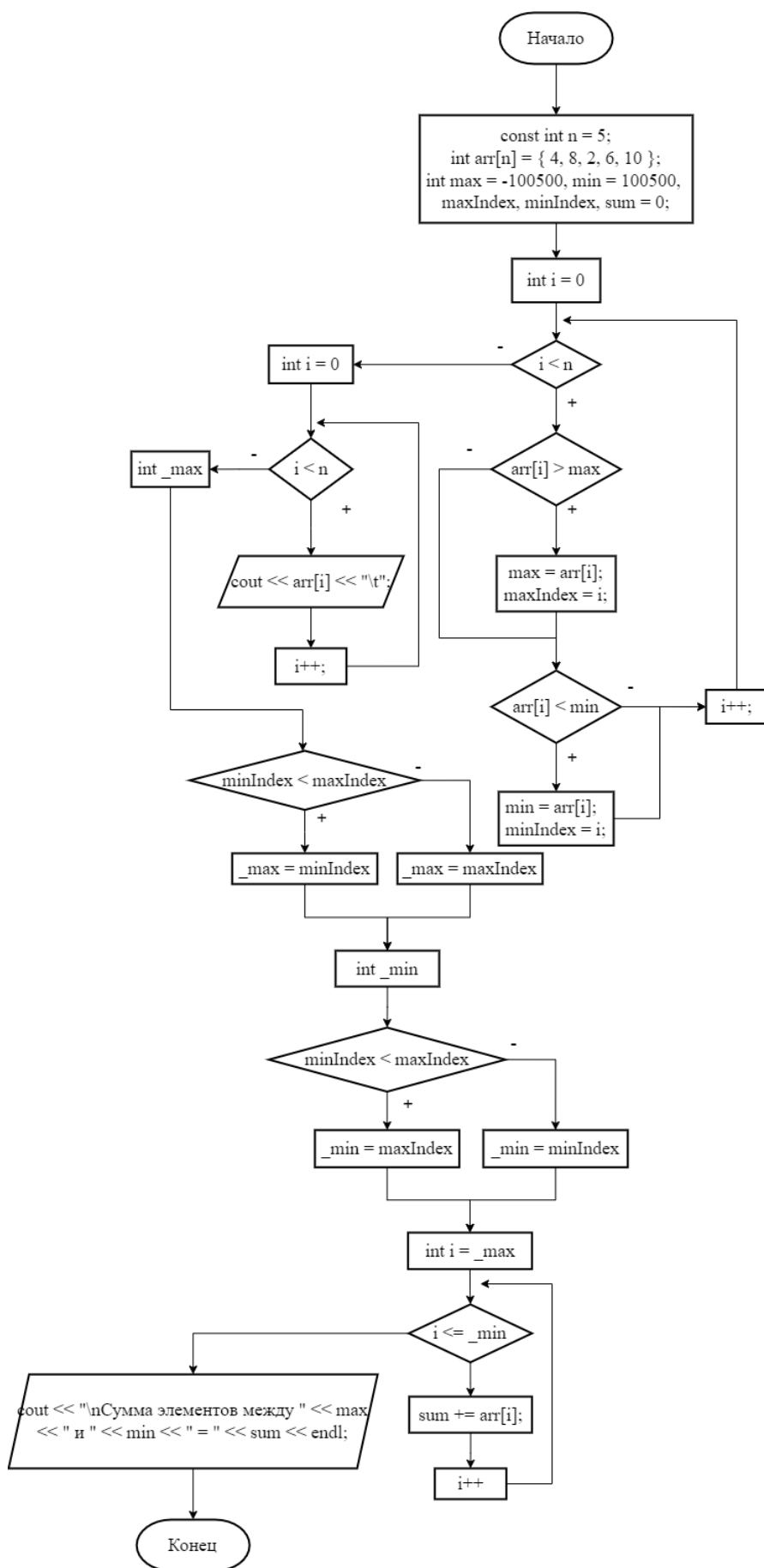


Рисунок 4 – Блок-схема 6 темы

7 ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ ДАННЫХ. СТРУКТУРЫ

Тип данных, который описывает программист, называют пользовательским. Описание пользовательского типа происходит по определённым правилам исходя из условий конкретной задачи.

Структура – пользовательский тип данных, объединяющий под одним именем разнотипные переменные, называемые полями.

Описание типа «структура» начинается с ключевого слова `struct` и указания имени описываемого типа. Далее в фигурных скобках следует описание полей. Завершается описание типа точкой с запятой.

Пример:

```
struct <имя типа> {<описание полей>};  
struct TypeExample  
{  
    char pole1; float pole2; int pole3[10];  
};
```

Объявление переменной описанного типа подчиняется тем же правилам, что и объявление переменной любого другого типа:

```
<имя типа> <имя переменной>;
```

Пример:

```
TypeExample Ex1, Ex2;
```

Объявить переменную типа «структура» можно только после описания типа, однако во время описания возможно объявление поля – указатель на описываемый тип.

Объявление переменных может быть совмещено с описанием типа данных. В этом случае, после закрывающей фигурной скобки, до точки с запятой перечисляют объявляемые переменные:

```
struct TypeExample  
{
```



```
int pole1; float pole2; char pole3[10];  
} Ex; // Ex – переменная типа TypeExample
```

Поля структуры можно задать списком или обращаясь к каждому полю через уточняющие идентификаторы.

При инициализации списком в фигурных скобках через запятую перечисляют необходимые значения полей в порядке их объявления:

```
struct TypeExample  
{  
    int pole1; float pole2; char pole3[10];  
} ST{1, 2.3, "name"}; # используя копирование ST_Example ST1{1,  
3.14,"name1"}, ST2={10,2.4,"name1"};
```

Для обращения к полю структуры необходимо уточнить идентификатор – имя структуры, указав через точку имя поля:

<имя переменной – структуры>.<имя поля>

Пример:

```
ST1.pole1 = 17;
```

Допустимо копирование одной структуры в другую через оператор присваивания:

```
ST1 = ST2;
```

То же самое можно сделать, обратившись к каждому полю:

```
Ex1.pole1 = Ex2.pole1; Ex1.pole2 = Ex2.pole2;
```

Так как структура – это тип данных, возможно объявление массива структур и указателя на структуру:

```
ST_Example massiv[5]; ST_Example* pST = &Ex1;
```

Практическая часть

Задание: опишите структуру с именем NOTE, содержащую следующие поля: – Фамилия, имя; – Номер телефона; – Дата рождения (массив из трех чисел). Введите с клавиатуры данные в массив, состоящий из восьми

элементов типа NOTE. Упорядочите элементы массива по датам рождения. Выведите на экран информацию о человеке, номер телефона которого введен с клавиатуры. Если такого нет, выдайте на экран соответствующее сообщение.

Листинг приложения:

```
struct NOTE
{
    string fullName;
    string phoneNumber;
    int date[3];
};

void SortedNotes(NOTE* student, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if ((student[j].date[0] > student[j + 1].date[0]) &&
                (student[j].date[1] > student[j + 1].date[1]) && (student[j].date[2] >
                student[j + 1].date[2])) {
                string temp = student[j].fullName;
                student[j].fullName = student[j + 1].fullName;
                student[j + 1].fullName = temp;
            }
        }
    }
}

void TaskIndivid13_block7() {
    const int n = 8;
    NOTE student[n];
    for (int i = 0; i < n; i++) {
        cout << "Введите ФИО: ";
        getline(cin, student[i].fullName);
        cout << "Введите номер телефона: ";
        cin >> student[i].phoneNumber;
        cout << "Введите дату рождения (через пробел): ";
        cin >> student[i].date[0] >> student[i].date[1] >> student[i].date[2];
        cin.ignore();
    }
    SortedNotes(student, n);
    string searchPhoneNumber;
    cout << "Введите номер телефона для поиска: ";
    cin >> searchPhoneNumber;
    bool found = false;
    for (int i = 0; i < n; i++) {
        if (student[i].phoneNumber == searchPhoneNumber) {
            found = true;
            cout << "ФИО: " << student[i].fullName << "\n";
            cout << "Номер телефона: " << student[i].phoneNumber << "\n";
            cout << "Дата рождения: " << student[i].date[0] << "-" <<
            student[i].date[1] << "-" << student[i].date[2] << "\n";
        }
    }
    if (!found) {
        cout << "Человек с таким номером телефона не найден.\n";
    }
}

int main(){
    setlocale(LC_ALL, "rus");
    TaskIndivid13_block7();
}
```

}

Результат выполнения кода:

```
Введите ФИО: Rushev Alexey
Введите номер телефона: 89209165221
Введите дату рождения (через пробел): 08 02 2001
Введите ФИО: Pura Alexey
Введите номер телефона: 89995353535
Введите дату рождения (через пробел): 04 08 2005
Введите ФИО: Pervushkin Pavel
Введите номер телефона: 89999231234
Введите дату рождения (через пробел): 13 05 2005
Введите ФИО: Kosmachev Dmitriy
Введите номер телефона: 89201923456
Введите дату рождения (через пробел): 14 08 2005
Введите ФИО: Vasilev Yaroslav
Введите номер телефона: 89201234212
Введите дату рождения (через пробел): 04 06 2006
Введите ФИО: Gladkov Nikita
Введите номер телефона: 89999999999
Введите дату рождения (через пробел): 09 09 2009
Введите ФИО: Nikitin Andrey
Введите номер телефона: 82912034112
Введите дату рождения (через пробел): 05 05 2000
Введите ФИО: Knutov Andrew
Введите номер телефона: 89201922233
Введите дату рождения (через пробел): 17 06 2001
Введите номер телефона для поиска: 89209165221
ФИО: Rushev Alexey
Номер телефона: 89209165221
Дата рождения: 8-2-2001
```

