

Министерство транспорта Российской Федерации  
Федеральное агентство железнодорожного транспорта  
Омский государственный университет путей сообщения

---

## ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Учебное пособие

Омск 2022

УДК 004.42(075.8)  
ББК 32.973я73  
Д13

**Основы программирования на языке C++:** Учебное пособие / А. И. Давыдов, Е. С. Калинина, И. Л. Саля, С. А. Ступаков; Омский гос. ун-т путей сообщения. Омск, 2022. 86 с.

Рассмотрены особенности языка программирования C++, порядок разработки приложений в интегрированной среде *Microsoft Visual Studio* средствами *Visual C++*, программирование базовых структур алгоритмов и их реализация при решении вычислительных задач.

Структура учебного пособия представляет собой изложение теоретического и практического материала, в котором представлены основные подходы к созданию проектов в соответствии с особенностями языка C++ и программируемых алгоритмов.

Предназначено для студентов и аспирантов очной и заочной форм обучения всех специальностей (направлений подготовки), изучающих основы программирования на языке C++. Может быть использовано в качестве пособия для самостоятельной работы при изучении программирования для пользователей с начальным уровнем подготовки.

Библиогр.: 5 назв. Табл. 6. Рис. 50.

Рецензенты: доктор техн. наук, доцент С. А. Бессоненко;  
канд. техн. наук, доцент Е. Ю. Копытов;  
канд. техн. наук, доцент А. Г. Малютин.

ISBN 978-5-949-41295-4

## ОГЛАВЛЕНИЕ

Введение .....	7
1. Языки программирования .....	8
1.1. Общие сведения о языках программирования .....	8
1.2. Виды языков программирования и их особенности.....	8
1.3. Трансляция программного кода высокоуровневых ЯП .....	10
2. Среда разработки приложений Microsoft Visual Studio. Компилятор Visual C++.....	11
2.1. Общие сведения.....	11
2.2. Основные понятия C++ .....	11
2.3. Алфавит языка C++.....	11
2.4. Типы данных C++ .....	12
2.5. Синтаксис идентификаторов и ключевых слов в C++ .....	13
2.6. Особенности объявления констант и переменных. Комментарии .....	14
2.7. Встроенные функции .....	15
3. Запись арифметических выражений.....	17
3.1. Правила построения математических выражений.....	17
3.2. Оператор присвоения. Операции инкремента и декремента.....	18
3.3. Логические операторы.....	19
4. Среда разработки приложений Microsoft Visual Studio.....	20
4.1. Создание проекта C++ в среде Microsoft Visual Studio .....	20
4.2. Области окна проекта .....	23
5. Создание программного кода.....	24
5.1. Разделы окна программного кода.....	24
5.2. Структура программного кода.....	26
5.3. Влияние типа данных на результат вычислений .....	29
5.4. Компиляция программного кода. Запуск программы .....	30
6. Линейные вычислительные процессы .....	33
7. Разветвляющиеся вычислительные процессы.....	35
7.1. Оператор условного перехода if.....	35

7.1.1. Простая форма оператора if .....	35
7.1.2. Альтернативная форма оператора if .....	36
7.1.3. Сложная форма оператора if.....	37
7.2. Объединение условий с помощью логических операций.....	39
7.3. Оператор выбора switch case.....	42
8. Циклические вычислительные процессы .....	44
8.1. Понятие цикла .....	44
8.2. Арифметический цикл.....	45
8.2.1. Цикл со счетчиком .....	48
8.2.2. Вычисление максимума и минимума .....	50
8.2.3. Цикл с разветвлением.....	52
8.3. Вычисление сумм и произведений.....	54
8.4. Итерационные циклы.....	58
8.4.1. Оператор цикла while .....	58
8.4.2. Оператор цикла do...while .....	59
8.5. Вложенные циклы .....	60
9. Функции в C++ .....	62
9.1. Понятие функции .....	62
9.1.1. Создание функции.....	63
9.1.2. Определение функции и ее вызов .....	64
9.2. Области видимости переменных в программе.....	65
9.3. Возвращаемое значение.....	66
9.4. Аргументы функции .....	66
9.4.1. Фактические аргументы .....	67
9.4.2. Формальные аргументы .....	67
9.5. Алгоритм вызова функции.....	68
9.6. Примеры использования функций .....	69
10. Массивы.....	73
10.1. Понятие массива и его инициализация.....	73
10.2. Примеры решения задач по обработке массивов .....	74

10.2.1 Вставка и удаление элемента массива .....	77
10.2.2. Сортировка массива .....	81
10.3. Многомерные массивы .....	84
11. Контрольные вопросы.....	84
Библиографический список.....	85



## ВВЕДЕНИЕ

Язык программирования C++ является одним из наиболее популярных языков программирования. Область его применения обширна – от создания операционных систем, разнообразных прикладных программ, драйверов устройств до приложений для операционных систем, высокопроизводительных серверов, игр.

Существует множество реализаций языка C++ как бесплатных, так и коммерческих, которые используются в различных компьютерных платформах. Например: *GCC* (набор компиляторов), *Visual C++* (компилятор для приложений от *Microsoft*), *Intel C++ Compiler* (оптимизирующий компилятор от *Intel*), *Embarcadero (Borland) C++ Builder* (компилятор быстрой разработки приложений) и др.

Язык C++ является наследником языка программирования C, но не включает в себя все его конструкции.

Данное пособие ориентировано на последовательное изучение C++ – начиная от знакомства со средой разработки, типов данных, особенностей ввода-вывода данных до программирования алгоритмических структур. Подробно рассмотрены основные операции по созданию проектов в среде *Microsoft Visual Studio* и типовые примеры решения математических задач.

Приведенные в пособии примеры соответствуют принципу «от простого к сложному» и рассматриваются в соответствии со структурой языка и классификацией алгоритмов:

- алфавит;
- типы данных;
- переменные и константы;
- арифметические выражения;
- логические операции;
- входной и выходной потоки;
- линейные процессы;
- разветвляющиеся процессы;
- циклические процессы;
- пользовательские функции;
- одномерные массивы.

В библиографическом списке в конце пособия приведена литература для углубленного изучения материала.

# 1. ЯЗЫКИ ПРОГРАММИРОВАНИЯ

## 1.1. Общие сведения о языках программирования

*Язык программирования (ЯП)* – это искусственный язык, предназначенный для реализации алгоритмов.

Современные ЯП являются языками высокого уровня по отношению к уровню *машинных команд* (команды в двоичном коде) электронных вычислительных машин (ЭВМ). ЭВМ оперирует только такими командами, поэтому программу, написанную на языке высокого уровня, необходимо перевести (транслировать) на язык машинных команд. Эту операцию выполняют обслуживающие программы-*трансляторы* (*интерпретаторы* и *компиляторы*).

ЯП являются искусственными языками со строго определенным *алфавитом*, *синтаксисом* и *семантикой*:

- алфавит (символы);
- синтаксис (правила написания команд);
- семантика (правила построения программы).

С точки зрения определения даты рождения языка программирования для ЭВМ существует некоторая двойственность:

в первой половине XIX в. Ада Лавлейс описала вычислительную машину и ввела основополагающие понятия управления ею;

первый язык программирования в современном представлении зародился во время Второй мировой войны на вычислительной машине Z4 немецкого изобретателя Конрада Цузе.

## 1.2. Виды языков программирования и их особенности

Языки программирования можно разделить на языки низкого, высокого и сверхвысокого уровня:

ЯП низкого уровня – это средство записи инструкций для компьютера на аппаратном языке, т. е. в машинных кодах (в виде последовательности нулей и единиц). Языки низкого уровня строго ориентированы на определенный тип компьютерного оборудования;

ЯП высокого уровня – это языки, ориентированные не на систему команд того или иного процессора, а на систему операторов (команд), характерных для записи определенного класса алгоритмов;



ЯП сверхвысокого уровня – это языки с высоким уровнем абстракции, используемые для специфических приложений и задач. В сверхвысокоуровневых языках программирования описывается лишь принцип «что нужно сделать». В связи с этой ограниченностью они могут использовать синтаксис, который значительно отличается от других ЯП.

Сочетание языков высокого и низкого уровня дает оптимальные результаты в решении сложных вычислительных задач.

Первый ЯП для ЭВМ первого поколения (начало 40-х гг. XX в.) являлся *машинным*, т. е. представлял собой только числовой код, в котором записывались команды, операнды и данные.

Затем (в середине 50-х гг. XX в.) были созданы *машинно-ориентированные языки* программирования символьного кодирования. Это система обозначений, используемая для представления в удобочитаемой форме программ, записанных в машинном коде. Несмотря на высокую трудоемкость, ими часто пользуются профессиональные системные программисты, например, при разработке программ общего или специального программного обеспечения (ПО), особенно в тех случаях, когда эти программы должны быть максимально компактными и быстродействующими (например, Ассемблер).

Развитие компьютерной техники и ЯП привело к разработке парадигмы, основанной на использовании *процедурно-ориентированных* ЯП (1954 г.), которые относятся к языкам программирования высокого уровня. В их основу положен принцип последовательности действий, позволяющий решить поставленную задачу с помощью заранее составленных шаблонов (процедур и функций). Этот вид ЯП применяется только для конкретного класса задач, вне которого он неэффективен (*Fortran, Basic, C*).

*Процедурное программирование* основано на записи последовательности операторов (инструкций), задающих процедуру (последовательность шагов) решения задачи. Одним из основных операторов является оператор присваивания, служащий для изменения содержимого областей памяти компьютера.

*Структурное программирование* – методология разработки программного кода, в основе которой лежит представление программы в виде иерархической структуры блоков. Повторяющиеся фрагменты программы (представляющие собой логически целостные вычислительные блоки) оформляются в виде подпрограмм (*процедур* или *функций*).

Следующим шагом в эволюции языков программирования стало появление *объектно-ориентированных языков* программирования, к которым относится и рассматриваемый в данном пособии язык C++.

Объектно-ориентированные языки программирования представляют собой методологию программирования, основанную на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

*Объектно-ориентированное программирование* (ООП) – парадигма программирования, в которой основными концепциями являются понятия *объектов* и *классов*.

*Объект* – сущность в адресном пространстве вычислительной системы, появляющаяся при создании *класса*.

*Класс* – это тип, описывающий устройство объекта.

*Прототип* – это объект-образец, по образу и подобию которого создаются другие объекты, причем следующий объект наследует свойства родительского объекта. Например, в качестве объекта можно рассматривать чертеж здания, а класса – выстроенное здание; классы – автомобили Тойота, объект – транспортное средство.

### **1.3. Трансляция программного кода высокоуровневых ЯП**

Программу, написанную на ЯП высокого уровня, необходимо перевести (транслировать) на язык машинных команд, «понятных» ЭВМ. Эту операцию выполняют обслуживающие программы-трансляторы (*интерпретаторы* и *компиляторы*).

*Интерпретаторы* построчно переводят программу в машинные коды и при отсутствии синтаксических и семантических ошибок в строке она полностью выполняется. Интерпретаторы более удобны при отладке программ, так как реализуют диалоговый стиль разработки (например, Бэйсик, Лого и др.).

*Компиляторы* после проверки программного кода на отсутствие ошибок преобразуют весь текст в машинные коды в ходе одного непрерывного процесса. В результате этого создается отдельный файл в машинных кодах, исполнение которого не требует наличия на компьютере среды, в которой создавался программный код. В этом заключается преимущество компиляторов перед интерпретаторами, кроме этого необходимо отметить высокую скорость выполнения программы (для языков Фортран, Паскаль, С, C++).