Рисунок 5 – Выполнение кода задания 7 темы

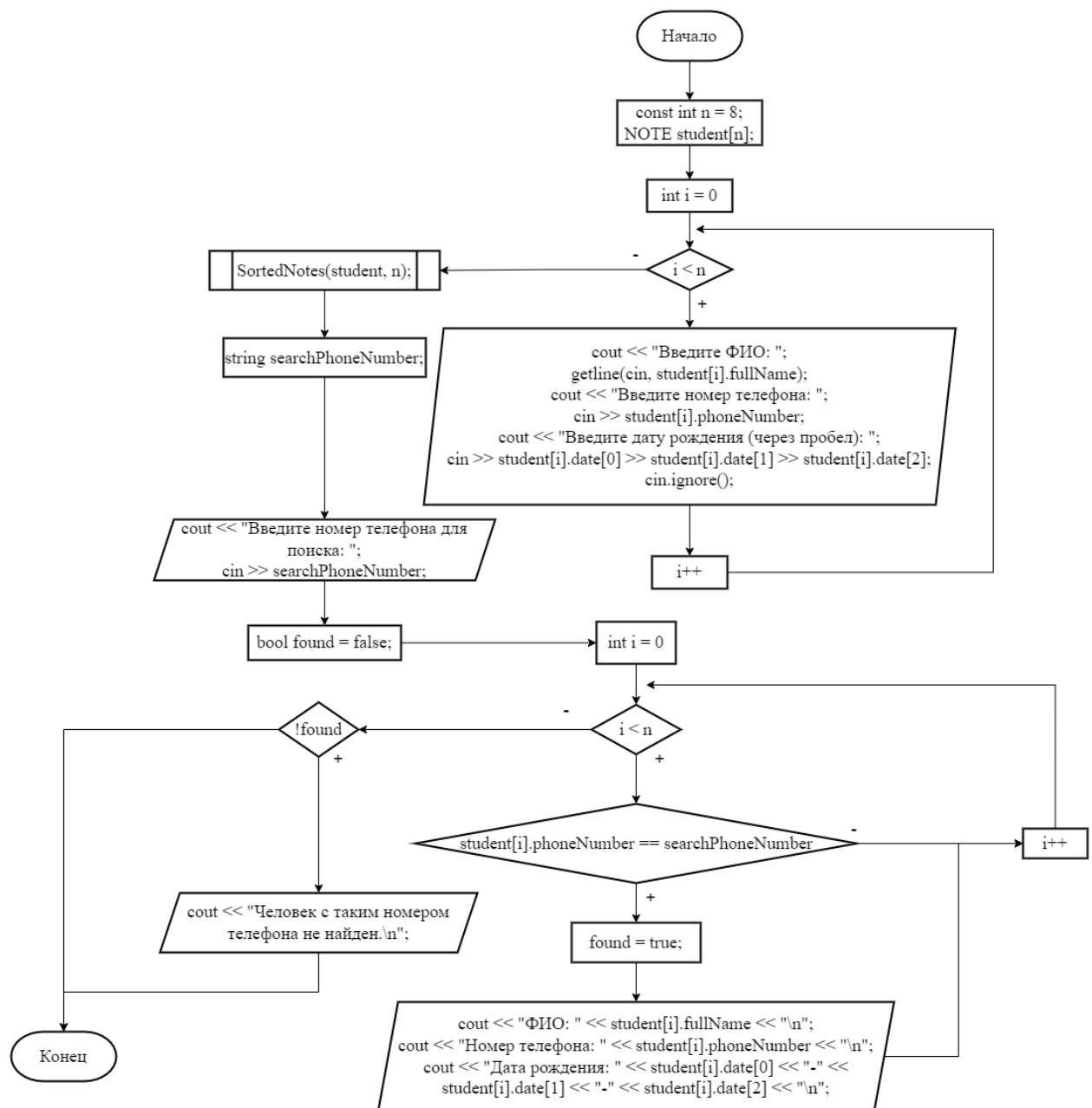


Рисунок 6 – Блок-схема задания 7 темы

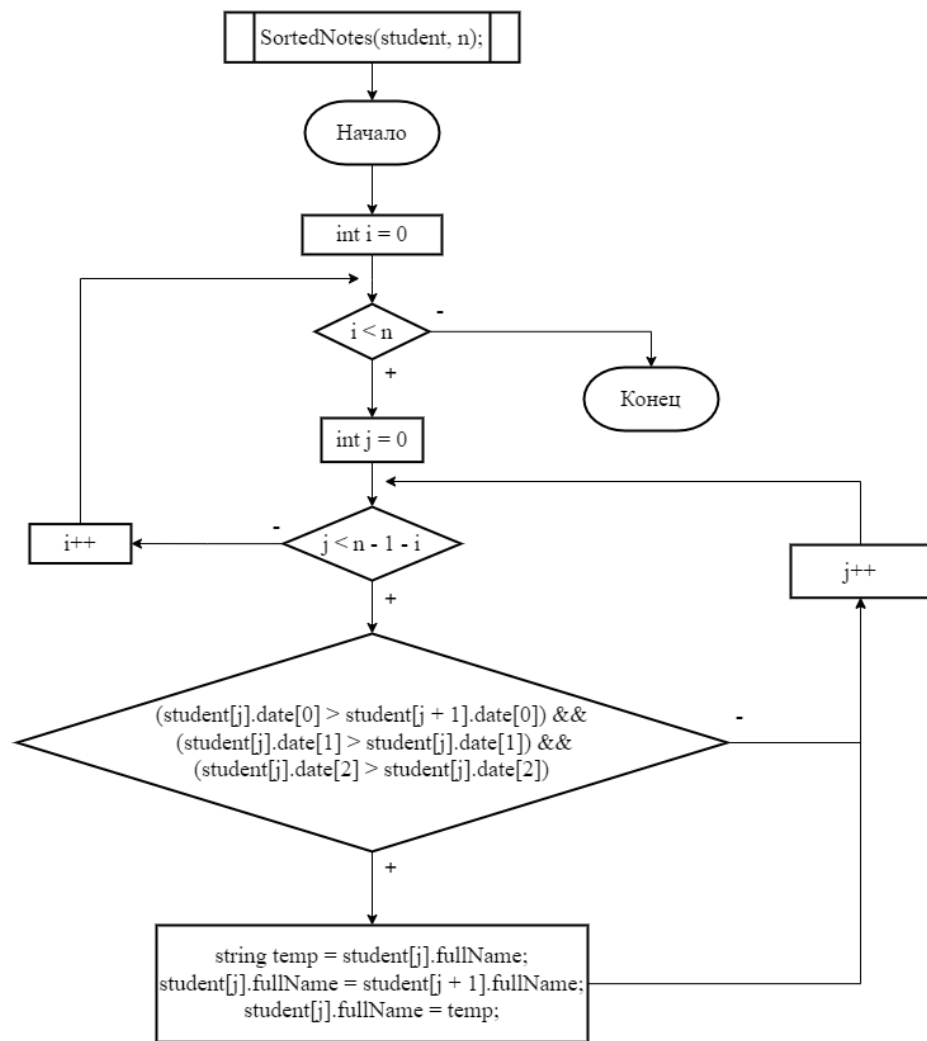


Рисунок 7 – Блок-схема функции сортировки задания 7 темы

8 ФУНКЦИИ

Если в рамках решения некоторой задачи необходимо несколько раз выполнить определённую подзадачу, то эту подзадачу рекомендуется оформить в виде функции.

Код, оформленный в виде функции, должен полностью решать одну определённую задачу и удовлетворять определению и свойствам алгоритма.

После написания кода функции к ней можно обратиться в любом месте программы, что позволяет избежать дублирования кода.

Итак, функция – это именованная последовательность описаний и операторов, направленных на решение некой задачи.

`main()` – это главная функция. С этой функции начинается выполнение программы на языке C++. Функции, созданные пользователем (программистом), должны быть предварительно объявлены и определены. Таким образом, при работе с функцией выделяют три этапа: объявление, определение и вызов.

Объявление функции – это её заголовок. Определение функции – описание действий, которые выполняет функция. Вызов функции – это выполнение описанных действий с конкретными параметрами.

Объявление функции (или заголовок функции) содержит имя функции, тип возвращаемого значения, список формальных параметров в круглых скобках. Наличие круглых скобок обязательно, даже если список параметров отсутствует. Объявление функции заканчивается точкой с запятой.

Объявление функции должно быть выполнено до её вызова. Синтаксис объявления функции:

[<класс памяти>] тип возвращаемого значения <имя функции>
([список формальных параметров]);

Определение может быть совмещено с объявлением функции

или вынесено в отдельный модуль. Определение функции содержит заголовок функции и тело функции. Телом функции называют последовательность действий, заключённых в фигурные скобки.

Класс памяти – необязательная часть заголовка функции, задающая область видимости функции. Если класс памяти не указать при объявлении функции, то будет использован класс памяти по умолчанию – `extern`. Такие функции доступны для вызова во всех модулях программы.

Если функция объявлена с классом памяти `static`, то она доступна для вызова только в том модуле, в котором определена.

Тип возвращаемого значения – это тип значения, которое будет возвращено из функции в точку её вызова. Тип возвращаемого значения в заголовке функции должен присутствовать обязательно. В случае если возвращать значение из функции не нужно, то в качестве типа возвращаемого значения указывают `void`. Имя функции может быть любым идентификатором, отвечающим требованиям именованию в C++. Наличие круглых скобок после имени функции обязательно. Скобки могут быть пустыми или содержать список формальных параметров. Если параметров несколько, их перечисляют через запятую. При объявлении функции можно указать только тип параметров; при определении функции нужно указывать и тип, и имя параметров. Параметры, указываемые при объявлении и определении функции, называют формальными. При вызове функции указывают фактические параметры, которые будут подставлены на место формальных.

Пример объявления функции, не принимающей параметры и не возвращающей значение:

```
void f1 ();
```

Пример объявления функции, принимающей два вещественных параметра типа `float` и возвращающей в точку вызова значение типа `float`:

```
float f (float, float);
```

В приведённом примере объявления имена параметров отсутствуют. При определении функции имена должны быть указаны, например, так:

```
float f (float c, float d){ ... };
```

Объявленная функция должна возвращать значение типа float. Для возвращения значения из функции используют оператор return. После ключевого слова return нужно указать возвращаемое значение соответствующего типа.

Пример:

```
float f (float c, float d){ return c+d;};
```

Для вызова функции, исполнения её кода с заданными при необходимости параметрами нужно указать имя функции и в круглых скобках – список фактических параметров. Тип фактических параметров обязательно должен соответствовать типу формальных параметров.

Вызов функций f1 и f:

```
f();
```

```
f1 (1.2,3.4);
```

```
или float S= f1 (1.2,3.4);
```

При втором варианте вызова функции f1 возвращаемое значение будет сохранено в переменную S. В первом случае возвращаемое значение будет потеряно.

Если возвращаемое значение не планируется использовать в дальнейших вычислениях и достаточно его вывода на экран, возможен такой подход:

```
cout << f1 (1.2,3.4);
```

Передача параметров в функцию

Существует три способа передачи параметров в функцию:

по значению;

указателю;

ссылке.

При передаче параметров в функцию по значению в области памяти функции создаётся локальная копия фактических параметров. Изменение этих значений внутри функции не влечёт за собой изменения внешних, переданных

значений. Изменения затрагивают только локальную копию, недоступную после выхода из функции.

Пример:

```
int f(int b)
{
    b++;
    return ++b;
}

int main()
{
    int b = 3; int c = f1(b);
}
```

В результате выполнения этого участка кода в переменную *c* будет сохранено значение, возвращаемое функцией *f*. Переменная *c* примет значение, равное 5, значение переменной *b*, объявленной в функции *main*, останется неизменным.

При передаче значений по указателю и ссылке в область памяти функции будут переданы адреса соответствующих переменных. Обращение к параметрам в теле функции означает переход по адресу. Соответственно, изменение значений в теле функции приведёт к изменению значений, переданных в функцию. Для работы со значением через указатель его необходимо разыменовывать.

Пример:

```
int f2(int *b)
{
    (*b)++;
    return ++(*b);
}

int main()
{
    int b = 3; int* pb = &b;
    int b1 = f2(pb);
}
```

В приведённом примере значение переданной в функцию переменной будет увеличено на 2. Таким образом, значения переменных b и b1 будут равны 5.

При передаче параметра по ссылке нет необходимости в разыменовывании, такой подход уменьшает вероятность возникновения ошибок.

Пример:

```
int f3(int& c)
{
    (++c)++;
    return ++c;
}

int main()
{ int c = 3; int c1 = f3(c);
}
```

Опишем и вызовем функцию, принимающую параметры, переданные всеми тремя способами:

```
int f4(int a, int* b, int& c)
{a++; (*b)++; c++;
return a + *b + c;
}
```

Здесь int a – передан по ссылке, int* b – по указателю, int& c – по значению.

Объявим соответствующие переменные и вызовем функцию:

```
int main()
{int a = 3; int b = 3; int* pb = &b; int c = 3;
int S = f4(a,pb,c);
}
```

В теле функции значения всех переданных параметров увеличиваются на единицу. Таким образом, возвращаемое значение равно 12. При этом

значения внешних переменных *b* и *c* увеличатся на единицу, значение переменной *a* останется неизменным.

Передача массива в функцию реализуется через адрес его первого элемента. Таким образом, для организации передачи массива в функцию необходимо передать два параметра: адрес его начала и количество элементов. Так как массив передают в функцию через указатель, то любые действия, произведённые с элементами массива в теле функции, выполняются с элементами внешнего массива и возвращать массив из функции нет необходимости.

При работе с многомерными массивами необходимо передавать в функцию информацию о всех размерностях. Внутри функции многомерный массив рассматривается как одномерный.

Пример передачи массива в функцию:

```
void Sort(int* mas, int N) // объявление функции
```

```
{...}; // тело функции
```

```
...
```

```
int const n = 10;
```

```
int mas[n] = {1,-2,3,-4,5,-1,2,-3,4,-5};
```

```
Sort(mas, n, ); // вызов функции
```

Параметры со значениями по умолчанию

Параметры по умолчанию – это параметры, значение которых задано предварительно при объявлении функции. При вызове функции такие параметры можно не указывать. В этом случае будет использовано значение по умолчанию.

Параметры со значением по умолчанию обязательно должны находиться в конце списка формальных параметров. Если таких параметров несколько и один из них при вызове опущен, то и все следующие за ним параметры должны быть опущены. Таким образом, все эти параметры примут значение по умолчанию.

Пример:

```
void f (int a, int b=1, int c=0 ); // объявление ф-ции
...
f(1); // параметры b и c принимают значение по умолчанию
f(1,2); /* параметр b принимает значение «2», параметр c
принимает значение по умолчанию.*/
```

Практическая часть

Задание: сложение двух целочисленных переменных. Функция получает два целочисленных параметра и возвращает их сумму.

Листинг приложения:

```
void task8_1(int a, int b) {
    cout << "Сумма чисел " << a << " и " << b << " = " << a + b << endl;
}
int main()
{
    setlocale(LC_ALL, "rus");
    task8_1(2, 3);
}
```

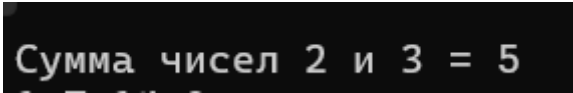
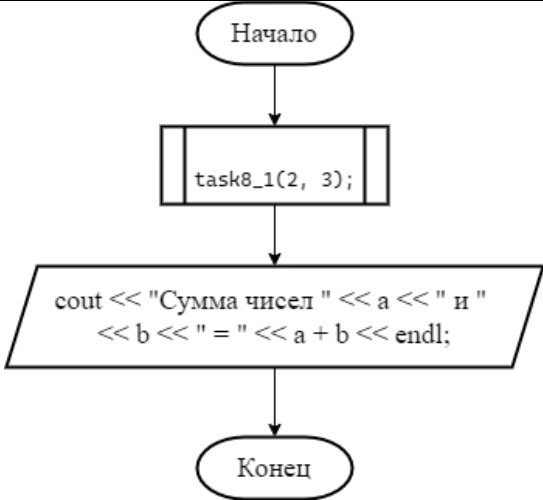
Результат выполнения кода:	Блок-схема
	 <pre> graph TD Start([Начало]) --> Call[task8_1(2, 3);] Call --> Print[/cout << "Сумма чисел " << a << " и " << b << " = " << a + b << endl;/] Print --> End([Конец]) </pre>

Таблица 6 – Выполнение кода и блок-схема задания 8 темы

9 ЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Под структурой данных понимают множество данных и множество связей между ними.

Динамические структуры данных – это структуры, в которых количество элементов и связи между этими элементами могут изменяться во время выполнения программы. По типу организации связей между элементами структуры могут быть линейными и нелинейными. Динамические структуры данных используют в ситуациях, когда заранее неизвестен объём данных, с которыми предстоит работать, и предполагается изменение (увеличение или уменьшение) их количества во время исполнения программы.