## 2. СРЕДА РАЗРАБОТКИ ПРИЛОЖЕНИЙ MICROSOFT VISUAL STUDIO. КОМПИЛЯТОР VISUAL C++

### 2.1. Общие сведения

C++ – компилируемый, статически типизированный ЯП высокого уровня, поддерживающий разные парадигмы (подходы) программирования и сочетающий свойства как высокоуровневых, так и низкоуровневых языков. Язык возник в начале 1980-х гг. благодаря усилиям сотрудника фирмы *Bell Labs* Бьёрни Страуструпа, который создал ряд усовершенствований к языку C.

Название «C++» происходит от языка C, к которому добавлен унарный оператор ++, обозначающий операцию *инкремента* (увеличения на единицу) значения переменной. В языке C++ (в сравнении с языком C) наибольшее внимание уделено поддержке объектно-ориентированного и обобщенного программирования.

### 2.2. Основные понятия C++

Язык C++ представляет собой строго определенный набор команд (исходный программный код), предназначенных для выполнения компьютером. Командами являются *ключевые* (зарезервированные) слова, которые являются основными блоками программного кода.

*Ключевые* слова представляют собой указания к выполнению различных операций, например, от указания типа данных до использования стандартных тригонометрических функций. Кроме этого с помощью ключевых слов создаются более сложные блоки программного кода – пользовательские функции, представляющие собой самостоятельные части программного кода и записанные в терминах простых стандартных функций.

Так как C++ является типизированным, то, приступая к написанию программного кода, следует определить типы используемых данных и ожидаемых результатов его выполнения. Нарушение этого условия может приводить к аварийному завершению выполнения программы (например, ввиду действий, несовместимых с данным типом) или получению неверного результата.

### 2.3. Алфавит языка C++

*Алфавит* ЯП – разрешенный к использованию стандартом языка набор символов, с помощью которых записываются ключевые служебные слова и

элементы текста программного кода. Алфавит языка C++ включает в себя следующее:

- прописные и строчные буквы латинского алфавита (A – Z, a – z);
- десятичные цифры (0 – 9);
- знаки арифметических операций (+, −, \*, /, \);
- знаки пунктуации: пробел, точка, запятая, точка с запятой, двоеточие, скобки (фигурные, круглые, квадратные), кавычки, апостроф;
- специальные символы (? , ! , & , # , % и др.).

Использование алфавита языка позволяет формировать следующие элементы программного кода:

- *идентификаторы* (имена объектов – констант, переменных, функций);
- ключевые (служебные) слова;
- знаки операций в математических выражениях;
- разделители (знаки пунктуации).

## 2.4. Типы данных C++

К фундаментальным понятиям любого языка программирования относят *тип* данных, который определяет область допустимых значений, набор и правила выполнения операций, которые можно выполнять над этими значениями, а также способ хранения данных (место хранения и объем в памяти компьютера). Элементарными конструкциями языка C++ являются типизированные данные, которые в программе могут использоваться в виде констант, переменных, массивов.

Наиболее распространенными типами данных являются числовой и символьный (текстовый). Основные числовые типы данных языка C++ и их размеры в памяти компьютера приведены в табл. 2.1.

Таблица 2.1

Основные типы данных языка C++

Тип данных	Название	Размер	Тип данных	Название	Размер
1	2	3	4	5	6
int	Целый	2 байта	long int	Длинный целый	4 байта
float	Вещественный	4 байта	short int	Короткий целый	2 байта

1	2	3	4	5	6
double	Вещественный двойной точности	8 байт	signed short int	Знаковый короткий целый	2 байта
char	Символьный	1 байт	unsigned long int	Беззнаковый длинный целый	2 байта

Каждый тип данных характеризуется своим диапазоном значений (см. табл. 2.2).

Таблица 2.2

Диапазоны некоторых типов данных

Тип данных	Диапазон	Размер, бит
int	$-2147483648 \div 2147483647$	16
float	$-3.4e-38 \div 3.4e+38$	32
double	$1.7e-308 \div 1.7e+308$	64
char	$0 \div 255$	8

При написании программного кода обязательным является объявление и отслеживание типов данных в процессе преобразований программы. В противном случае среда программирования C++ может выполнять самостоятельное преобразование типов данных, что приведет к потере точности вычислений.

## 2.5. Синтаксис идентификаторов и ключевых слов в C++

При записи идентификаторов (имен переменных, констант и функций, создаваемых программистом) в C++ необходимо строго выполнять определенные правила:

имена переменных, констант и функций должны начинаться только с латинской буквы (прописной или строчной). В имени допускается использование цифр или знака подчеркивания, например: *age*, *Age*, *AGE*, *summa\_1*, *Z*. Компилятор C++ различает прописные и строчные буквы;

имя не может содержать знаки операций, пунктуационные знаки (кроме символа подчеркивания) или специальные символы;

имя должно быть уникальным в программном коде, т. е. не должно повторяться;

в качестве имени недопустимо использовать ключевые (служебные) слова языка C++, наименования процедур, операторов, функций и др.

Каждая переменная, константа или функция характеризуется именем (*идентификатором*), типом и значением.

Общепринятая практика – присваивать идентификаторам информативные имена, например: *summa*, *massiv\_1*, *stroka*.

## 2.6. Особенности объявления констант и переменных. Комментарии

Константы и переменные представляют собой именованные ячейки памяти для хранения данных. Отличие константы от переменной заключается в том, что значение константы не может быть изменено при выполнении программы.

Для объявления констант или переменных необходимо определить правильное имя и тип данных, например: *int Sum* – объявление переменной целого типа с именем *Sum*.

Кроме этого необходимо задать для константы или переменной значение (выполнить *инициализацию*) в соответствии с объявленным типом до их первого вызова в программе. *Инициализация* константы или переменной – процедура объявления типа в совокупности с присвоением значения.

В C++ для представления константы рекомендуется использовать объявление идентификатора с типом и начальным значением и ключевым словом *const*:

```
const int n = 10.
```

Если значение константы или переменной не задано, то происходит аварийное завершение выполнения программы.

Возможно перечисление нескольких идентификаторов одного типа данных, например: *int a, b, c* или *int a=2, b=5, c*. Недопустимо в одной строке программного кода перечислять константы или переменные различных типов:

```
int a, b, double c.
```

При написании программного кода в некоторых случаях требуется внести пояснения к выполняемым в строке действиям, т. е. комментировать строки программного кода.

*Комментарии* представляют собой текст, который игнорируется компилятором, и служат только для пояснений о выполняемых действиях в отдельной

строке или целом блоке программного кода. При написании комментариев могут быть использованы латинские или кириллические символы.

В языке C++ используются два типа комментариев:

- для пояснений в строке – двойная наклонная черта (//);
- для пояснений в блоке (нескольких строках программы) – совокупность наклонной черты и звездочки (/\*).

Пример комментария:

```
int x, y; //объявление переменных целого типа
```

```
x = 5; //присвоение (инициализация) переменной x значения 5
```

При использовании блочных комментариев компилятор игнорирует все строки программного кода, которые следуют за символами (/\*), до тех пор, пока не встретится символ завершения блока комментариев – звездочка и наклонная черта (\*). Каждой открывающей блок комментариев паре символов (/\*) должна соответствовать закрывающаяся пара символов (\*).

Пример блока комментариев:

```
y = x++; /* переменной y присвоено значение переменной x, затем значение переменной x увеличивается на единицу */
```

## 2.7. Встроенные функции

C++ предлагает огромный набор встроенных функций, предназначенных для выполнения операций с данными различных типов. Ввиду узкой направленности настоящего пособия далее рассмотрим в основном математические функции.

Перечень некоторых встроенных функций приведен в табл. 2.3.

Таблица 2.3

Встроенные функции C++

Функция	Обозначение функции	Тип		Файл описания*
		аргумента	функции	
1	2	3	4	5
Функции с одним аргументом				
Абсолютное значение	abs(x)	int	int	<stdlib.h>
	fabs(x)	double	double	<math.h>

1	2	3	4	5
Арккосинус	acos(x)	double	double	<math.h>
Арсинус	asin(x)	double	double	<math.h>
Арктангенс	atan(x)	double	double	<math.h>
Косинус	cos(x)	double	double	<math.h>
Синус	sin(x)	double	double	<math.h>
Тангенс	tan(x)	double	double	<math.h>
Округление до большего целого	ceil(x)	double	double	<math.h>
Округление до меньшего целого	floor(x)	double	double	<math.h>
Экспоненциальная функция $e^x$	exp(x)	double	double	<math.h>
Степенная функция $x^y$	pow(x,y)	double	double	<math.h>
Логарифм натуральный	log(x)	double	double	<math.h>
Логарифм десятичный	log10(x)	double	double	<math.h>
Корень квадратный	sqrt(x)	double	double	<math.h>
Функции с двумя аргументами				
Остаток деления $x$ на $y$	fmod(x,y)	double	double	<math.h>
Возведение в сте- пень	pow(x, y)	double	double	<math.h>

\* – описание файлов см. в разд. 5.1.

Существуют строгие правила записи встроенных функций:

- имена функций записываются только строчными латинскими буквами;
- аргумент функции записывается только в круглых скобках;
- при записи сложных аргументов необходимо соблюдать баланс скобок (количество открытых и закрытых скобок должно быть одинаковым).

Использование встроенных функций возможно только при указании стандартных библиотек языка C++ (о подключении библиотек см. в разд. 5.1).

В табл. 2.3 приведены функции с одним и двумя аргументами. Для функций с двумя аргументами при перечислении аргументов используется запятая (табл. 2.4).



## Пояснения к использованию функций с двумя аргументами

Название функции	Имя функции	Примечание
1	2	3
Остаток деления $x$ на $y$ , проверка кратности, проверка четности /нечетности числа	<code>fmod(x,y)</code>	<p>1. Проверка кратности (для типа <i>double</i>):  <math>(fmod(x, y) == 0)</math> – условие <math>x</math> кратно <math>y</math>            Проверка кратности:  <math>(fmod(z, 7) == 0)</math> – условие <math>z</math> кратно 7</p> <p>2. Проверка четности числа:  <math>(fmod(x, 2) == 0)</math>            Проверка нечетности числа:  <math>(fmod(x, 2) != 0)</math></p>
Возведение в степень (показатель степени целый или вещественный)	<code>pow(x, y)</code>	<p>1. Показатель степени целого типа:  <math>x^2 \rightarrow \text{pow}(x, 2)</math></p> <p>2. Показатель степени вещественного типа:  <math>x^3 \rightarrow \text{pow}(x, 1./3)</math>            (точка после цифры 1 приводит результат деления к вещественному типу)</p>

## 3. ЗАПИСЬ АРИФМЕТИЧЕСКИХ ВЫРАЖЕНИЙ

## 3.1. Правила построения математических выражений

При записи арифметических выражений (табл. 3.1) в программном коде необходимо соблюдать следующие правила:

- 1) все символы математических операций в арифметическом выражении должны быть указаны (нельзя опускать знак умножения);
- 2) нельзя указывать подряд два знака математических операций;
- 3) должен соблюдаться баланс скобок – количество открытых и закрытых скобок должно быть одинаковым;
- 4) после имени функции должна следовать открывающая круглая скобка (указание знаков умножения или пробела является ошибкой);
- 5) приоритет выполнения операций соответствует правилам математики. Порядок выполнения операций может быть изменен с помощью скобок в выражении;

б) невозможно указывать подстрочные или надстрочные символы – запись выражения производится в строку.

Примеры записи математических выражений на языке C++ приведены в табл. 3.1.

Таблица 3.1

Примеры записи математических выражений на языке C++

Математическая запись	Запись на C++
$2a \cdot (-b)$	<code>2 * a * (-b)</code>
$\frac{ax^2 + bx + c}{\sqrt[3]{d - 2,5}}$	<code>(a * pow(x, 2) + b * x + c) / pow((d - 2.5), 1. / 3)</code>
$\sin^2 x - \cos x^3$	<code>pow(sin(x), 2) - cos(pow(x, 3))</code>

### 3.2. Оператор присвоения. Операции инкремента и декремента

Операторы присвоения являются *бинарными*, т. е. подразумевают наличие двух переменных в выражении. Простейший оператор присвоения символ «=» позволяет сохранить значение правого аргумента в левом. Пример:

*int a = 10 //объявление переменной a целого типа и инициализация значением 10;*

*int b = a + 100 //объявление переменной b целого типа, присвоение ей значения переменной a и сложение с числом 100. При этом b используется в качестве адреса, в который записывается результат вычисления a + 100.*

При выполнении вычислений часто возникает вопрос о необходимости увеличения или уменьшения значений переменной на единицу (например, подсчет количества событий). В языке C++ операция увеличения значения числовой переменной на единицу называется инкрементом, а уменьшения – декрементом. Выполнение операции инкремента осуществляется путем указания символа (++) рядом (перед или после) с именем переменной, для декремента указывается символ (--). Такие операции являются унарными, т. е. выполняются для одного операнда.

Для операторов инкремента/декремента существует две формы записи: перед именем переменной префиксная форма, после имени переменной – постфиксная. Например, *myAge++* (или *++myAge*) увеличивает значение переменной *myAge* на единицу, т. е. выполняется операция *myAge = myAge + 1*. Однако префиксная и постфиксная формы записи имеют разный смысл:

префиксный оператор инкремента/декремента вычисляется до присвоения (передачи значения), например:

```
int count = 1; //присвоить начальное значение переменной целого типа count
int NewCount = ++count; //присвоить значение переменной целого типа NewCount с помощью инкремента count в префиксной форме.
```

После выполнения действий получим:  $NewCount = 2$ ,  $count = 2$ ;

постфиксный оператор инкремента/декремента вычисляется после операции присвоения, для этого же примера:

```
int count = 1; //присвоить начальное значение переменной целого типа count
int NewCount = count++; //присвоить значение переменной целого типа NewCount с помощью инкремента count в постфиксной форме.
```

После выполнения действий получим:  $NewCount = 1$ ,  $count = 2$ .

Для изменения значения переменной не на единицу используются специальные операторы  $+=$ ,  $-=$ ,  $*=$ ,  $/=$ :

$x += 0.5$ ; //увеличивает значение переменной  $x$  на 0,5, т.е.  $x = x + 0.5$ ;

$y -= 0.3$ ; //уменьшает значение переменной  $y$  на 0,3;

$z *= 3$ ; //умножает значение переменной  $z$  на 3;

$d /= 8$ ; //делит значение переменной  $d$  на 8.

### 3.3. Логические операторы

Наиболее распространенной синтаксической конструкцией языка C++ являются выражения, содержащие арифметические операторы. Кроме этого в языке C++ существует целый класс логических операторов. Результатом применения таких операторов является ЛОЖЬ или ИСТИНА, т. е. значение булевого типа. Логические операторы подразделяются на два типа – простые и битовые. В табл. 3.2 приведены сведения только о логических операторах простого типа.

Таблица 3.2

Простые логические операторы

Символ	Понятие	Значение
1	2	3
!	Не (инверсия)	Истинно, если операнд принимает ложное значение
!=	Не равно	Истинно, если один операнд не равен второму

1	2	3
==	Равно	Истинно, если оба операнда равны друг другу
>, <	Больше, меньше	Истинно, если левый операнд больше (меньше) правого
>=, <=	Больше или равно, меньше или равно	Истинно, если левый операнд больше или равен (меньше или равен) правому
&&	И	Истинно, если оба операнда истинны
	Или	Истинно, если хотя бы один из операндов истинный

Примеры записи логических операторов:

*!sum //ИСТИНА, если значение sum ЛОЖЬ;*

*x != 3 //ИСТИНА, если x не равно 3;*

*n == m //ИСТИНА, если n равно m;*

*a > 100 //ИСТИНА, если a больше 100;*

*y > 123.45 //ИСТИНА, если y больше либо равен 123.45;*

*(summa == 5) && (razn > 10) //ИСТИНА, если summa равна 5 И при этом razn больше 10;*

*(s1 >= 5) || (Pr == 10) //ИСТИНА, если s1 больше или равна 5 ИЛИ Pr равна 10.*


#### 4. СРЕДА РАЗРАБОТКИ ПРИЛОЖЕНИЙ MICROSOFT VISUAL STUDIO

Комплект интегрированной среды разработки *Microsoft Visual Studio* включает в себя компилятор *Visual C++*. Компилятор *Visual C++* поддерживает перечень приложений и позволяет генерировать код как для платформы *.NET Framework*, так и для исполнения в среде *Windows*. Далее рассматривается одна из версий интегрированной среды – *Microsoft Visual Studio 2013*.

##### 4.1. Создание проекта C++ в среде Microsoft Visual Studio

В компьютерных классах кафедры «Информатика и компьютерная графика» (ИКГ) установлена версия *Microsoft Visual Studio 2013* (русифицированная версия). Запуск среды программирования *Visual C++* для создания проекта может быть осуществлен разными способами:

двойным щелчком левой кнопки мыши по значку (ярлыку) на рабочем столе;

использованием кнопки Пуск – Все программы – *Visual Studio 2013* –  *Visual Studio 2013*.

Использование любого из этих способов запускает начальную страницу *Microsoft Visual Studio* (рис. 4.1).

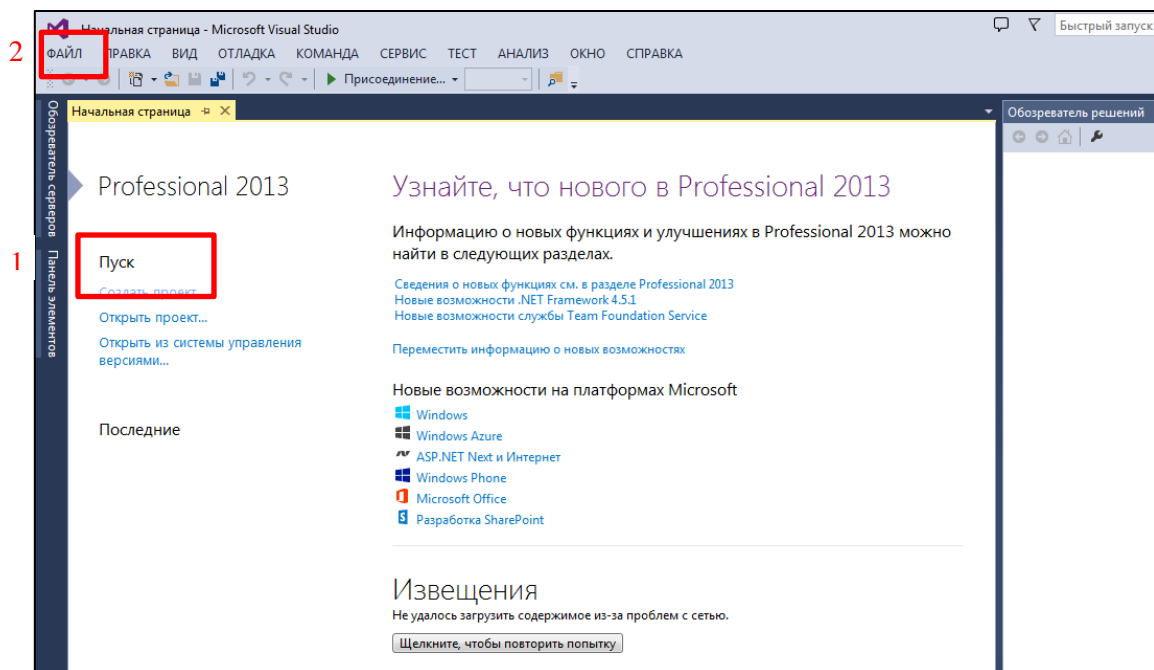


Рис. 4.1. Начальная страница *Microsoft Visual Studio*

Создание проекта после запуска среды возможно путем выбора надписи Создать проект (1) или в пункте меню *Файл* (2) *Создать – Проект*. В раскрывшемся окне *Создать проект* (рис. 4.2) выполнить следующие действия:

- выбрать *Установленные* (1) – *Шаблоны* (2) *Visual C++ – Консольное приложение Win32* (3);

- задать *Имя* проекта (4), например, **Lab4-1**;

- указать *Расположение* – путь к папке, в которой будет сохранен проект.

По умолчанию проект будет размещен на диске *C*, поэтому в случае необходимости следует нажать кнопку *Обзор...*(5).

После необходимых изменений и настроек нужно нажать на кнопку *ОК* (6), в следующем появившемся окне *Мастер приложений* (рис. 4.3) нажать кнопку *Готово* (7).

*Консольное приложение Win32* – это наиболее простой шаблон приложения, в котором создается начальная структура для написания программного кода. Созданный проект располагается в месте, указанном при его создании. В папке проекта содержатся вложенная папка *Debug*, которая содержит информа-

ция об изменениях и отладке программного кода, и основной файл проекта Имя\_проекта.cpp и файлы, генерируемые системой.