Связь элементов динамической структуры реализуют через указатели.

Списки, очереди, стеки, деки, деревья – примеры динамических структур данных.

Динамическую структуру данных, элементы которой связаны между собой определённым образом и не обязательно расположены в памяти подряд, называют списком.

Линейные списки могут быть однонаправленными и двунаправленными; и те и другие могут быть кольцевыми (закольцованными, замкнутыми).

Список называют кольцевым, если его последний элемент указывает на первый. Если кольцевой (или закольцованный) список – двунаправленный, то каждый элемент указывает на следующий и предыдущий; соответственно, последний элемент настроен на первый и предпоследний, а первый элемент указывает на второй и последний. Для организации динамической структуры данных каждый элемент списка должен содержать два вида полей: информационные поля и поля-указатели, служащие для связи между элементами. Количество информационных полей и их типы определяют исходя из условий конкретной решаемой задачи. Поля-указатели должны иметь тип данных – указатель на элемент списка. Количество полей-указателей

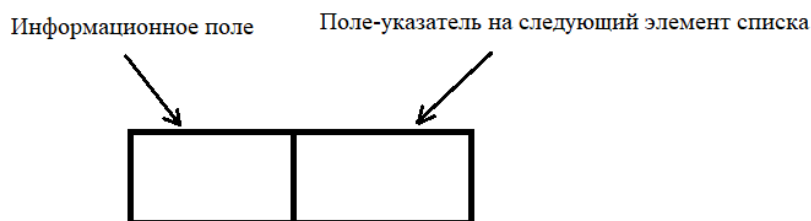
определяется видом списка: один указатель – для однонаправленных и два – для двунаправленных.

Элемент списка объединяет в себе разнотипные поля, поэтому в языке программирования C++ его описывают типом данных «структура», при этом под словом «структура» понимают тип данных, под словосочетанием «структура данных» – подход к организации данных.

Рассмотрим пример описания элемента динамического списка. Опишем структуру с двумя полями: информационным и полем-указателем на следующий элемент. Информационное поле будет целочисленным:

```
struct elem  
{int inf; elem * p;};
```

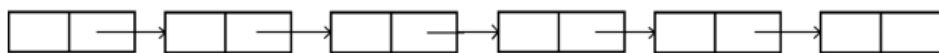
Поле p – это указатель на элемент (где элемент – это описываемая структура).



Создадим и проинициализируем элемент описанного выше типа:

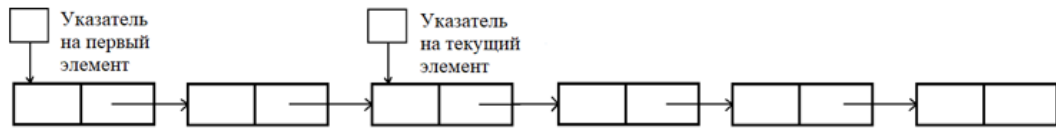
```
elem E1 = { 2, NULL };
```

Схематично показана структура данных «линейный список», который можно организовать, используя в качестве элементов описанный тип elem.

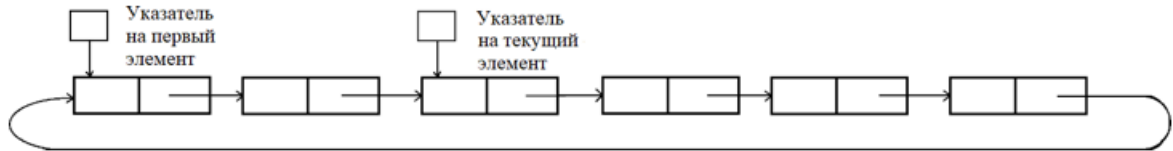


понятно, что для обращения к элементам такой структуры необходим как минимум один указатель на её начало для обращения к элементам.

Поэтому работа по созданию динамического списка начинается с объявления указателя на элемент списка. Для дальнейшей корректной работы может понадобиться указатель на текущий элемент. Результат объявления настройки таких указателей показан ниже.



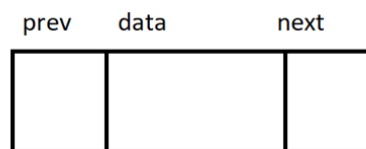
Если список кольцевой, то последний элемент необходимо настроить на первый.



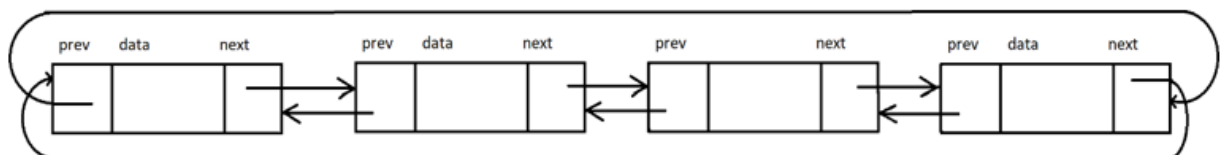
Если список двунаправленный, то каждый его элемент должен содержать два поля-указателя, настроенные на следующий и на предыдущий элементы:

```
struct elem
{
    elem* prev; int data; elem* next;
};
```

Схематично элемент типа struct elem представлен ниже.



Ниже наглядно показан кольцевой двунаправленный список. Для первого элемента такого списка предыдущим элементом является последний элемент, следующим элементом для последнего – первый.



Для работы с динамическими структурами данных необходимо определить следующие операции:

- создание первого элемента;
- добавление элемента в начало или конец списка;
- поиск элемента по ключу;

- вставка элемента по ключу;
- упорядочивание по ключу;
- удаление элемента из начала или конца списка;
- удаление элемента по ключу.

При добавлении и исключении элементов из списка необходимо проверять список на пустоту следующим образом:

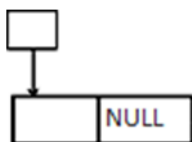
$head \neq NULL$, где $head$ – указатель на первый элемент списка (иначе называемый головой, или вершиной, списка).

Понятно, что алгоритм добавления элемента в список будет зависеть от того, первый это элемент списка или нет. То же касается и удаления. Предварительно нужно убедиться в том, есть ли элементы в списке.

Добавление элемента в однонаправленный линейный список

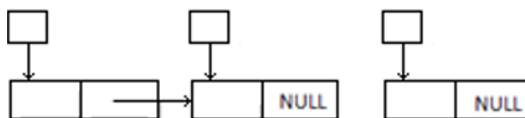
Добавление первого элемента:

- захватите память под элемент;
- заполните соответствующим образом информационные поля, поле-указатель настройте на $NULL$.

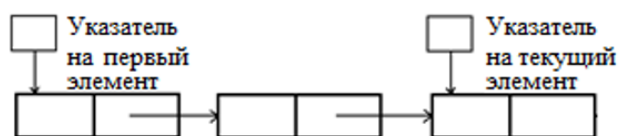


Добавление последующих элементов:

- захватите память под элемент (рис. 14);



- настройте соответствующим образом поля-указатели данного и предыдущего элементов списка; настройка полей-указателей будет зависеть от того, куда будет добавлен новый элемент, в голову или хвост списка.



Вставка элемента по ключу

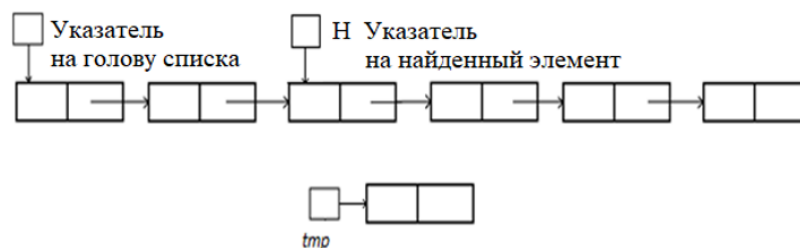
Перед тем как вставить элемент, следует реализовать поиск по ключу нужного элемента.

Например, необходимо вставить новый элемент за элементом с искомым значением. Для этого нужно настроить вспомогательный указатель на голову списка и перемещать его в цикле на следующий элемент до тех пор, пока не будет достигнуто искомое значение либо конец списка.

Для того чтобы найти искомое значение, нужно сравнить ключ с информационным полем элемента списка, на который настроен вспомогательный указатель:

`if (H->inf==k) H=H->p; // фрагмент реализации поиска, где H – указатель на найденный элемент, k – ключ.`

Результат поиска элемента схематично представлен ниже.



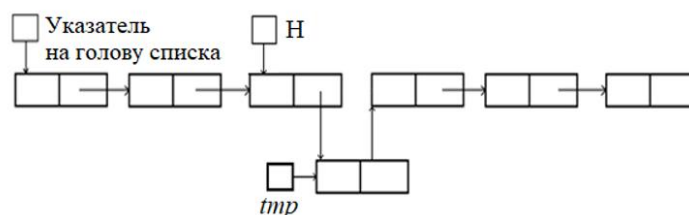
Далее необходимо настроить поле-указатель нового элемента списка на элемент, на который указывает элемент с совпавшим ключом:

`tmp->p=H->p;`

Поле-указатель найденного элемента следует настроить на новый элемент:

`H->p=tmp;`

Результат такой перенастройки представлен ниже.



Если необходимо вставить новый элемент перед элементом с заданным ключом, то проверяют на совпадение с ключом элемент, следующий за элементом, на который настроен вспомогательный указатель.

Отдельно рассмотрим ситуацию, когда нужно вставить элемент перед первым. Далее представлена следующая ситуация: новый элемент уже создан, но ещё не вставлен в список.



В этом случае поле-указатель нового элемента настраивают на первый элемент, после чего перенастраивают указатель на голову списка на вновь добавленный элемент. Либо поле-указатель первого элемента настраивают на новый в зависимости от организации направления списка, таким образом, если на схеме стрелки будут направлены в другую сторону, это означает, что поля-указатели настроены на предыдущий элемент, а не на следующий. Ниже схематично представлен результат добавления элемента в голову списка.



Исключение элемента из списка

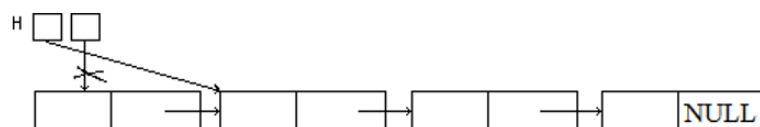
Прежде чем освободить память, занимаемую элементом, необходимо выполнить перенастройку указателей.

Алгоритм удаления первого элемента:

- настройте вспомогательный указатель на голову;



- перенастройте указатель на голову на следующий элемент;



- освободите память, занимаемую первым элементом.

Удаление элемента по ключу

Для реализации этой задачи необходимо два вспомогательных указателя – указатель на удаляемый элемент и указатель на предыдущий элемент.

Поле-указатель элемента, предшествующего удаляемому, настраивают на элемент, следующий за удаляемым, после чего освобождают память, занимаемую удаляемым из списка элементом.

Линейный список, организованный по принципу «первый вошёл, первый вышел» (FIFO – first in, first out), называют очередью.

Для наглядности целесообразно представить очередь в бытовом понимании, например очередь в магазин: кто первый в очередь встал, тот первым товар и получит. В случае динамического списка элемент, который был добавлен в такой список первым, будет извлечён из очереди первым.

Для реализации такого подхода необходимо объявить указатель на голову (вершину) списка и указатель на хвост (последний элемент) очереди для возможности добавления элементов в конец очереди. Эти два указателя можно объединить в одной переменной – структуре. Эта структура будет не того же типа, что и элементы списка; это другой пользовательский тип данных, который должен быть описан отдельно после описания структуры «элемент списка». Например, так:

```
struct Q
{
    elem* head; elem* tail;
};
```

Если планируется работа не более чем с одним списком, то поля head и tail можно объявить отдельными переменными вне структуры. Для добавления элемента в очередь потребуется дополнительный указатель:

```
elem* tmp;
tmp = new elem;
```

Алгоритм добавления элемента в очередь:

– захватите память под новый элемент списка:

```
elem* tmp;
tmp = new elem;
```

- заполните его поля:

```
tmp->inf = a;
```

- если это первый элемент списка (т. е. указатель на голову ни на что не указывает, `head == NULL`), то настройте его на новый элемент:

```
head = tmp;
```

Этот же элемент будет последним в силу единственности, т. е. указатель на хвост нужно также настроить на него;

- если элемент не первый, то указатель на голову не трогают.

Перенастройте указатель на хвост на вновь захваченный элемент:

```
tail = tmp;
```

- предварительно настройте на него же поле-указатель предыдущего элемента очереди:

```
tail->p = tmp;
```

Линейный список, организованный по принципу «последний вошёл, первый вышел» (LIFO – last in, first out), называют стеком.

Алгоритм добавления элемента в стек:

- захватите память под элемент и заполните его информационное поле:

```
elem* tmp;
```

```
tmp = new elem; tmp->inf = a;
```

- дальнейшая настройка зависит от того, первый это элемент или в стеке уже есть значения:

```
(head == NULL);
```

- если элемент первый, указатель на вершину стека настройте на этот элемент:

```
head = tmp; tmp->p = NULL;
```

- если в стеке уже есть значения, перенастройте указатели таким образом, чтобы не потерять связь между элементами и при этом новый элемент стал вершиной стека:

```
tmp->p = head; head = tmp;
```

При написании функций работы с динамическими списками параметры-указатели передают через ссылки; если указатель передать по значению, то изменённое значение не сохранится после выхода из функции.