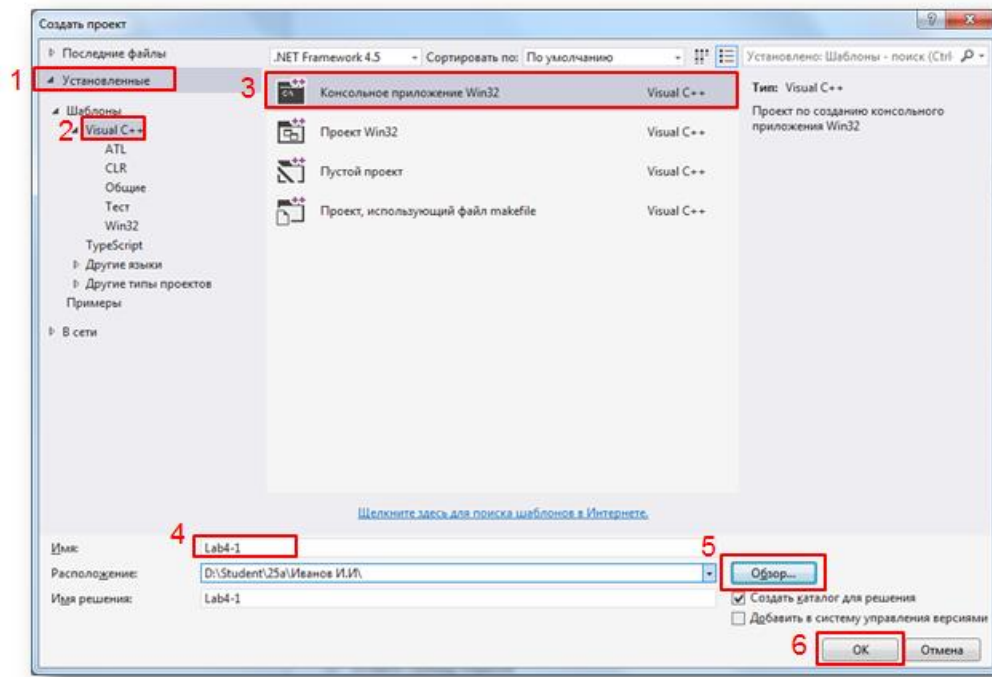


Рис. 4.2. Создание проекта в *Microsoft Visual Studio*

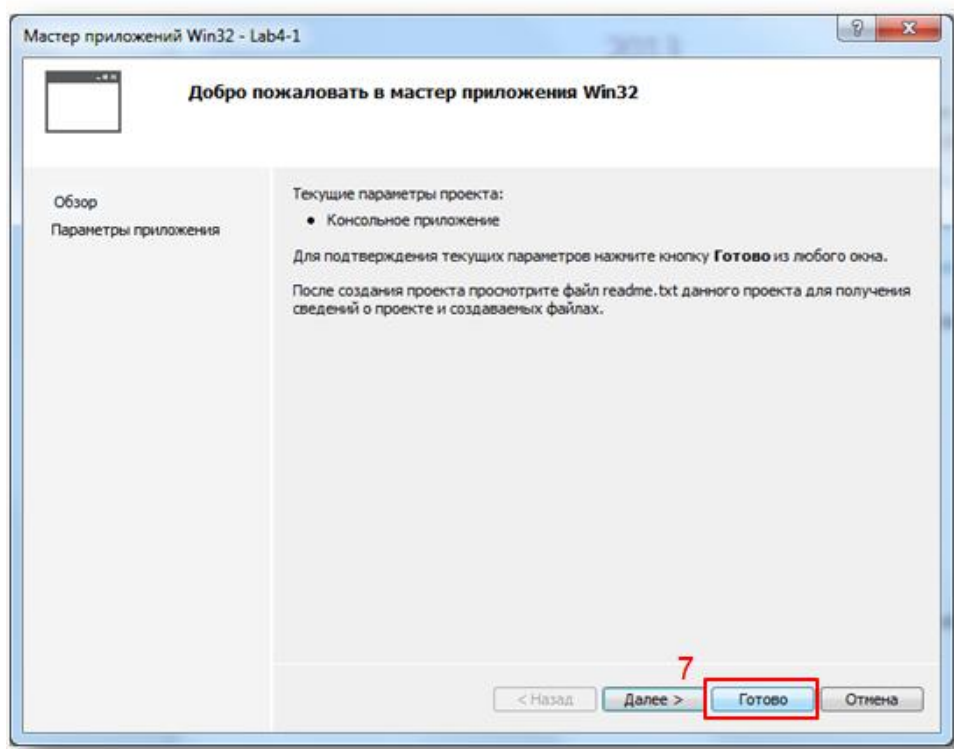


Рис. 4.3. Окно Мастера приложений

## 4.2. Области окна проекта

Окно проекта (рис. 4.4) разделено на следующие области – окно проводника (1), окно программного кода (2) и окно с диагностической информацией (3).

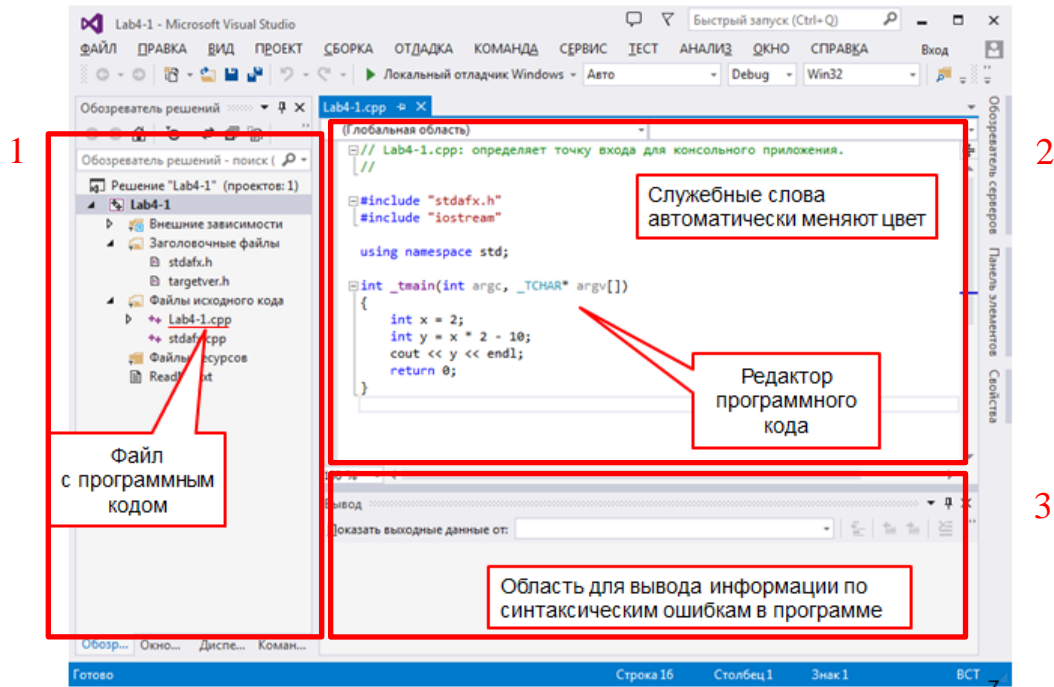


Рис. 4.4. Окно проекта с программным кодом

В окне проводника отображается список файлов проекта, в том числе: *stdafx.h* – заголовочный файл, отвечающий за перекомпиляцию программного кода после внесения в него изменений;

*имя\_файла.cpp* – файл проекта с программным кодом, расширение файла указывает на язык C++.

Заголовочный файл (*.h* от англ. *head*) – подключаемый файл, содержимое которого автоматически добавляется *препроцессором* в исходный текст. Использование заголовочных файлов в C++ – основной способ подключить к программе типы данных и другие конструкции языка, в общем случае они содержат любые конструкции языка программирования из стандартных библиотек.

*Препроцессор* C++ (англ. *pre processor*, предобработчик) – программа, подготавливающая код программы на языке C++ к компиляции. Препроцессор выполняет следующие действия:

- заменяет некоторые сочетания символов специальными символами;

- заменяет комментарии пустыми строками (с удалением пробелов и символов табуляции);
- включает в программный код указанные в разделе подключений заголовочные файлы;
- выводит диагностические сообщения.

Этапы работы препроцессора:

- лексический анализ программного кода (синтаксический анализ при этом не выполняется);
- обработка директив (указаний) препроцессору;
- выполнение подстановок символов и программного кода заголовочных файлов.

## 5. СОЗДАНИЕ ПРОГРАММНОГО КОДА

### 5.1. Разделы окна программного кода

При создании консольного приложения *Win32* описанным в разд. 4.1 способом в окне программного кода сверху вниз расположены следующие элементы:

раздел комментариев – начинается символами *//* и содержит пояснения к созданному файлу (текст зеленого цвета);

раздел подключений, в котором указываются директивы препроцессору о подключении заголовочных файлов из стандартных библиотек (*#include*);

раздел программного кода – начинается с главной функции программного кода *int \_tmain(int argc, \_TCHAR\* argv[ ])*, которая в свою очередь содержит операторные скобки *{}*.

Каждая строка раздела подключений начинается символом *#*, который служит указанием для препроцессора о необходимости чтения и обработки строк с этим символом в первую очередь.

В строках программного кода, начинающихся с *#include* (хэш-инклюд) записывается название требующегося заголовочного файла с указанием пути в папку *<include>*. Для *#include "..."* – поиск заголовочного файла выполняется в текущей папке проекта, которая была указана при создании проекта. Для *#include <...>* – поиск заголовочного файла выполняется в папках, содержащих файлы стандартной библиотеки (пути к этим папкам зависят от реализации компилятора).



Одним из вариантов упрощений при написании программного кода является написание в разделе объявлений строки *using namespace std*, которая представляет собой директиву препроцессору для использования стандартного пространства имен при вводе-выводе данных. Эта строка должна завершаться символом «;».

Для написания программного кода для решения простейшей математической задачи (с использованием тригонометрических функций) необходимо подключить следующие заголовочные файлы:

*#include <math.h>* – подключение заголовочного файла, отвечающего за использование математических функций;

*#include <iostream>* – подключение заголовочного файла (библиотеки) потокового ввода-вывода. Особенностью компилятора в *Visual C++* является написание *iostream* без указания расширения *.h*;

*#include <iomanip>* – подключение заголовочного файла, содержащего описание действий при работе с потоковыми операциями ввода-вывода. Например, возможно использование в программном коде функции *setprecision*, позволяющей выполнять форматирование вывода данных – указанного количества символов в выводимом значении.

Раздел программного кода начинается строкой *int \_tmain(int argc, \_TCHAR\* argv[ ])*.

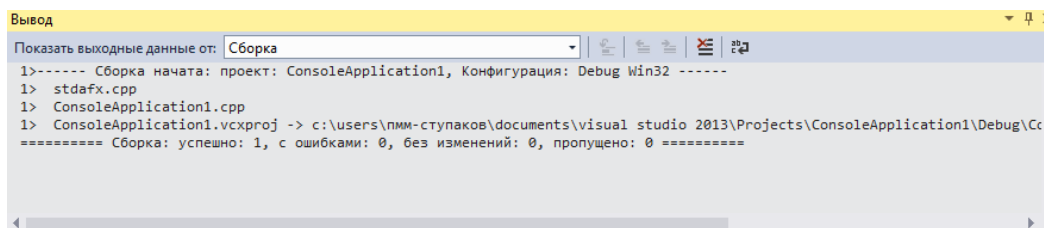
*int \_tmain* – главная функция целого типа (имя *\_tmain* встречается только **один раз** в программе и не может быть использовано повторно);

*int argc* – счетчик количества аргументов функции *\_tmain*, передаваемых системе для запуска программы;

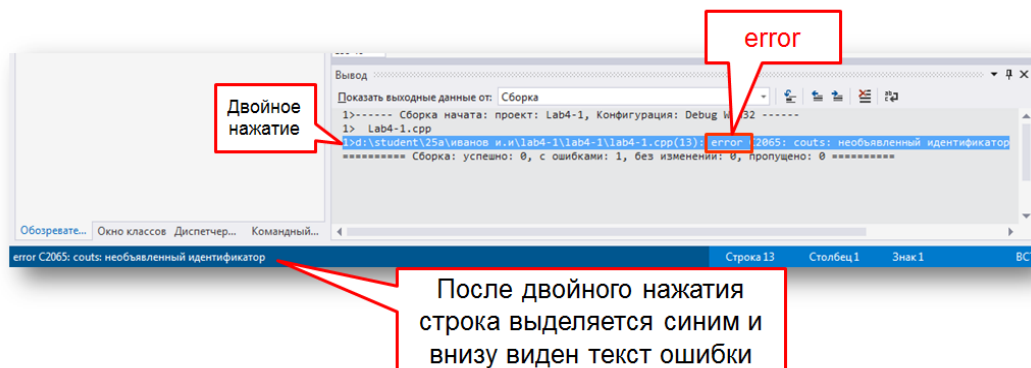
*\_TCHAR\* argv[ ]* – массив указателей, каждый из которых ссылается на один из аргументов.

После главной функции указаны операторные скобки {}, в которых располагается *тело* программы. Перед закрывающей скобкой (}) указывается ключевое слово *return 0*, которое служит для возврата в точку, где была вызвана функция, *0* – результат, возвращаемый системе.