Практическая часть

Задание: элементы целочисленного массива запишите в очередь. Напишите функцию извлечения элементов из очереди до тех пор, пока первый элемент очереди не станет чётным.

Листинг приложения:

```
struct elem1 {
    int a;
    elem1* next;
};
struct Q {
    elem1* h;
    elem1* t;
};
void add(Q& q, int k) {
    elem1* tmp = new elem1{ k, NULL };
    if (q.h == NULL) {
        q.h = tmp;
        q.t = tmp;
    }
    else {
        q.t->next = tmp;
        q.t = tmp;
    }
}
void izvl(elem1*& h) {
    while (h->a % 2 != 0) {
        h = h->next;
    }
}
void print(elem1*& h) {
    elem1* tmp = h;
    while (tmp != NULL) {
        cout << tmp->a << " ";
        tmp = tmp->next;
    }
}
void task9_1() {
    Q q = { NULL, NULL };
    int const n = 4;
    int m[n];
    for (int i = 0; i < n; i++) {
        m[i] = rand() % 20;
        add(q, m[i]);
        cout << m[i] << " ";
    }
    izvl(q.h);
    cout << endl;
```

```

        print(q.h);
    }
    int main()
    {
        setlocale(LC_ALL, "rus");
        task9_1();
    }

```

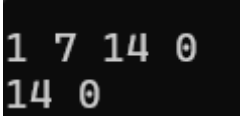
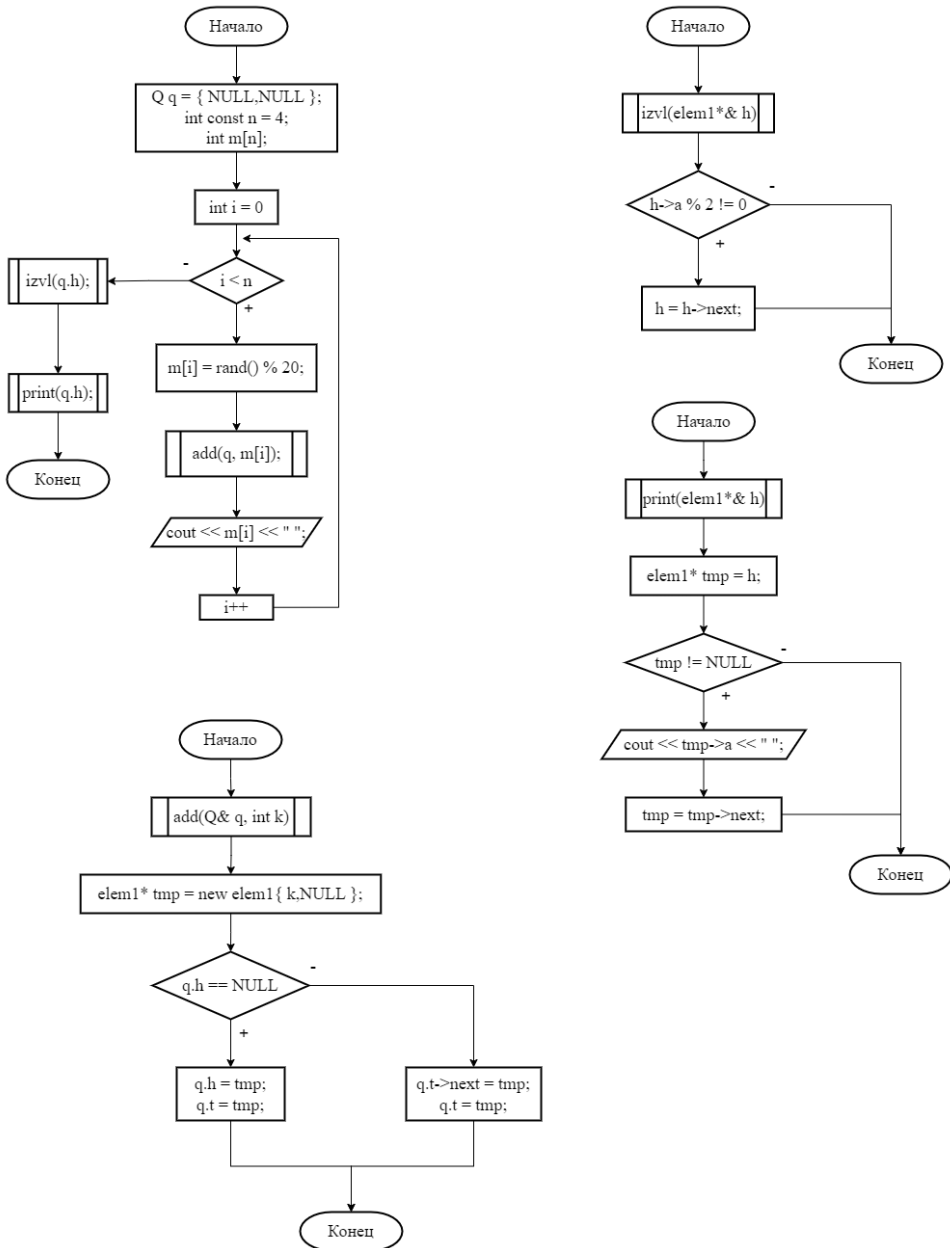
Результат выполнения кода:	Блок-схема
	 <pre> graph TD subgraph Flowchart1 [Flowchart 1: Initialization] Start1([Начало]) --> Init["Q q = { NULL, NULL }; int const n = 4; int m[n];"] Init --> i0["int i = 0"] i0 --> Loop1{ } Loop1 -- "+" --> Print1["print(q.h);"] Print1 --> End1([Конец]) Loop1 -- "-" --> Rand["m[i] = rand() % 20;"] Rand --> Add["add(q, m[i]);"] Add --> Cout1["cout << m[i] << \" \";"] Cout1 --> iinc["i++;"] iinc --> Loop1 end subgraph Flowchart2 [Flowchart 2: Traversal] Start2([Начало]) --> Print2["print(elem1 * h);"] Print2 --> Assign2["elem1 * tmp = h;"] Assign2 --> Loop2{ } Loop2 -- "+" --> Cout2["cout << tmp->a << \" \";"] Cout2 --> Next2["tmp = tmp->next;"] Next2 --> Loop2 Loop2 -- "-" --> End2([Конец]) end subgraph Flowchart3 [Flowchart 3: Insertion] Start3([Начало]) --> AddNode["add(Q & q, int k);"] AddNode --> NewNode["elem1 * tmp = new elem1 { k, NULL };"] NewNode --> NullCheck{q.h == NULL} NullCheck -- "+" --> Assign3["q.h = tmp; q.t = tmp;"] NullCheck -- "-" --> NextAssign["q.t->next = tmp; q.t = tmp;"] Assign3 --> End3([Конец]) NextAssign --> End3 end </pre>

Таблица 7 – Выполнение кода и блок-схема задания 9 темы

10 ВВЕДЕНИЕ В ЯЗЫК ПРОГРАММИРОВАНИЯ PYTHON

Как известно, переменная – это ячейка в памяти компьютера, имеющая имя и хранящая некоторое значение определенного типа (например, строка, целое число, вещественное число), тип значения переменной в языке программирования Python определяется автоматически. Имя переменной может состоять из заглавных и строчных латинских букв, цифр и знака подчеркивания, других символов в имени переменной быть не может. Имя переменной не может начинаться с цифры и не может содержать пробелы, знаки препинаний и символы арифметических операций. Рекомендуется давать переменным имена «со смыслом».

Для того чтобы переменная хранила значение, это значение нужно в нее «положить», т. е. сохранить. Для этого служит оператор присваивания =

Примеры: `a = 5`; `b = 3.14`; `name = 'N. A.'`

Для обозначения комментариев в языке программирования

Python используют символ #.

Ввод значения с клавиатуры выполняется при помощи команды `input()`. При ее выполнении программа будет ожидать, пока пользователь введет значение и нажмет на клавиатуре кнопку Enter, введенное значение будет сохранено в переменную. Например, так:

```
name = input() # ввод строки
```

Для того чтобы ввести число, необходимо преобразовать строку, введенную командой `input()`, в нужный формат при помощи функций `int()` или `float()`

Например, строка `c1 = int(input())` реализует ввод целого числа и его запись в переменную `c1`. Строка кода `c2 = float(input())` реализует ввод вещественного числа и его запись в переменную `c2`.

Внутри круглых скобок функции ввода `input()` можно написать параметр-подсказку для пользователя. Например, так:

```
a = int(input('Введите целое число: '))
```

Эта подсказка будет отображена на экране перед вводом значения переменной.

Для вывода на экран какой-либо информации используется команда (функция) `print()`.

После ключевого слова `print` в круглых скобках указывают то, что необходимо вывести: это может быть число, строка (в кавычках), переменная, арифметическое выражение и т. д. Функция `print` может содержать несколько аргументов, перечисляемых через запятую. Например: `print(5, 5+8), print('Answer:', (5+8)*14)`

Для того чтобы вывести на экран одинарные кавычки, их необходимо заключить в двойные кавычки, и наоборот, чтобы вывести двойные кавычки, их необходимо заключить в одинарные.

Например, строка кода `print("Hello!")` выведет на экран строку "Hello!"

Если внутри круглых скобок указать несколько аргументов через запятую, то они будут выведены через пробел. Если этот пробел не нужен, его можно убрать, указав в качестве последнего аргумента `sep=""`. Например, так:

```
print('My', 'Little', 'Friend', sep="")
```

В результате работы этой строки кода на экране появится строка: MyLittleFriend

Каждая новая команда `print()` выводит на экран текст с новой строки. Отменить переход на новую строку можно, указав в качестве последнего параметра `end=""`.

Таким образом, две строки кода:

```
print('My ', end="") print('friend')
```

выведут на экран одну строку: My friend

Отступы (сдвиги относительно левой границы) используются в языке программирования Python для разделения блоков программы между собой. Для того чтобы операторы были отнесены к одному блоку, необходимо разместить их на одном уровне.

Арифметические операции

В табл. 8 приведены основные арифметические операции языка программирования Python.

Знаки операций могут быть использованы в простых или составных выражениях. Для повышения приоритета операции используют круглые скобки.

Таблица 8

Основные арифметические операции языка Python

Знак операции	Операция	Пример использования
+	Сложение	$a + b$
-	Вычитание	$a - b$
*	Умножение	$a * b$
/	Деление	a / b
**	Возведение в степень	$a ** b$
//	Деление нацело	$a // b$
%	Остаток от деления	$a \% b$

В табл. 9 приведены составные операторы присваивания. Использование таких операторов позволяет сократить запись.

Таблица 10

Составные операторы присваивания

Знак операции	Пример использования	Альтернатива
/=	$x /= 3$	$x = x / 3$
*=	$x *= 3$	$x = x * 3$
+=	$x += 3$	$x = x + 3$
-=	$x -= 3$	$x = x - 3$
%=	$x \% = 3$	$x = x \% 3$
//=	$x //= 3$	$x = x // 3$
**=	$x ** = 3$	$x = x ** 3$

Практическая часть

Задание: с клавиатуры введите площадь основания пирамиды и её высоту (по одному значению в строке). Выведите на экран объем пирамиды.

Листинг приложения:

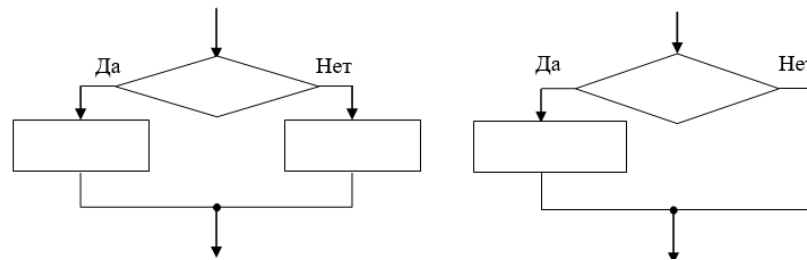
```
S = int(input("Введите площадь основания пирамиды: "))
h = int(input("Введите высоту пирамиды: "))
V = ((1/3)*(S * h))
print("Объем пирамиды = ", V)
```

Результат выполнения кода:	Блок-схема
<pre>Введите площадь основания пирамиды: 23 Введите высоту пирамиды: 33 Объем пирамиды = 253.0</pre>	<pre>graph TD Start([Начало]) --> Input[/S = int(input("Введите площадь основания пирамиды: ")) h = int(input("Введите высоту пирамиды: "))/] Input --> Process[V = ((1/3)*(S * h))] Process --> Output[/print("Объем пирамиды = ", V)/] Output --> End([Конец])</pre>

Таблица 11 – Выполнение задания и блок-схема задания 10 темы

11 ОПЕРАТОР ВЕТВЛЕНИЯ В ЯЗЫКЕ PYTHON

Условный оператор (оператор ветвления) используется при необходимости выполнить определённые команды в зависимости от значения некоторого условия. Оператор ветвления имеет две формы – полную и сокращённую. Структурные схемы конструкций представлены ниже.



В любом случае для реализации ветвления необходимо указать ключевое слово `if`, после которого следует логическое выражение, завершающееся двоеточием. Например:

```
if cash >= 150:
```

Далее следует блок инструкций, который будет выполнен, если логическое выражение истинно. Сокращённая конструкция на этом заканчивается. В случае полной конструкции далее следует указать ключевое слово `else`, после которого нужно поставить двоеточие, и блок инструкций, который будет выполнен, если условие не выполняется.