Окно диагностических сообщений при компиляции программы содержит строки с информации о процессе компиляции (рис. 5.1).



а



б

Рис. 5.1. Окно диагностических сообщений:

а – результат успешной компиляции; б – наличие ошибок при компиляции

## 5.2. Структура программного кода

При написании программного кода синтаксические ошибки в ключевых словах или при использовании букв кириллицы автоматически подчеркиваются красной волнистой линией. При компиляции такой программы список ошибок отображается в окне диагностических сообщений (рис. 5.1, б).

Ошибки (*error*) необходимо начинать рассматривать с первой (верхней) строки в окне. Двойное нажатие на строке с ошибкой устанавливает курсор в строке программного кода, содержащей ошибку.

После исправления ошибок необходимо заново выполнить компиляцию программы и убедиться в ее успешном завершении (рис. 5.1, а).

В общем виде структура программного кода выглядит так:

```
#include "stdafx.h"
#include <имя библиотеки>
#include <имя библиотеки>
using namespace std;

int _tmain(int argc, _TCHAR* argv[ ])
```

```

    {
Строки программного кода; // комментарий
    /* блок
    комментариев */
return 0;
    }

```

Рассмотрим пример написания программного кода для решения простой задачи.

Сложить значения двух переменных разных типов и вывести результат:

```

#include "stdafx.h"
#include <iostream>
#include <math.h>
using namespace std;

```

```

int _tmain(int argc, _TCHAR* argv[ ]) //главная функция
{ //открывающая операторная скобка
    int slag_1=3; //инициализация slag_1
    double slag_2=3.7; //инициализация slag_2
    double summa=slag_1+cos(slag_2);
    cout<<"summa="<<summa<<endl; //вывод на экран тексто-
вой константы summa= и значения вычислений
    return 0; //завершение работы главной функции
} //закрывающая операторная скобка

```

Поясним строку `cout<<"summa="<<summa<<endl`:

- `cout<<` – выходной поток (си-аут) и оператор вставки;
- `"summa="` – текстовая константа. Для использования текстовой константы в выходном потоке необходимо обозначить ее символами (`"`);
- `<<summa` – оператор вставки в консоль печати для вывода значения идентификатора `summa`;
- `<<` – оператор вставки;
- `endl` – оператор перевода курсора в начало новой строки и очистка буфера вывода.

Кроме оператора `endl` для управления курсором предусмотрены специальные символы:

`\n` – символ перевода курсора в начало новой строки, не предусматривает очистки буфера вывода;

$\backslash t$  – символ горизонтальной табуляции;  
 $\backslash v$  – символ вертикальной табуляции;  
 $\backslash r$  – возврат курсора в начало текущей строки.

Специальные символы записываются в конце текстовой константы `cout<<"suma=/n"<<suma`. При этом записывать оператор `endl` не нужно.

Данная задача решена с использованием процедуры инициализации переменных разных типов *int* и *double*. Результат вычислений приводится к вещественному типу, так как для идентификатора *suma* объявлен тип *double*, кроме этого аргумент функции *cos()* должен быть вещественного типа. Каждая строка программного кода оканчивается символом (;), исключение составляют только строка главной функции *\_tmain* и строки с операторными скобками { }.

Внесем изменения в программный код – для инициализации идентификаторов *slag\_1* и *slag\_2* введем значения с клавиатуры. Приведем только изменяемую часть программного кода:

```
int slag_1; //объявление типа slag_1
double slag_2; //объявление типа slag_2
cout<<"Ввести значения slag_1, slag_2"; //вывод на экран текстовой константы Ввести значения...;
cin>> slag_1>> slag_2;
```

...

– `cin>> slag_1>> slag_2` – входной поток (си-ин);

– `>>slag_1>>slag_2` – операторы извлечения и идентификаторы. При исполнении данной строки программа ожидает ввода значения (на экране показан мигающий курсор), после указания которого продолжается ее выполнение. Так как во входном потоке указаны два идентификатора, то вводимые значения разделяются нажатием клавиши *Backspace* или нажатием клавиши *Enter*.

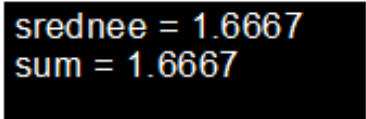
Нельзя во входном потоке выполнять перечисление идентификаторов через запятую – для этого используется оператор извлечения (>>).

Для обработки кириллицы в выходном потоке (только для комментариев выходного потока) необходимо в разделе объявлений записать строку `#include <locale.h>`, а в операторных скобках главной функции *\_tmain* – функцию `setlocale(LC_ALL, "Russian")`.

### 5.3. Влияние типа данных на результат вычислений

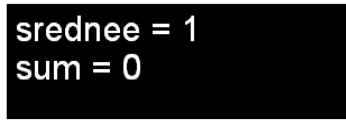
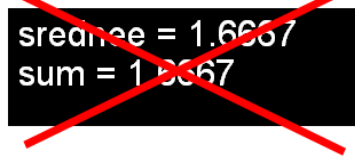
Так как язык C++ является типизированным, т. е. строго следит за объявленными типами идентификаторов, то это влияет на результат вычислений. Ниже приведены примеры, иллюстрирующие влияние типов данных на результат вычислений.

#### 1. Объявление идентификаторов *вещественного* типа:

Программный код	Результат вычислений
<pre>{   double x=1, x1=2, x2=2;   double srednee=(x+x1+x2)/3;   double sum=x/3+x1/3+x2/3;   cout&lt;&lt;"srednee = "&lt;&lt;srednee&lt;&lt;endl;   cout&lt;&lt;"sum = "&lt;&lt;sum&lt;&lt;endl;   return 0; }</pre>	

Все идентификаторы вещественного типа, поэтому и результат вычислений вещественный.


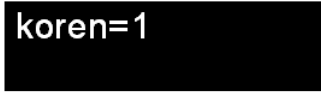
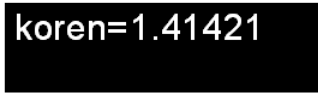
#### 2. Объявление идентификаторов *целого* типа в программном коде:

Программный код	Результат работы программы
<pre>{   int x=1, x1=2, x2=2;   int srednee=(x+x1+x2)/3;   int sum=x/3+x1/3+x2/3;   cout&lt;&lt;"srednee = "&lt;&lt;srednee&lt;&lt;endl;   cout&lt;&lt;"sum = "&lt;&lt;sum&lt;&lt;endl;   return 0; }</pre>	 

Все идентификаторы целого типа, поэтому и результат вычислений целый.

Далее приведены примеры, иллюстрирующие особенности объявления идентификаторов различных типов, потери точности вычислений и приведения результата вычислений к объявленному типу.

3. Объявление идентификаторов *целого* и *вещественного* типов в программном коде и результаты выполнения программных кодов:

Программный код	Результат работы программы
<pre>{   int x=2;   int koren=pow(x,1./2);   cout&lt;&lt;"koren="&lt;&lt;koren;   return 0; }</pre>	
<pre>{   double x=2;   int koren=pow(x,1./2);   cout&lt;&lt;"koren="&lt;&lt;koren;   return 0; }</pre>	
<pre>{   int x=2;   double koren=pow(x,1./2);   cout&lt;&lt;"koren="&lt;&lt;koren;   return 0; }</pre>	

#### 5.4. Компиляция программного кода. Запуск программы

После окончания написания программного кода необходимо выполнить его компиляцию. Компиляция – преобразование исходного программного кода в *объектный модуль*. Объектный модуль – двоичный файл, который содержит в себе особым образом подготовленный исполняемый код на языке ассемблера (машинный код, содержащий инструкции на языке процессора компьютера), объединенный с другими объектными файлами при помощи редактора связей (компоновщика) для получения готового исполняемого модуля. На следующем этапе модуль связывается с внешними библиотеками. При создании объектного файла выполняется проверка исходного программного кода на наличие/отсутствие синтаксических ошибок.

Весь этот процесс называется сборкой, запуск процесса сборки показан на [рис. 5.2](#).

После получения исполняемого файла выполняется его запуск (с отладкой – пошаговой проверкой – либо без отладки – без проверки). Эти последовательность действий показана на [рис. 5.3](#).

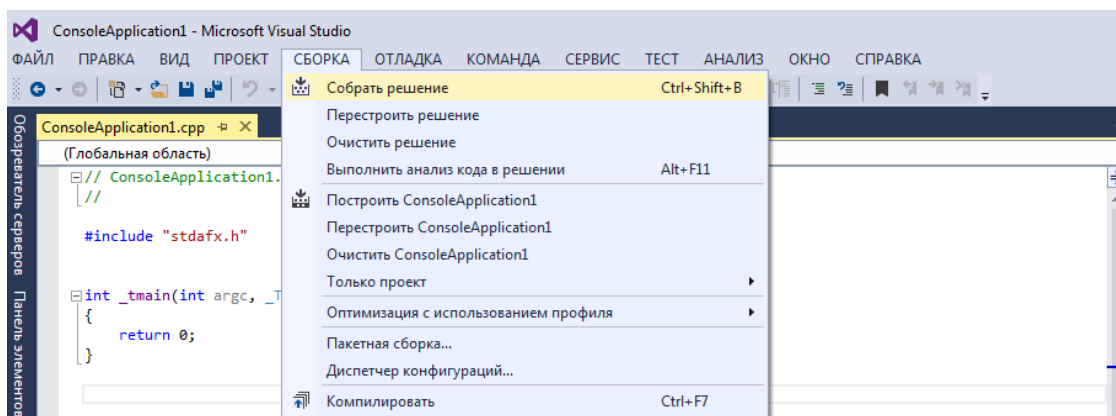


Рис. 5.2. Создание исполняемого файла



Рис. 5.3. Запуск исполняемого файла

Для запуска программы можно использовать клавиши:

- F11 – пошаговый запуск программы;
- Ctrl + F5 – запуск программы без отладки.

При пошаговом запуске программы для выполнения следующей строки программы необходимо нажимать F11. Для выхода из этого режима нажать Shift + F5. Использование комбинации Ctrl + F5 выполняет программу безусловно.

После внесения изменений в программный код необходимо выполнять обязательную повторную сборку для получения обновленного исполняемого файла. При игнорировании повторной сборки и запуске программы открывается окно с сообщением «Следующий проект устарел» ([рис. 5.4](#)) – необходимо выполнить повторную сборку. Для того чтобы указанное окно больше не появ-

лялось при следующих запусках программы, можно поставить «галочку» внизу окна «Больше не выводить это окно».

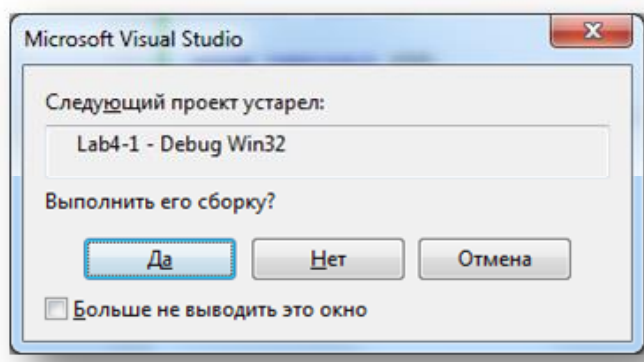


Рис. 5.4. Сообщение о необходимости повторной сборки

В случае синтаксических ошибок, выявленных на этапе повторной компиляции, возникает окно с соответствующим диагностическим сообщением о необходимости исправления ошибок в программном коде (рис. 5.5).

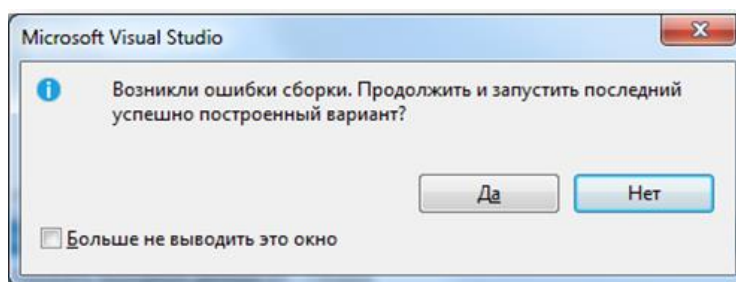


Рис. 5.5. Сообщение о возникших при компиляции ошибках

При запуске последнего успешно построенного варианта скомпилированной программы изменения, внесенные в проект, не будут приняты, и, соответственно, исполнение программы и результат будут прежними. Поэтому для создания измененного варианта программы необходимо перестроить решение (рис. 5.6).

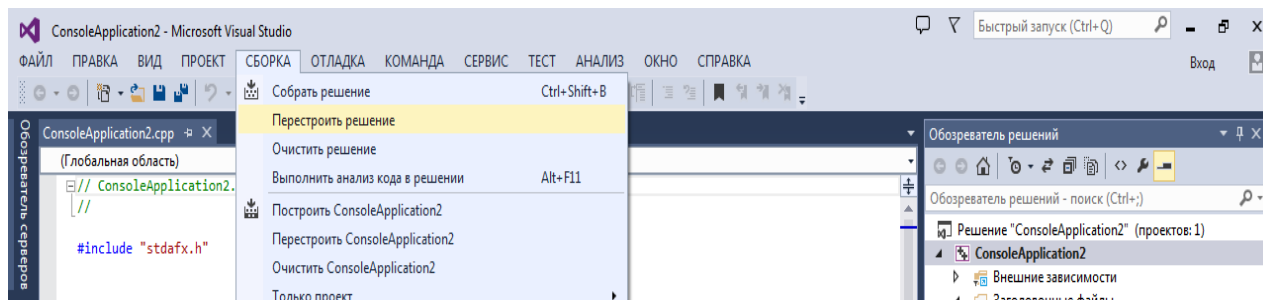


Рис. 5.6. Вкладка меню Сборка – Перестроить решение



## 6. ЛИНЕЙНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ ПРОЦЕССЫ

С основными понятиями и классификацией алгоритмов можно ознакомиться в работе [2].

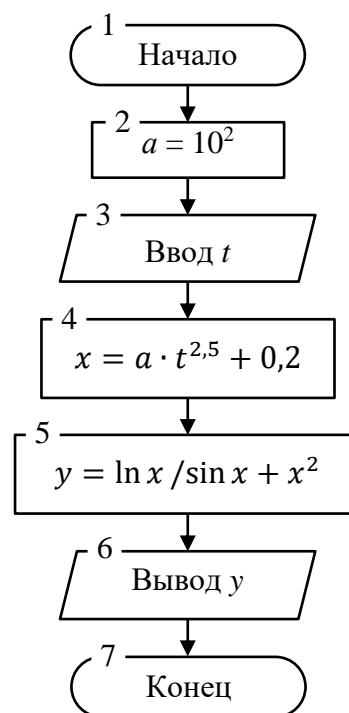
Алгоритм – строго определенная последовательность арифметических и логических действий для решения поставленной задачи.

Программы, реализующие алгоритм линейного вычислительного процесса, являются самыми простыми [2]. Они содержат только операторы ввода данных, присваивания и вывода результатов. Операторы записываются последовательно друг за другом в естественном порядке их следования, необходимом для решения задачи и получения результата и выполняются только один раз.

Рассмотрим пример 6.1 решения задачи с помощью линейного вычислительного процесса в C++. Необходимо составить графическую схему алгоритма (ГСА) и программу вычисления значения функции  $y = \ln x / \sin x + x^2$ . При этом задано, что  $x = a \cdot t^{2,5} + 0,2$ ,  $a = 10^2$ ,  $t$  – произвольное. Решение представлено на рис. 6.1.

```
1 #include "stdafx.h"
2 #include <iostream>
3 #include <math.h>
4 using namespace std;
5 int _tmain(int argc, _TCHAR* argv[ ])
6 {
7     int a=1e2;
8     double t;
9     cin>>t;
10    double x=a*pow(t,2.5)+0.2;
11    double y=log(x)/sin(x)+pow(x,2);
12    cout<<"y = "<<y<<endl;
13    return 0;
14 }
```

*а*



*б*

Рис. 6.1. Программный код (а) и ГСА (б) решения примера 6.1 с помощью линейного вычислительного процесса в C++

В программном коде (рис. 6.1, а) для последующего пояснения строки пронумерованы. Рассмотрим действия, выполняемые в каждой строке программного кода:

1 – указание препроцессору на подключение к данному программному коду заголовочного файла «*stdafx.h*», отвечающего за перекомпиляцию, который находится в одной папке с текущим проектом;

2 – указание препроцессору на подключение к данному программному коду файла (библиотеки) «*iostream*», отвечающего за возможность использования в программном коде операторов управления входным (*cin*) и выходным (*cout*) потоками;

3 – указание препроцессору на подключение к данному программному коду файла (библиотеки) «*math.h*», отвечающего за возможность использования в программном коде встроенных функций (*pow(x,y)*, *log(x)*, *sin(x)* и др.);

4 – использование в программном коде по умолчанию пространства имен «*std*», что упрощает запись строк, касающихся входного и выходного потоков (например, запись *cout<<"..."* вместо *std::cout<<"..."*);

5 – заголовок основной функции (процедуры) «*\_tmain*», которая содержит основную часть программного кода и выполняется при запуске проекта в среде C++;

6 – открытая операторная скобка функции (процедуры) «*\_tmain*»;

7 – инициализация переменной *a*;

8 – объявление типа переменной *t*;

9 – ввод значения переменной *t* с помощью входного потока (в данном случае с помощью клавиатуры);

10 – объявление типа переменной *x* и расчет ее значения;

11 – объявление типа переменной *y* и расчет ее значения;

12 – вывод результата расчета, значение переменной *y* выводится с помощью выходного потока на экран консольного приложения;

13 – окончание программного кода, возврат числа 0 функции (процедуре), запустившей выполнение текущей функции (процедуры) «*\_tmain*» программно;

14 – закрытая операторная скобка функции (процедуры) «*\_tmain*».

В ГСА (рис. 6.1, б) блоки расположены согласно записанному на рис. 6.1, а программному коду. Рассмотрим действия, представленные в каждом блоке ГСА, которые пронумерованы:

1 – начало графического описания алгоритма;

- 2 – присвоение переменной  $a$  заданного значения;
- 3 – ввод значения переменной  $t$  в процессе выполнения программы;
- 4 – расчет значения переменной  $x$  по заданной формуле;
- 5 – расчет значения переменной  $y$  по заданной формуле;
- 6 – вывод численного значения переменной  $y$  на экран;
- 7 – конец графического описания алгоритма.

## 7. РАЗВЕТВЛЯЮЩИЕСЯ ВЫЧИСЛИТЕЛЬНЫЕ ПРОЦЕССЫ

Большинство задач невозможно реализовать только с помощью линейной структуры, поскольку они обычно содержат различные условия, в зависимости от выполнения которых выбирается то или иное направление решения.

### 7.1. Оператор условного перехода **if**

#### 7.1.1. Простая форма оператора *if*

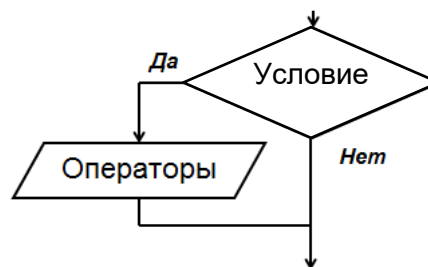
Самая простая – безальтернативная – форма записи, которая используется тогда, когда в случае истинности условия необходимо выполнить некоторый набор операторов, а при ложности (невыполнении) условия никаких действий не выполнять. Шаблон программного кода и ГСА для данной формы оператора *if* представлены на [рис. 7.1](#).

```

if (Условие)
{
    ...; //Ветвь «Да»:
    ...; //Операторы при выполнении Условия
}

```

*а*



*б*

**Рис. 7.1.** Программный код (*а*) и ГСА (*б*) простой формы оператора *if*

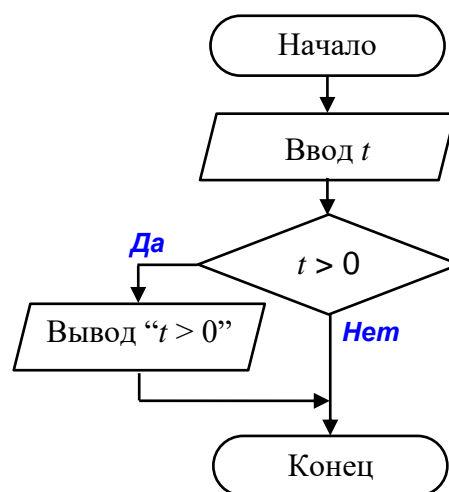
Условие – это логическое выражение, записанное с помощью операций отношения и принимающее значение ИСТИНА или ЛОЖЬ. Если условие истинно, то выполняются операторы, записанные после открытой операторной скобки «{». Если условие ложно, то никаких действий не выполняется. В программе управление передается оператору, следующему за закрытой операторной скобкой «}» в любом случае (при выполнении/невыполнении) логического выражения.

Рассмотрим [пример 7.1](#), который демонстрирует особенности использования простой формы оператора *if*.

При вводе с помощью клавиатуры произвольного вещественного числа  $t$  вывести сообщение « $t > 0$ », если введенное число положительное, в противном случае не выводить никаких сообщений. Решение представлено на [рис. 7.2](#) в виде программного кода и ГСА. Особенность реализации простой формы оператора *if* состоит в том, что строка, относящаяся к ветви «Да», не заключена в операторные скобки. Данная запись допускается в тех случаях, когда ветвь «Да» реализована с помощью одной строки программного кода.

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[ ])
{
    double t;
    cin>>t;
    if (t > 0)
        cout<<" t > 0"<<endl; //Ветвь Да
    return 0;
}
```

*а*



*б*

**Рис. 7.2.** Программный код (*а*) и ГСА (*б*) решения [примера 7.1](#)

### 7.1.2. Альтернативная форма оператора *if*

В тех случаях, когда при истинности условия необходимо выполнить один набор операторов, а при ложности – другой, применяется альтернативная форма оператора *if*. Шаблон программного кода и ГСА для данной формы оператора *if* представлены на [рис. 7.3](#).

Рассмотрим [пример 7.2](#), который выполняется с помощью альтернативной формы оператора *if*.

При вводе с помощью клавиатуры числа, которое не равно нулю, выполнить деление числа 10 на введенное число, в противном случае вывести сообщение «*Error*». Решение представлено на [рис. 7.4](#) в виде программного кода и ГСА. В данном решении, как и в решении [примера 7.1](#), отсутствуют операторные скобки у строк программного кода для ветви «Да» и для ветви «Нет». Это

допускается в том случае, когда каждая ветвь реализована с помощью одной строки программного кода.