Ключевое слово `else` должно быть расположено строго под ключевым словом `if`, команды, входящие в блок инструкций, – с отступом относительно расположения ключевых слов `if` и `else`. Рекомендуется отступ в четыре пробела, но не менее одного.

Синтаксис и пример использования оператора «условие» в языке программирования Python представлены ниже.

Синтаксис:

```
if <условие>:
```

```
<блок инструкций 1>
```

```
else:
```

<блок инструкций 2> Пример:

```
if d==1: print("true")
```

```
else:
```

```
print ("false")
```

В примере, приведённом выше, на экран будет выведено слово true при равенстве значения переменной d единице, в противном случае, если значение переменной не равно единице, на экран будет выведено слово false.

Логическое выражение, используемое в условной конструкции, может быть простым или составным. В простых условиях используются следующие знаки операций: ==, !=, >, >=, <=.

Например: $a > 0$ или $x \neq y * z$.

Составным называют условие, состоящее из двух или более простых условий, соединенных логическими операциями: and, or, not. Например, так: $a < b$ and $b < 10$ или так: $x \geq 0$ or $x < 10$, not ($a == 0$).

Операция or (логическое ИЛИ) требует выполнения хотя бы одного из условий. Конструкция <условие 1> or <условие 2> or

<условие 3> будет принимать значение «ложь», только если все простые условия ложны. Операция and (логическое И) возвращает значение «истина» при выполнении всех условий: <условие 1> and

<условие 2> and <условие 3> будет принимать истинное значение, только если все простые условия истинны. Причем если условие 1 ложно, то условие 2 проверяться уже не будет.

Операция not <условие 1> (логическое НЕ) будет принимать ложное значение («ложь»), если условие 1 истинно, и наоборот. Например, следующие два условия равносильны: $A > B$ и not ($A \leq B$).

Логические операции можно комбинировать между собой, но важно помнить о приоритете: первой выполняется операция not, затем and, последней выполняется операция or.

При необходимости изменить порядок выполнения операций используют круглые скобки.

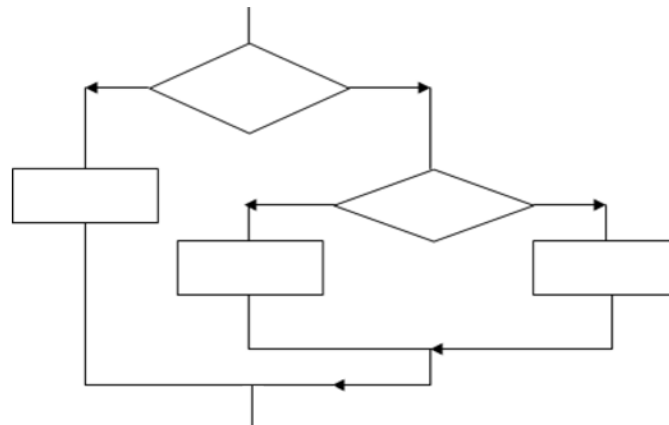
Пример: $a > 0$ or $a < 100$ and $a \% 2 == 0$

Равносильно: $(a > 0$ or $a < 100)$ and $a \% 2 == 0$

Вложенные условные конструкции

Внутри условных конструкций можно использовать любые инструкции языка Python, в том числе и условную конструкцию. В этом случае говорят о вложенном ветвлении. При таком подходе вложенные блоки имеют больший размер отступа.

Ниже представлен пример структурной схемы вложенного ветвления.



Множественные вложенные условные операторы можно сократить, используя каскадную конструкцию `if-elif-else`. Ключевое слово `elif` заменяет конструкцию `else: if`. При этом ключевые слова располагаются строго друг под другом, а блоки инструкций расположены с одинаковым отступом. Пример использования такой конструкции:

```
d=int(input()) if d==0:
print("zero") elif d>0:
print ('plus') else:
print('minus')
```

В языке Python существуют встроенные функции `max()` и `min()`. Функция `max(<последовательность>)` возвращает максимальное значение из последовательности, функция `min(<последовательность>)` – минимальное значение из последовательности. Например:

```
m = max(a, b, c)
n = min(a+b, c // 3)
```

Практическая часть

Задание: пользователь вводит число – год. Необходимо определить, является ли год високосным, вывести на экран соответствующую надпись, а также количество дней в году. Високосным год является, если его номер кратен 4, но не кратен 100, а также если он кратен 400.

Листинг приложения:

```
date = int(input("Введите год: "))
if(date % 4 == 0 and date % 100 != 0 or date % 400 == 0):
    year = True
    days = 366
else:
    year = False
    days = 365
if year:
    print(f"{year} год - является високосным. В нем {days} дней")
else:
    print(f"{year} год - не является високосным. В нем {days} дней")
```

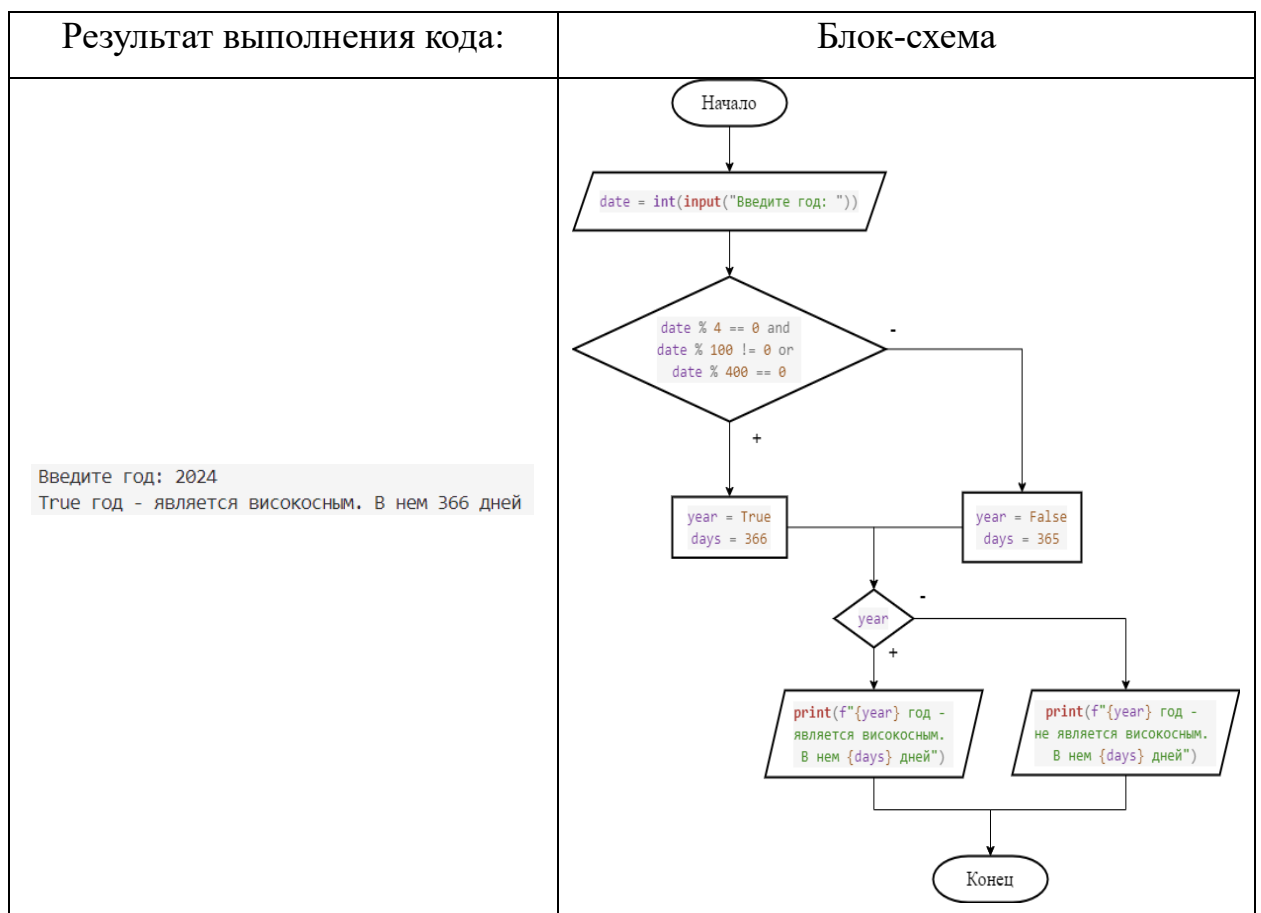


Таблица 12 – Выполнение кода и блок-схема задания 11 темы

12 ОПЕРАТОРЫ ОРГАНИЗАЦИИ ЦИКЛОВ ЯЗЫКА PYTHON

Цикл – конструкция языка программирования, использующаяся при необходимости выполнить определённую последовательность действий несколько раз. В языке Python существуют два оператора, позволяющих реализовать цикл.

Оператор while

При применении цикла while последовательность команд – тело цикла – выполняется до тех пор, пока условие истинно. Принцип работы оператора следующий: проверяется условие, если оно истинно, то выполняются команды из тела цикла, затем условие проверяется снова. Действия повторяются до тех пор, пока условие не станет ложным.

Синтаксис оператора while:

```
while <логическое выражение>:
```

```
<операторы тела цикла>
```

Команды тела цикла записываются с одинаковым отступом относительно ключевого слова while. Ниже приведён пример организации цикла при помощи оператора while:

```
n=int(input()) i=0
```

```
while i<n: print(i) i+=1
```

Способы прерывания циклов

Оператор break используют при необходимости досрочно завершить цикл. Когда break будет выполнен в теле цикла, последний будет досрочно прерван, управление будет передано оператору, следующему за оператором цикла.

Оператор continue используют при необходимости в одном из случаев пропустить часть тела цикла и перейти к следующей итерации. Когда оператор

continue выполнен, текущая итерация будет прервана и снова проверено условие.

Ниже приведены примеры использования операторов прерывания цикла:

```
x=0
while x<10: x+=1
    if x==5: break
x+=1
print('x=',x) print('Out of loop')
```

В результате работы этого фрагмента кода на экран будут выведены следующие строки:

```
x= 2
x= 4
Out of loop x=0
while x<10: x+=1
    if x==5: continue
x+=1
print('x=',x) print('Out of loop')
```

Результат на экране будет таким:

```
x=0
while x<10: x+=1
    if x==5: continue
x+=1
print('x=',x) print('Out of loop')
```

В языке Python в циклах так же, как и в условном операторе, можно использовать ключевое слово else. В этом случае блок внутри else выполняется один раз, как только условие цикла станет ложным. Обычно такой подход используют для проверки способа выхода из цикла. Блок внутри else выполнится только в том случае, если выход из цикла произошел без помощи break:


```
x=0
while x<5: print(x) x+=1
else:
    print('The End')
```

Результат работы:

```
0
1
2
3
4
```

The End

Модернизируем тело цикла, используя условие и оператор прерывания:

```
x=0
while x<5: if x==3:
    break print(x) x+=1
else:
    print('The End')
```

Результат работы будет таким:

```
0
1
2
```

Оператор for

Оператор for реализует цикл с параметром. Используют этот оператор в ситуациях, когда количество повторений цикла заранее известно.

Синтаксис оператора for:

```
for <переменная-параметр> in <множество значений перемен- ной-
параметра>:
    <тело цикла>
```

Переменная-параметр принимает начальное значение из множества значений, далее выполняется первая итерация, после чего значение

переменной-параметра автоматически изменяется на следующее значение и снова выполняется тело цикла, затем значение переменной- параметра будет вновь изменено и так далее до достижения переменной-параметром конечного значения.

Рассмотрим пример. Следующий фрагмент кода выведет на экран строку «я – студент» три раза:

```
for i in 1,2,3:  
    print ("я – студент")
```

Оператор цикла for так же, как и оператор while, может прерываться при помощи break и continue, содержать часть else, выполняющуюся только если не был выполнен досрочный выход из цикла.

Значения переменной-параметра удобно задавать с помощью функции range, позволяющей генерировать ряд чисел в рамках заданного диапазона.

Функция range может быть вызвана тремя способами: с одним, двумя или тремя аргументами.

В первом случае функция генерирует ряд чисел от нуля до конечного числа, не включая его: range (10) – будет сгенерирован ряд чисел от 0 до 9.

Во втором случае функция принимает два аргумента – начальное и конечное значения; будет сгенерирован ряд чисел от начального числа до конечного, не включая конечное, например: range (1,10) – будет сгенерирован ряд чисел от 1 до 9.

Практическая часть

Задание: введите с клавиатуры последовательность целых чисел, завершающуюся числом 0. Вычислите, сколько было введено положительных чисел и сколько отрицательных, выведите два этих числа через пробел.