```
if (Условие)
{
    ...; //Ветвь «Да»: Операторы
    ...; //при выполнении Условия
}
else
{
    ...; //Ветвь «Нет»: Операторы
    ...; //при невыполнении Условия
}
```

*a*

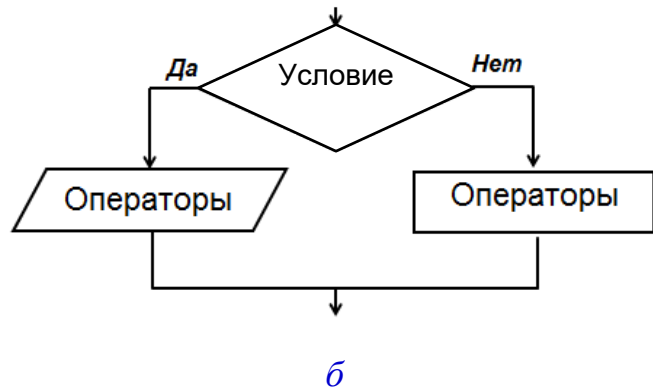


Рис. 7.3. Программный код (a) и ГСА (б) альтернативной формы оператора *if*

```
# include "stdafx.h"
# include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[ ])
{
    double x;
    cin>>x;
    if (x != 0)
        cout<<10/x<<endl; //Ветвь Да
    else
        cout<<" Error"<<endl; //Ветвь Нет
    return 0;
}
```

*a*

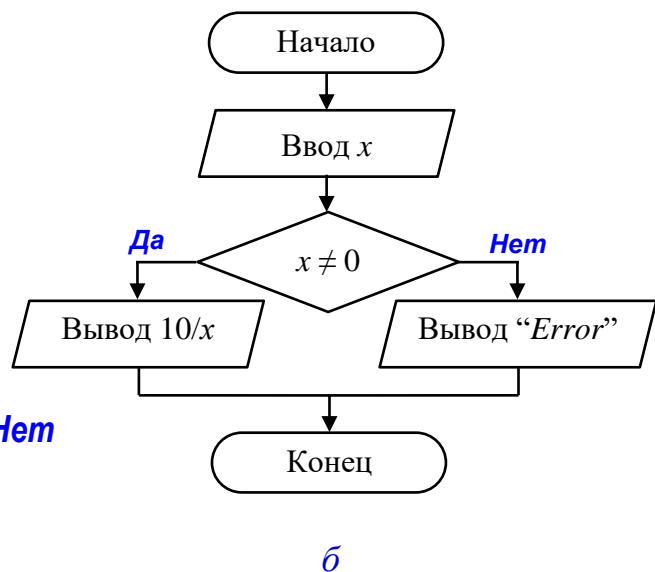


Рис. 7.4. Программный код (a) и ГСА (б) решения примера 7.2

### 7.1.3. Сложная форма оператора *if*

В тех случаях, когда необходимо реализовать выбор из более чем двух ветвей ГСА («Да» и «Нет»), применяется альтернативная (сложная) форма оператора *if* с вложенными операторами *if*. Шаблон программного кода для данной формы оператора *if* представлен на рис. 7.5, на рис. 7.6 приведена ГСА. На рис. 7.5 вложенное условие обозначено рамкой, оно находится в ветви «Нет1» ГСА.

```

if (Условие1)
{
    ...; //Ветвь «Да1»: Операторы
    ...; //при выполнении Условия1
}
else
{
    ...; //Ветвь «Нет1»: Операторы
    ...; //при невыполнении Условия2
    if (Условие2)
    {
        ...; //Ветвь «Да2»: Операторы
        ...; //при выполнении Условия2
    }
    else
    {
        ...; //Ветвь «Нет2»: Операторы
        ...; //при невыполнении Условия2
    }
}

```

Рис. 7.5. Программный код сложной формы оператора *if*

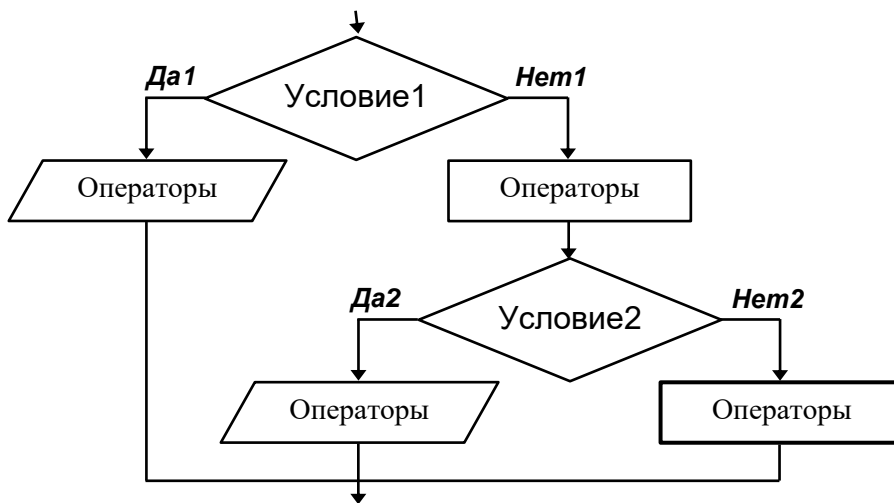


Рис. 7.6. ГСА сложной формы оператора *if*

Рассмотрим [пример 7.3](#), решение которого выполняется с помощью сложной формы оператора *if*.

Предусмотреть ввод вещественного числа  $x$  с помощью клавиатуры, а затем выполнить расчет значения  $y$  согласно следующим условиям:

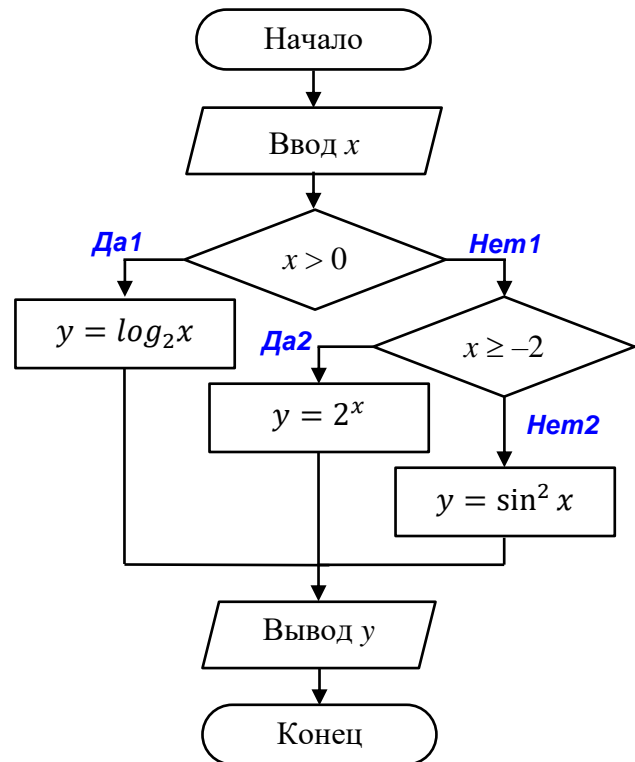
$$y = \begin{cases} \log_2 x & \text{при } x > 0, \\ 2^x & \text{при } -2 \leq x \leq 0, \\ \sin^2 x & \text{при } x < -2. \end{cases}$$

Результат расчета вывести на экран.

Решение [примера 7.3](#) представлено на [рис. 7.7](#) в виде программного кода ([рис. 7.1, а](#)) и ГСА ([рис. 7.1, б](#)).

```
#include "stdafx.h"
#include <iostream>
#include <math.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[ ])
{
    double x, y;
    cin >> x;
    if (x > 0)
        y = log(x)/log(2); //Ветвь Да1
    else
        if (x >= -2)
            y = pow(2,x); //Ветвь Да2
        else
            y = pow(sin(x),2); //Ветвь Нет2
    cout << "y = " << y << endl;
    return 0;
}
```

*а*



*б*

**Рис. 7.7.** Программный код (*а*) и ГСА (*б*) для решения примера 7.3

## 7.2. Объединение условий с помощью логических операций

Логические операции позволяют образовать сложное (составное) условие из нескольких простых (двух или более) условий. Эти операции упрощают структуру программного кода. Без использования логических операций количество блоков *if* в ГСА увеличивается в несколько раз в зависимости от условия задачи. В C++ используются следующие логические операции:

- логическая операция И, в программе обозначается «&&»;
- логическая операция ИЛИ, в программе обозначается «||»;
- логическая операция НЕ или логическое отрицание, в программе обозначается «!» (см. табл. 3.2).

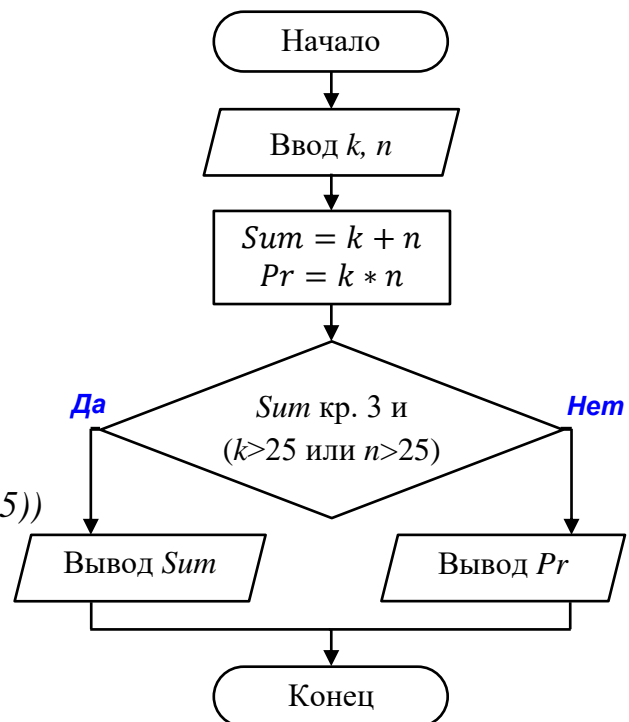
Если необходимо записать условие  $x > 0$  И  $y > 0$ , то в скобках оператора *if* необходимо записать:  $x > 0 \ \&\& \ y > 0$ . Если необходимо записать условие  $z > 0$  ИЛИ  $m > 0$ , то в скобках оператора *if* необходимо записать:  $z > 0 \ || \ m > 0$ . Если необходимо записать условие  $N \neq 0$ , то в скобках оператора *if* необходимо записать:  $N != 0$ .

Рассмотрим пример 7.4 (программный код и ГСА представлены на рис. 7.8), решение которого выполняется с помощью альтернативной формы оператора *if* и использованием условия с логическими операциями && (И) и || (ИЛИ).

Даны два числа –  $k$  и  $n$ . Если их сумма кратна трем и при этом хотя бы одно из них больше 25, вывести на экран их сумму. В противном случае вывести на экран их произведение.