Листинг приложения:

```
pol = 0  
otr = 0  
while True:
```

```

number = int(input("Введите число: "))
if number == 0:
    break
elif number > 0:
    pol += 1
else:
    otr += 1
print(f"Положительных чисел: {pol}, отрицательных чисел: {otr}")

```

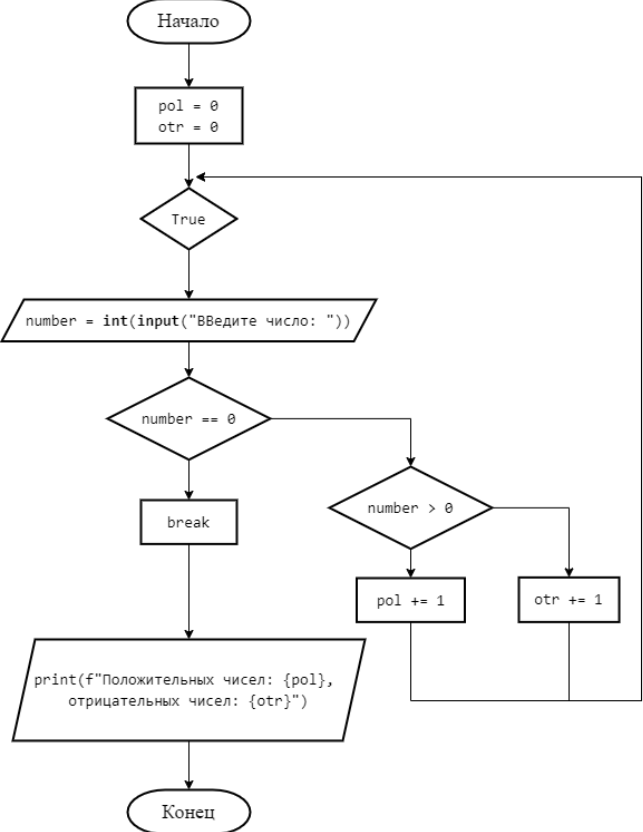
Результат выполнения кода:	Блок-схема
<div data-bbox="245 871 796 1128" style="background-color: #f0f0f0; padding: 10px;"> Введите число: 123 Введите число: 43 Введите число: -5 Введите число: -3 Введите число: 2 Введите число: -4 Введите число: 3 Введите число: 0 Положительных чисел: 4, отрицательных чисел: 3 </div>	 <pre> graph TD Start([Начало]) --> Init[pol = 0
otr = 0] Init --> LoopStart{True} LoopStart --> Input[/number = int(input("Введите число: "))/] Input --> IsZero{number == 0} IsZero --> Break[break] Break --> Output[/print(f"Положительных чисел: {pol},
отрицательных чисел: {otr}")/] Output --> End([Конец]) IsZero --> IsPos{number > 0} IsPos --> PolInc[pol += 1] IsPos --> OtrInc[otr += 1] PolInc --> Join(()) OtrInc --> Join Join --> LoopStart </pre>

Таблица 13 – Выполнение кода и блок-схема задания 12 темы

13 СТРОКИ

Строки – объекты, состоящие из последовательности символов. Для того чтобы присвоить строку переменной, ее надо заключить в одинарные кавычки: `s = 'I like study'`. Строку можно считать со стандартного ввода функцией `input()`. Любой другой объект в Python можно привести к строке. Для этого нужно использовать функцию `str()`, передав ей в качестве параметра объект, переводимый в строку. И наоборот, при необходимости строку можно привести к другому классу, например к `int`:

```
s='1234' # строка
```

```
c=int(s) # c – целое число, хранящее значение 1234 a=567 # a – целое  
число, хранящее значение 567 b=str(a) # b – строка
```

Операции для работы со строками

+ – оператор сложения строк (конкатенации). Возвращает строку, состоящую из двух строк-параметров:

```
s='abcd' c='efg' d=s+c
```

В переменной `d` будет содержаться строка `abcd`.

* – оператор умножения строки на число. Создает несколько копий строки-параметра:

```
s='abc' d=s*3
```

Переменная `d` примет значение `abcsabcsabc`.

Срезы – извлечение из строки подстроки; это может быть один символ или некоторый фрагмент строки. Существуют три типа срезов:

Срез с одним параметром. `s[i]` – это срез, состоящий из одного символа, который имеет индекс `i`. Нумерация символов строки начинается с нуля. Например, если `s == 'Python'`, то `s[0] == 'P'`, `s[1] == 'y'`, `s[2] == 't'` и т. д. Каждый объект, который получается в результате среза `s[i]`, – это тоже строка типа `str`. Если указать отрицательное значение индекса, то номер будет отсчитываться с конца, начиная с номера `-1`, т. е. `-1` это номер последнего символа.

Срез с двумя параметрами. `s[a:b]` возвращает подстроку из `b`-а символов, начиная с символа с индексом `a`, до символа с индексом `b`, не включая его. Например, `s[1:4] == 'yth'`. То же самое получится, если написать `s[-4:-1]`. Можно использовать как положительные, так и отрицательные индексы в одном срезе, например: `s[1:-1]` – это строка без первого и последнего символов (срез начинается с символа с индексом 1 и заканчивается индексом 1, не включая его).

Если опустить второй параметр, но поставить двоеточие, то срез берется до конца строки. Например, чтобы удалить из строки первый символ (его индекс равен 0), можно взять срез `s[1:]`. Если опустить первый параметр, то можно взять срез от начала строки, т. е. удалить из строки последний символ можно при помощи среза `s[:-1]`. Срез `s[:]` совпадает с самой строкой `s`.

Срез с тремя параметрами. `s[a:b:step]` – третий параметр задает шаг, таким образом, в срез будут включены символы с индексами `a`, `a + step`, `a + 2step` и т. д. до `b`, не включая `b`. При задании значения третьего параметра, равного 2, в срез попадет каждый второй символ, а если взять шаг -1, то символы будут идти в обратном порядке. Например, можно перевернуть строку срезом `s[::-1]`.

Функция `len()` возвращает количество элементов строки, переданной в функцию в качестве аргумента, т. е. длину этой строки. Например: `len('abcdef') == 6`

Функция `len()` может использоваться не только для строк, но и для любых последовательностей.

Оператор принадлежности `in` возвращает `True`, если подстрока входит в строку, и `False`, если подстрока не входит в строку.

Оператор `not in` возвращает `True`, если подстрока не входит в строку, и `False` в противном случае.

Для фрагмента кода:

```
s='qwerty' print('wer' in s) print('rwe' in s) print('ty' in s) print('try' in s)
```

Результат будет таким:

True False True False

Рассмотрим некоторые методы работы со строками.

Метод – это функция, применяемая к объекту (в данном случае – к строке). Для вызова метода следует указать после имени объекта через точку имя метода с круглыми скобками, содержащими необходимые параметры:

<имя объекта>.<имя метода>(<параметры>)

Метод count() возвращает количество вхождений одной строки в другую строку. Например, s.count(t) возвращает число вхождений строки t внутри строки s. При этом подсчитываются только непересекающиеся вхождения (т. е. вхождения, которые не перекрывают друг друга).

Метод find() находит в строке данную подстроку, которая передается в качестве параметра, и возвращает индекс первого вхождения искомой подстроки. Если же подстрока не найдена, то метод возвращает значение -1.

Метод rfind() возвращает индекс последнего вхождения данной подстроки в строке.

Метод replace(old, new[, k]) заменяет старую подстроку на новую. У метода есть третий необязательный параметр: с его помощью можно указать количество вхождений (первых), которые нужно заменить. Если его не писать, заменятся все вхождения.

Применим к строке S="баобаб" описанные методы. Результат применения каждого метода приведён в комментариях:

```
print(S.count('ба')) // на экране – 2
print(S.count('б')) ## на экране – 3
print(S.find('ба')) ## на экране – 0
print(S.rfind('ба')) ## на экране – 3
print(S.replace('ба', '*')) ## на экране – *о*б
print(S.replace('ба', '*',1)) ## на экране – *обаб
```

Практическая часть

Задание: с клавиатуры вводится строка. Замените в этой строке все появления букв a, b на букву G.

Листинг приложения:

```
str1 = input("Введите строку: ")
str2 = str1.replace('a', 'G')
str2 = str2.replace('b', 'G')
print("Измененная строка:", str2)
```

Результат выполнения кода:	Блок-схема
<div>Введите строку: asdsadASSvdagdawe Измененная строка: GsdsGdASSvdGgdGwe</div>	<pre>graph TD; Start([Начало]) --> Input[/str1 = input("Введите строку: ")/]; Input --> Process1[str2 = str1.replace('a', 'G')]; Process1 --> Process2[str2 = str2.replace('b', 'G')]; Process2 --> Output[/print("Измененная строка:", str2)/]; Output --> End([Конец]);</pre>

Таблица 14 – Выполнение кода и блок-схема задания 13 темы

14 МАССИВЫ. СПИСКИ

Для создания массива в языке программирования Python необходимо указать ключевое слово `array` с пустыми круглыми скобками или со скобками, содержащими список значений.

Массив может содержать элементы только одного типа данных (например, только целые числа). Списки в Python имеют более широкий функционал, поэтому их используют чаще.

Список – именованный упорядоченный изменяемый набор объектов произвольных типов. Нумерация элементов начинается с 0.

Для создания списка существует несколько способов. Рассмотрим каждый из них.

Перечисление элементов списка в квадратных скобках. Например:

```
d=[1,2,3,4,-9,5] d1=[]
```

```
d2=[1,'w',123,'Python',3.14]
```

где `d` – список целых чисел, `d1` – пустой список, `d2` – список, содержащий разнотипные данные, в том числе и другой список.

Для того чтобы обратиться к определенному элементу списка, необходимо написать имя списка, а затем индекс требуемого элемента в квадратных скобках: например, запись `a[0]` соответствует нулевому элементу списка `a`.

На экран можно вывести как список целиком (при этом список будет заключён в квадратные скобки), так и его отдельные элементы:

```
print(d2) print(d2[0])
```

Приведение любого другого итерируемого объекта (например, строку) к списку при помощи функции `list()`. Например:

```
S="Python" List_S=list(S)
```

В списке `List_S` каждый элемент является строкой. Если вывести этот список на экран `print(List_S)`, результат будет таким:


```
['P', 'y', 't', 'h', 'o', 'n']
```

Заполнение пустого списка или изменение готового списка при помощи цикла. Например:

```
a=[0]*10 # создали список из 10 нулей  
for i in range(10):  
    a[i]=i # заполнили список значениями от 0 до 9
```

Генератор списков. Использование генератора списков очень похоже на цикл for, например:

```
a=[a for a in "Python"] print(a)
```

Список a принимает значение из шести строк, каждая из которых равна символу строки "Python". Вывод списка на экран даст следующий результат:

```
['P', 'y', 't', 'h', 'o', 'n']
```

При генерации элементов возможно задание повторения символа в рамках одного элемента, как показано в следующем примере:

```
a=[a*3 for a in "Python"]
```

Список a примет следующие значения:

```
['PPP', 'yyy', 'ttt', 'hhh', 'ooo', 'nnn']
```

Возможно наложение ограничений при генерации списка, например:

```
a=[a*2 for a in "Python" if a!="t"]
```

 Список a будет заполнен следующим образом:

```
['PP', 'yy', 'hh', 'oo', 'nn']
```

Операции для работы со списками

Со списками можно выполнять те же операции, что и со строками.

Оператор конкатенации + возвращает список, состоящий из элементов других списков-операндов. Например:

```
d=[1,2,3,4,-9,5] d1=[1,'w',123,'Python',[1,2,3],3.14]  
d2=d+d1 print(d2)
```

На экран будет выведено:

```
[1, 2, 3, 4, -9, 5, 1, 'w', 123, 'Python', [1, 2, 3], 3.14]
```

Оператор * умножения списка на число создает несколько дубликатов списка. Например:

```
d=[1,2,3,'w']
```

```
d2=d*3
```

Будет создан список d2:

```
[1, 2, 3, 'w', 1, 2, 3, 'w', 1, 2, 3, 'w']
```

Из списков можно извлекать срезы по тем же правилам, что и из строк:

```
d=['P','y','t','h','o','n'] d2=d[2:4]
```

Список d2 будет равен ['t', 'h']

```
print(d[2:-1:]) – на экран будет выведено ['t', 'h', 'o']
```

В случае среза `print(d[2::-1])` на экран будет выведено

```
['t', 'y', 'P'].
```

Методы для работы со списками

Для вызова метода необходимо указать имя этого метода после имени списка, разделив их точкой. После имени метода ставят круглые скобки, содержащие при необходимости список параметров:

`append(x)` – добавляет элемент x в конец списка

```
S=[1,2,3,4]
```

```
S.append(5)
```

Список S будет содержать следующие элементы: [1, 2, 3, 4, 5].

`extend(listname2)` – добавляет в конец списка все элементы списка

listname2

```
S=[1,2,3,4,5]
```

```
S1=[6,7,8,9,]
```

Список S будет содержать следующие элементы: [1, 2, 3, 4, 5, 6, 7, 8, 9].

`insert(i, x)` – вставляет значение x на i-ю позицию списка:

```
S=['P','y','t','h','o','n']
```

```
S.insert(7, '!')
```

Список примет значение ['P', 'y', 't', 'h', 'o', 'n', '!']. `remove(x)` – удаляет из списка первый элемент, равный x: `S.remove('!')`

Список примет первоначальное значение.

```
S=['a','b','a','b','a','b']
```

```
S.remove('a')
```

Первый элемент будет удалён из списка. Список примет значение ['b', 'a', 'b', 'a', 'b']

`pop(i)` – удаляет элемент списка с индексом *i*. Если индекс не задан, то удаляется последний элемент списка:

```
S=['P','y','t','h','o','n']
```

```
S.pop(0)
```

```
S.pop()
```

Список *S* примет значение ['y', 't', 'h', 'o']

`sort()` – сортирует список в порядке возрастания по умолчанию и в порядке убывания, если задать значение параметра `reverse = True`:

```
S=[1,5,0,4,-3,-2,2,-1,6,3]
```

```
S.sort()
```

Список *S* примет значение [-3, -2, -1, 0, 1, 2, 3, 4, 5, 6]

```
S.sort(reverse = True)
```

Значение списка *S* [6, 5, 4, 3, 2, 1, 0, -1, -2, -3] `count(x)` – возвращает количество элементов со значением *x* в

списке:

```
S=['a','a','b','a','b','a','b']
```

```
c=S.count('a') ## c=4
```

`index(x, a, b)` – возвращает индекс первого элемента со значением *x* в списке в диапазоне индексов от *a* до *b*, не включая последний (диапазон можно не указывать, тогда поиск будет осуществлён по всему списку):

```
S=['a','a','b','a','b','a','b']
```

```
c=S.index('a',2,6) ##c=3 d=S.index('a') ## d=0
```

`copy()` – копирует список и возвращает копию в точку вызова:

S1=*S*.copy() ## Строка *S1* равна строке *S* `clear()` – очищает список:

```
S.clear()
```

`str.split(separator[, maxsplit])` – разбивает строку `str` на части по разделителю `separator` и возвращает эти части в виде списка. Если разделитель не задан, то в качестве разделителя используется пробел и символ новой строки. Необязательный параметр `maxsplit` определяет максимальное количество частей, на которые можно разбить строку. Например:

```
str='aaaaabaaabaab' str1='ab ab abbb' s=str.split('a')
```

```
##s=["", "", "", "", "b", "", "", "b", "", "b"]
```

```
s1=str.split('a',2)
```

```
## ["", "", 'aaabaaabaab'] s2=str1.split()
```

```
##['ab', 'ab', 'abbb']
```

Функция `map(function, iterable_object)` применяет функцию `function` к каждому элементу итерируемого объекта, переданного вторым аргументом, т. е., по сути, заменяет цикл `for`, перебирающий элементы. `map()` может принимать несколько итерируемых объектов в качестве аргументов функции, отправляя в функцию по одному элементу каждого итерируемого объекта за раз. Функция `map()` возвращает не список, а объект `map` (итератор), поэтому для приведения его к списку нужно использовать `list()`. Например:

```
S=[1,2,3,4,5]
```

```
S1=list(map(float, S ))
```

Список `S1` примет значения `[1.0, 2.0, 3.0, 4.0, 5.0]`

Практическая часть

Задание: дан список натуральных чисел. Удалите из него первое четное число, имеющее нечетный индекс. Выведите измененный список.

Листинг приложения:

```
array = [3, 5, 7, 8, 12, 9, 10, 11]
index = 0
for i in range(len(array)):
    if array[i] % 2 == 0 and i % 2 != 0:
        index = i
```

```

        break
    if index != 0:
        del array[index]
print(array)

```

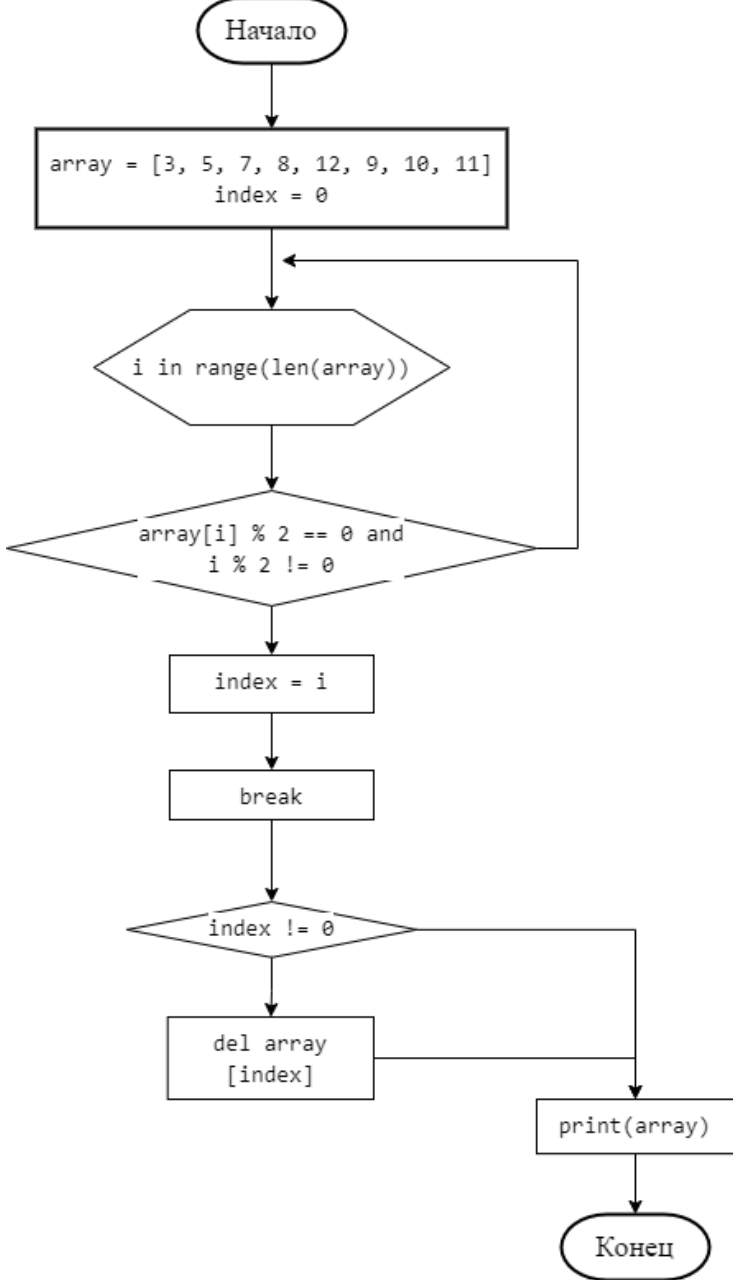
Результат выполнения кода:	Блок-схема
<div data-bbox="240 1019 715 1072" style="background-color: black; color: white; padding: 5px; text-align: center;"> [3, 5, 7, 12, 9, 10, 11] </div>	 <pre> graph TD Start([Начало]) --> Init["array = [3, 5, 7, 8, 12, 9, 10, 11] index = 0"] Init --> Loop{i in range(len(array))} Loop --> Cond1{"array[i] % 2 == 0 and i % 2 != 0"} Cond1 -- True --> SetIndex["index = i"] SetIndex --> Break[break] Break --> Cond2{index != 0} Cond1 -- False --> Print["print(array)"] Cond2 -- True --> Del["del array [index]"] Del --> Print Cond2 -- False --> Print Print --> End([Конец]) </pre>

Таблица 15 – Выполнение кода и блок-схема задания 14 темы

15 ФАЙЛЫ

Для организации работы с файлом необходимо открыть его с помощью функции `open(имя файла)`, указав в круглых скобках имя открываемого файла, например:

```
f = open(«filename.txt»)
```

В качестве второго параметра можно указать режим открытия файла (табл. 16).

Таблица 16

Режим открытия файла

Режим	Обозначение
<i>r</i>	Открытие на чтение (является значением по умолчанию)
<i>w</i>	Открытие на запись, содержимое файла удаляется, если файла не существует, создается новый
<i>x</i>	Открытие на запись, если файла не существует, иначе исключение
<i>a</i>	Открытие на дозапись, информация добавляется в конец файла
<i>b</i>	Открытие в двоичном режиме
<i>t</i>	Открытие в текстовом режиме (является значением по умолчанию)
<i>+</i>	Открытие на чтение и запись

Режимы могут быть объединены, например `'rb'` – чтение в двоичном режиме. По умолчанию режим равен `'rt'`.

Для того чтобы прочитать одну строку из файла, используют функцию `readline()`:

`a = f.readline()` – из файла `f` будет считана строка и сохранена в переменной `a`

`a = int(f.readline())` – считанная строка будет переведена в целое число.

При этом файл с программой и файл с данными должны лежать в одной папке.

`f.readlines()` – возвращает список строк, считанных из файла.

Для записи данных в файл необходимо открыть файл для записи и использовать метод `write()`, возвращающий число записанных символов. После окончания работы с файлом его обязательно нужно закрыть с помощью метода `close()`:

`f.close()`

Практическая часть

Задание: в файле электронной таблицы в каждой строке содержатся шесть неотрицательных целых чисел. Определите количество строк таблицы, для которых выполнены оба условия: – в строке только одно число повторяется трижды, остальные числа не повторяются; – утроенная сумма повторяющихся чисел строки не больше произведения неповторяющихся чисел.

Листинг приложения:

```
import math
with
open('C:\\Users\\Алексей\\Desktop\\Учеба\\github\\OsnovyProgramm\\SecondSemestr\\
Практика учебная\\часть 15\\table.txt', 'r') as file:
    count = 0
    for line in file:
        numbers = list(map(int, line.split()))
        unique_numbers = set(numbers)
        for num in unique_numbers:
            if numbers.count(num) == 3 and num * 3 <= math.prod([x for x in
numbers if x != num]):
                count += 1
                break
print(count)
```

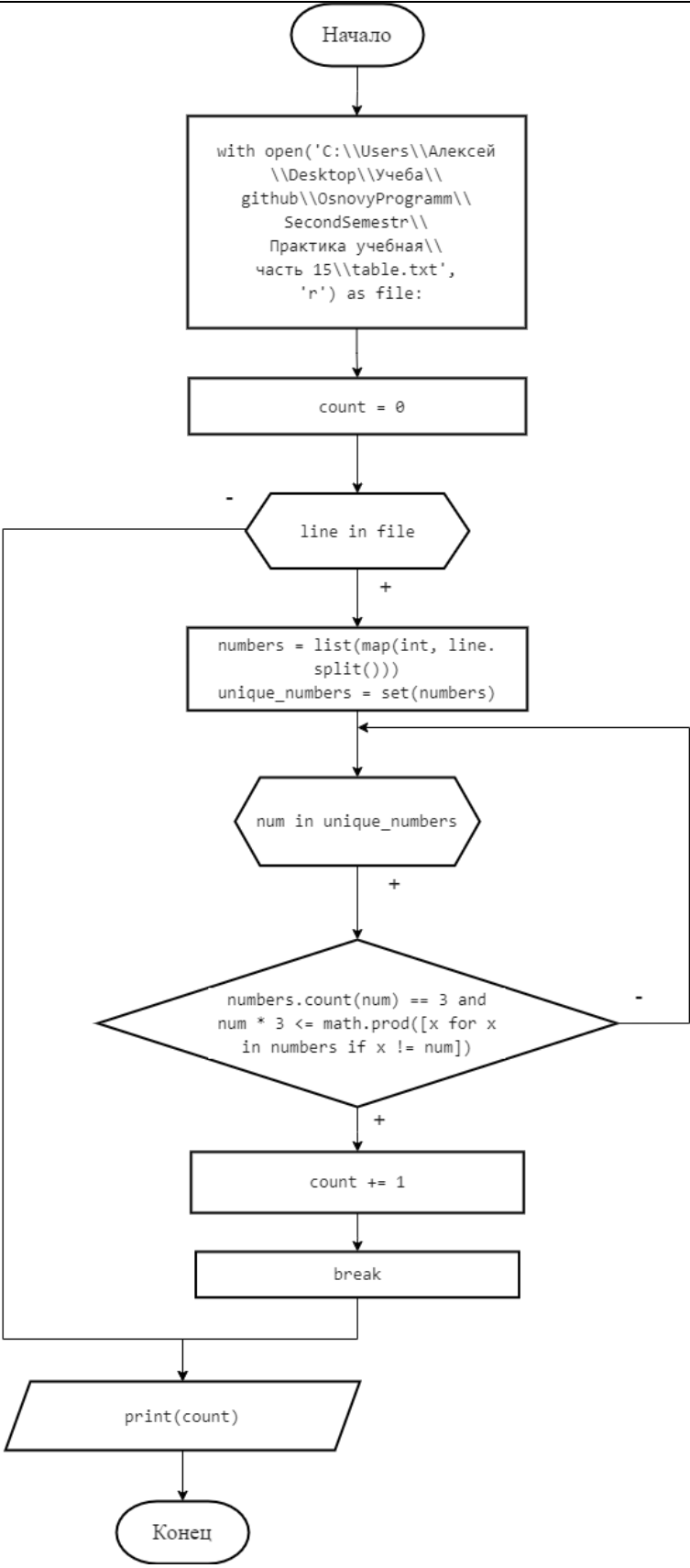
<p>Результат выполнения кода:</p>	<p>Блок-схема</p>
<div data-bbox="240 882 673 1090" data-label="Code-Block"> <pre>часть 15 > ≡ table.txt 1 1 2 3 4 5 6 2 2 3 4 5 6 6 3 3 3 3 7 8 9</pre> </div> <p data-bbox="339 1115 571 1153">Данные файла</p> <div data-bbox="414 1258 493 1314" data-label="Text"> <p>1</p> </div> <p data-bbox="316 1339 593 1377">Результат работы</p>	 <pre> graph TD Start([Начало]) --> OpenFile[with open('C:\\Users\\Алексей\\Desktop\\Учеба\\github\\OsnovyProgramm\\SecondSemestr\\Практика учебная\\часть 15\\table.txt', 'r') as file:] OpenFile --> Count0[count = 0] Count0 --> LineInFile{{line in file}} LineInFile -- + --> ProcessLine[numbers = list(map(int, line.split())) unique_numbers = set(numbers)] ProcessLine --> NumInSet{{num in unique_numbers}} NumInSet -- + --> Decision{numbers.count(num) == 3 and num * 3 <= math.prod([x for x in numbers if x != num])} Decision -- - --> LineInFile Decision -- + --> CountInc[count += 1] CountInc --> Break[break] Break --> PrintCount[/print(count)/] PrintCount --> End([Конец]) </pre>

Таблица 16 – Выполнение кода и блок-схема задания 15 темы

16 ПРОЦЕДУРЫ. ФУНКЦИИ

Подпрограмма – часть программы, имеющая имя и решающая свою отдельную задачу. Все подпрограммы располагаются в начале основной программы и могут быть вызваны внутри основной программы по имени. Подпрограммы используют, когда в разных частях основной программы необходимо выполнять одни и те же действия несколько раз. В таком случае повторяемые операторы (решающие отдельную задачу) оформляются в виде подпрограммы, к которой можно обращаться и вызывать ее выполнение из разных частей программы. Подпрограммы позволяют избежать дублирования кода.

Каждая подпрограмма должна решать только одну задачу (что-то вычислять, выводить какие-либо данные и т. д.). Подпрограммы разделяют на два вида: процедуры и функции. Подпрограммы-процедуры выполняют какие-то действия (например, выводят что-то на экран). Подпрограммы-функции возвращают значение, т. е. результатом их работы является значение (число, строка, список и т. д.). В языке программирования Python подпрограммы – процедуры и функции – описываются одинаково.

Функция – это подпрограмма, принимающая аргументы и возвращающая значение. Функции могут быть встроенными, а могут быть пользовательскими, описанными программистом. Пример вызова встроенной функции:

```
l=[1,2,3,4,5]
```

```
s=sum(l)
```

Переменная S принимает значение суммы элементов списка l.

Для того чтобы создать собственную функцию, её необходимо объявить и описать. Для этого используется конструкция `def`, после которой указывают имя функции, круглые скобки, содержащие при необходимости список аргументов (параметров), и двоеточие. Далее после отступа следует описание

тела функции. Для возвращения результата из функции необходимо использовать ключевое слово `return`, после которого указывают возвращаемое значение. `return` завершает выполнение функции.

Для вызова функции нужно написать ее имя в круглых скобках, содержащих при необходимости список передаваемых параметров.

Ниже приведено описание и вызов функции `Sum_ch`, принимающей на вход список и возвращающей сумму чётных элементов этого списка.

```
def Sum_ch(l):  
    S=0  
    for i in l: if i%2==0:  
        S+=i  
    return S  
l=[1,2,3,4,5]  
s=Sum_ch(l)  
print(s)
```

В данном случае значение переменной `s` будет равно 6.

Если подпрограмма не возвращает, её называют процедурой.

Пример описания и вызова процедуры приведён ниже.

```
def Err():  
    print("Ошибка! Повторите ввод")  
    n=-1  
    while n<0: n=int(input())  
    if n<0: Err()
```

Переменную, объявленную внутри функции, называют локальной. Использовать такие переменные можно только внутри функции.

Глобальные переменные объявляют вне функции. В функциях такие переменные доступны, но изменять значение такой переменной внутри функции нельзя. Например:

```
x=1  
def F():  
    x=2  
    x+=1  
    print("x =",x) ## x=3  
    x+=1  
F()
```

```
print("x =",x) ## x=2
```

Функцию, которая внутри своего же тела вызывает сама себя, называют рекурсивной.

Приведём пример вычисления факториала целого неотрицательного числа с помощью рекурсивной функции:

```
n=int(input()) def f(n):  
    if n >1:  
        return n*f(n-1) else:  
        return 1 ff= f(n)  
    print("f(n) =",ff)
```

По умолчанию глубина рекурсии в языке Python ограничена 1000 ВЫЗОВОВ.

Практическая часть

Задание: напишите функцию XOR_cipher, принимающую два аргумента: строку, которую нужно зашифровать, и ключ шифрования, и возвращающую строку, зашифрованную путем применения функции XOR (^) над символами строки с ключом. Напишите также функцию XOR_uncipher, которая по зашифрованной строке и ключу восстанавливает исходную строку.

Листинг приложения:

```
def XOR_cipher(text, key):  
    ciphered_text = ""  
    for char in text:  
        ciphered_text += chr(ord(char) ^ key)  
    return ciphered_text  
def XOR_uncipher(ciphered_text, key):  
    return XOR_cipher(ciphered_text, key) # Функция XOR является своим обратным  
действием  
# Пример использования  
text = "Привет, мир!"  
key = 42  
ciphered_text = XOR_cipher(text, key)  
print("Зашифрованный текст:", ciphered_text)  
unciphered_text = XOR_uncipher(ciphered_text, key)
```

```
print("Расшифрованный текст:", unciphered_text)
```

Результат выполнения кода:	Блок-схема
<div>Зашифрованный текст: eXВИП▲ ЖВЖ</div> <div>Расшифрованный текст: Привет, мир!</div>	<pre>graph TD Start([Начало]) --> Init["text = 'Привет, мир!' key = 42"] Init --> EncFunc["ciphered_text = XOR_cipher(text, key)"] EncFunc --> EncInit["ciphered_text = ''"] EncInit --> EncLoop{char in text} EncLoop -- "+" --> EncCalc["ciphered_text += chr(ord(char) ^ key)"] EncCalc --> EncLoop EncLoop -- "-" --> EncReturn["return ciphered_text"] EncReturn --> EncPrint[/print("Зашифрованный текст:", ciphered_text)/] EncPrint --> DecFunc["unciphered_text = XOR_uncipher(ciphered_text, key)"] DecFunc --> DecReturn["return XOR_cipher(ciphered_text, key)"] DecReturn --> DecInit["ciphered_text = ''"] DecInit --> DecLoop{char in text} DecLoop -- "+" --> DecCalc["ciphered_text += chr(ord(char) ^ key)"] DecCalc --> DecLoop DecLoop -- "-" --> DecReturn DecReturn --> DecPrint[/print("Расшифрованный текст:", unciphered_text)/] DecPrint --> End([Конец])</pre>

Таблица 17 – Выполнение кода и блок-схема задания 16 темы

ЗАКЛЮЧЕНИЕ

Учебная практика позволила ближе познакомиться с осваиваемой профессией, погрузиться в атмосферу командной работы. Был закреплён материал, изученный во время теоретической подготовки в течение учебного года, который применялся для решения одной задачи знания из нескольких дисциплин. Кроме того, некоторые навыки получилось усовершенствовать, а также закрепить и углубиться в теоретических знаниях. Решение большого количества типовых и исследовательских задач позволило наработать мощную базу, благодаря которой теперь легче войти в профессию.

Таким образом, своевременное прохождение учебной практики, решение достаточного количества практических задач позволило усовершенствовать профессиональные компетенции, сделать большой шаг в освоении профессии.

ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ

Что такое `map()`?

Встроенная в Python функция, которая используется для применения функции к каждому элементу итерируемого объекта (например, списка или словаря) и возврата нового итератора для получения результатов. Функция `map()` возвращает объект `map` (итератор), который можно использовать в других частях программы.

Синтаксис.

Синтаксис функции `map()` следующий:

```
map(function, iterable, ...)
```

Где:

- `function`: функция, которая будет применяться к каждому элементу итерируемого объекта.
- `iterable`: один или несколько итерируемых объектов (например, списки, кортежи и т.д.).

Функция `map()` возвращает итератор, поэтому для получения результата в виде списка нужно использовать функцию `list()`.

Пример использования.

Пример 1: Преобразование списка чисел в их квадраты

```
numbers = [1, 2, 3, 4, 5]
```

```
squared = map(lambda x: x**2, numbers)
```

```
print(list(squared)) # Выведет: [1, 4, 9, 16, 25]
```

Пример 2: Преобразование типов данных

```
strings = ["1", "2", "3", "4"]
```

```
numbers = map(int, strings)
```

```
print(list(numbers)) # Выведет: [1, 2, 3, 4]
```

Пример 3: Комбинирование нескольких итерируемых объектов

```
a = [1, 2, 3]
```

```
b = [4, 5, 6]
```

```
summed = map(lambda x, y: x + y, a, b)
```

```
print(list(summed)) # Выведет: [5, 7, 9]
```

Пример 4: Применение пользовательской функции

```
def increment(x):
```

```
    return x + 1
```

```
numbers = [1, 2, 3]
```

```
incremented = map(increment, numbers)
```

```
print(list(incremented)) # Выведет: [2, 3, 4]
```

Пример 5: Обработка строк в списке

```
fruits = ["apple", "banana", "cherry"]
```

```
upper_fruits = map(str.upper, fruits)
```

```
print(list(upper_fruits)) # Выведет: ['APPLE', 'BANANA', 'CHERRY']
```

Пример 6: Использование нескольких итерируемых объектов

```
names = ["Alice", "Bob", "Charlie"]
```

```
ages = [25, 30, 35]
```

```
def combine(name, age):
```

```
    return f'{name} is {age} years old.'
```

```
combined = map(combine, names, ages)
```

```
print(list(combined)) # Выведет: ['Alice is 25 years old.', 'Bob is 30 years old.', 'Charlie is 35 years old.']
```

В каждом из этих примеров ключевым моментом является применение функции `map()` для трансформации или комбинирования данных, что позволяет сделать код более кратким и выразительным по сравнению с традиционными циклами.

Где применяется.

Функция `map()` в Python широко применяется для выполнения одной и той же операции над каждым элементом итерируемого объекта (например, списка, кортежа и т.д.). Вот несколько распространенных случаев использования функции `map()`:

1. Преобразование списков и других итерируемых объектов:

Например, можно использовать `map()`, чтобы применить функцию ко всем элементам списка:

```
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x**2, numbers)
print(list(squared)) # Выведет: [1, 4, 9, 16, 25]
```

2. Преобразование типов данных:

Когда необходимо преобразовать все элементы списка из одного типа в другой:

```
strings = ["1", "2", "3", "4"]
numbers = map(int, strings)
print(list(numbers)) # Выведет: [1, 2, 3, 4]
```

3. Обработка данных из нескольких итерируемых объектов: `map()` также может принимать несколько итерируемых объектов:

```
a = [1, 2, 3]
b = [4, 5, 6]
summed = map(lambda x, y: x + y, a, b)
print(list(summed)) # Выведет: [5, 7, 9]
```

4. Применение пользовательских функций:

Можно определить свою функцию и использовать `map()`, чтобы применить ее к каждому элементу итерируемого объекта:

```
def increment(x):
    return x + 1
numbers = [1, 2, 3]
```



```
incremented = map(increment, numbers)
print(list(incremented)) # Выведет: [2, 3, 4]
```

5. Обработка строк:

Например, можно использовать `map()` для применения строковых методов ко всем элементам списка строк:

```
fruits = ["apple", "banana", "cherry"]
upper_fruits = map(str.upper, fruits)
print(list(upper_fruits)) # Выведет: ['APPLE', 'BANANA', 'CHERRY']
```

6. Комбинирование с другими функциями высшего порядка:

Можно комбинировать `map()` с другими функциями высшего порядка, такими как `filter()` и `reduce()` из модуля `functools`:

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x**2, numbers)
summed = reduce(lambda x, y: x + y, squared)
print(summed) # Выведет: 55
```

Функция `map()` возвращает итератор, а не список, так что для отображения результата обычно нужно преобразовать его в список с помощью функции `list()`.

Плюсы и минусы.

Плюсы:

1. Читабельность:

- Краткость кода: код, использующий `map()`, часто более компактный и выразительный, чем эквивалентный цикл `for`.
- Функциональный стиль: `map()` хорошо вписывается в функциональный стиль программирования, когда функции не имеют побочных эффектов и работают только с входными данными.

2. Производительность:

- Ленивое выполнение: `map()` возвращает итератор, так что элементы вычисляются по мере необходимости, что экономит память в случае обработки больших данных.

- Оптимизация: в некоторых случаях `map()` может работать быстрее, чем ручной цикл, за счет внутренней оптимизации.

3. Параллельное выполнение:

- Функции, используемые с `map()`, могут быть легко распараллелены с помощью библиотек, таких как `multiprocessing`, что может обеспечить значительное ускорение.

Минусы:

1. Ограниченный функционал:

- Одиночная операция: `map()` может применяться только для одной функции за раз. Для более сложных операций, таких как фильтрация или сложные вычисления, придется использовать дополнительные инструменты или более сложные конструкции.

- Чтение кода: новичкам в Python может быть сложнее понять код, использующий `map()`, по сравнению с аналогичным циклом `for`.

2. Неинтуитивные ошибки:

- Отсутствие побочных эффектов: функция, передаваемая в `map()`, не должна иметь побочных эффектов. Если она изменяет внешний контекст, это может привести к трудноуловимым ошибкам.

- Ленивые вычисления: возвращаемый итератор может вызвать неожиданное поведение, если вы ожидаете немедленного выполнения операции.

3. Необходимость поддержки функций:

- Совместно с `lambda` и `func`: в случае сложных операций, вам, возможно, потребуется определить несколько функций или использовать `lambda`, что может сделать код менее читаемым.