```
#include "stdafx.h"
#include <iostream>
#include <math.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[ ])
{
    double k, n, Sum, Pr;
    cin>>k;
    cin>>n;
    Sum = k + n;
    Pr = k * n;
    if (fmod(Sum,3)==0 && (k>25 || n>25))
        cout<< Sum<<endl; //Ветвь Да
    else
        cout<< Pr <<endl; //Ветвь Нет
    return 0;
}
```

*а*



*б*

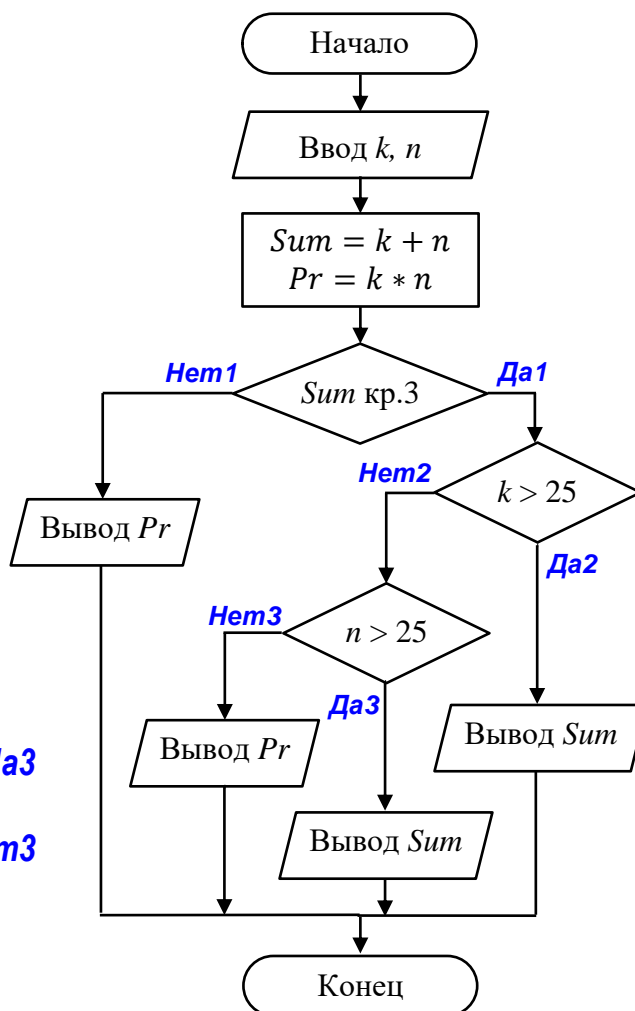
Рис. 7.8. Программный код (а) и ГСА (б) для решения примера 7.3



Данный пример решения выполнен с помощью сложной записи условия, но эту задачу можно решить и другим способом – с несколькими условиями, каждое из которых содержит запись условия без логических операций && (И) и || (ИЛИ). На [рис. 7.9](#) представлен вариант решения задачи из примера 7.3 без использования логических операций в условиях, но с использованием сложной формы записи оператора *if*.

```
# include "stdafx.h"
# include <iostream>
# include <math.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[ ])
{
    double k, n, Sum, Pr;
    cin>>k;
    cin>>n;
    Sum = k + n;
    Pr = k * n;
    if (fmod(Sum,3)==0)
    {
        if (k>25)
            cout<< Sum <<endl; //Ветвь Да2
        else
            if (n>25)
                cout<< Sum <<endl; //Ветвь Да3
            else
                cout<< Pr <<endl; //Ветвь Нет3
    }
    else
        cout<< Pr <<endl; //Ветвь Нет1
    return 0;
}
```

*а*



*б*

**Рис. 7.9.** Программный код (*а*) и ГСА (*б*) для решения [примера 7.4](#) без использования логических операций && (И) и ||

Решение примера 7.4 с использованием логических операций значительно сокращает ГСА и программный код. Кроме этого исключаются повторные операции, которые записываются в ГСА и программном коде. Например, операция вывода суммы на [рис. 7.8, а](#) выполняется один раз, а на [рис. 7.9, а](#) – два раза. В случае использования представленного программного кода для решения дру-

гой похожей задачи и при использовании решения на [рис. 7.9, а](#) без логических операций необходимо будет выполнить корректирование двух строк с выводом результатов, что увеличивает шансы на ошибку пользователя. В связи с этим можно сделать вывод о том, что наиболее рациональным способом для решения подобных задач является использование логических операций, что позволяет исключить дублирование блоков ГСА и строк программного кода.

### 7.3. Оператор выбора *switch-case*

На практике нередкими оказываются случаи, когда для решения поставленной задачи требуется многократно использовать *if-else*-конструкции. Например, проверка принадлежности значения переменной к множеству заданных диапазонов.

Одна из подобных задач – это вывод текстовой константы («один», «два», «три» и т. д.) в соответствии с введенным числом (1, 2, 3 и т. д.).

Оператор *switch-case* – это удобная замена длинной *if-else*-конструкции, которая позволяет сравнивать значение переменной целого (или любого другого перечислимого) типа с несколькими предложенными вариантами.

Оператор выбора состоит из служебных слов *switch* и *case*. Синтаксис оператора *switch-case* показан в программном коде (см. [рис. 7.9](#)).

В скобках после оператора *switch* указывается переменная, значение которой сравнивается с перечисленными в строках после *case* константами.

Если значение переменной в *switch* совпадает с одной из констант *case*, то выполняется часть программного кода в этой же строке. Выполнение заканчивается ключевым словом *break*. Если ключевое слово *break* отсутствует в этой строке или блоке операторов в операторных скобках {}, то выполнение программы продолжается до окончания *switch-case*.

В случае, если не произошло совпадений ни с одной из указанных в строках *case* констант, выполняется программный код, записанный в строке *default*. Строка с ключевым словом *default* является необязательной.

Шаблон записи рассматриваемого оператора выбора представлен на [рис. 7.10](#).

*switch* (*x*)

{

case *A*: оператор 1; *break*;

case *B*: оператор 2; *break*;

case *C*: оператор 3.1;

оператор 3.2;

оператор 3.3;

*break*;

case *D*: оператор *M*; *break*;

default: оператор *N*; *break*;

}

*x* – переменная типа *int* или *char*

*A*, *B*, *C*, *D* – константы, например, 3 или “А”

с которыми сравнивается значение переменной *x*

Программный код одного *case* может быть записан в виде одной строки

или в виде нескольких строк

Строка *default* является необязательной

Рис. 7.10. Шаблон записи оператора выбора *switch-case*

Рассмотрим [пример 7.5](#), который выполняется с помощью оператора выбора *switch-case*. По введенному номеру месяца определить квартал года (1 – 3, 4 – 6, 7 – 9, 10 – 12) и вывести его в виде слова. На [рис. 7.11](#) представлен программный код решения. В примере пропущены строки, отвечающие за подключение библиотек, и заголовок основной функции (*\_tmain*).

```
int x; //объявление переменной x
```

```
cin>>x; //ввод значения переменной x (число от 1 до 12)
```

```
int n = (x+2)/3; // деление числа x + 2 на 3 и отбрасывание целой части,  
                // так как переменная типа int не имеет дробной части,  
                // то в результате в переменной n будет порядковый номер  
                // квартала
```

```
switch(n)
```

```
{
```

```
case 1: cout<< “Первый квартал” <<endl; break;
```

```
case 2: cout<< “Второй квартал” <<endl; break;
```

```
case 3: cout<< “Третий квартал” <<endl; break;
```

```
case 4: cout<< “Четвертый квартал” <<endl; break;
```

```
default: cout<< “Введите число от 1 до 12” <<endl; break;
```

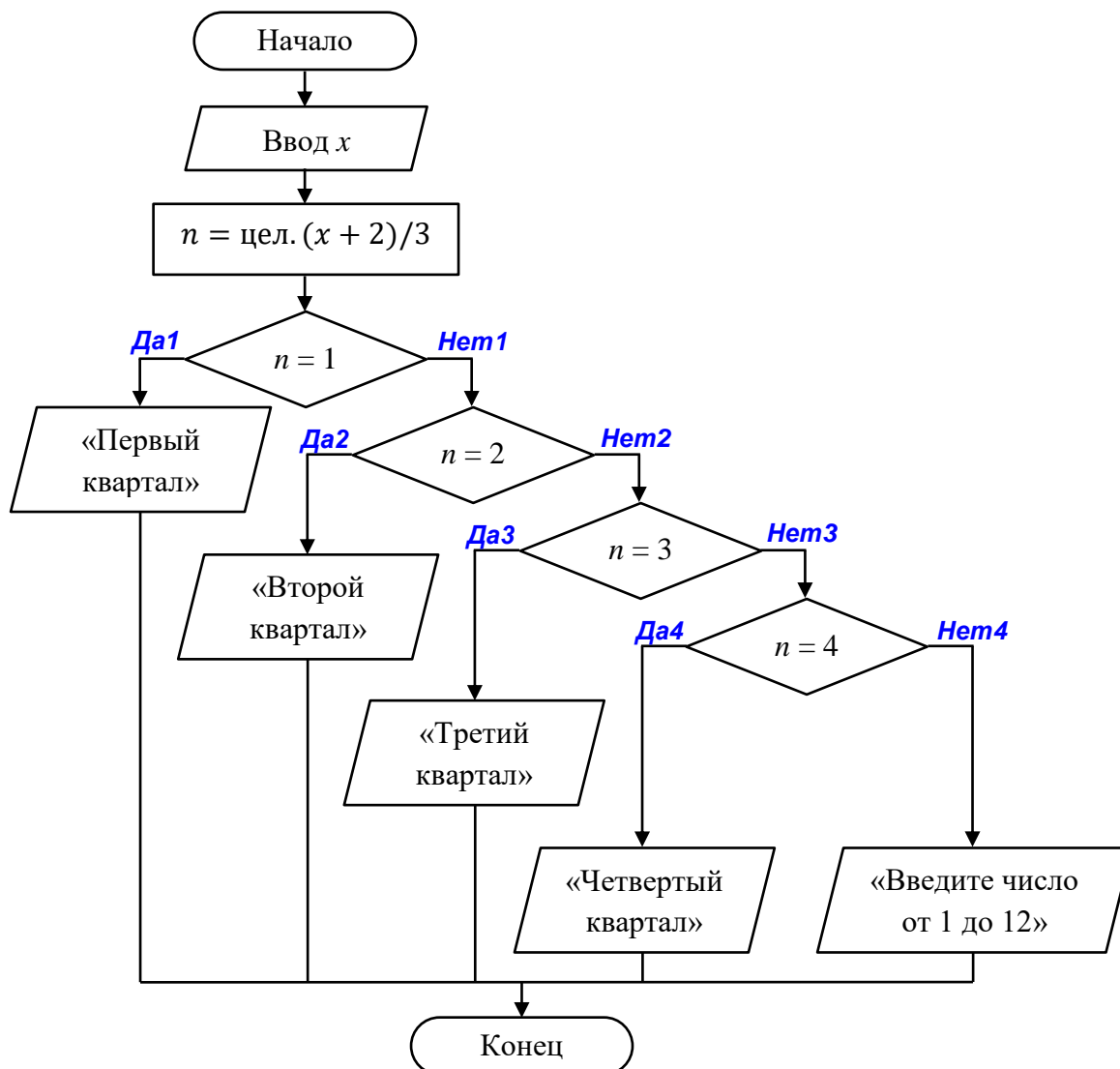
```
}
```

```
// строка default будет выполняться в том случае, когда n не будет
```

```
// находиться в предусмотренном диапазоне (1 – 4)
```

Рис. 7.11. Программный код решения примера 7.5

Графическая схема алгоритма для программного кода, содержащего оператор выбора *switch case*, содержит количество условных блоков (ромбов), соответствующее количеству операторов *case*. ГСА для [примера 7.5](#) представлена на [рис. 7.12](#).



[Рис. 7.12](#). Графическая схема алгоритма к [примеру 7.5](#)

## 8. ЦИКЛИЧЕСКИЕ ВЫЧИСЛИТЕЛЬНЫЕ ПРОЦЕССЫ

### 8.1. Понятие цикла

При решении некоторых задач приходится многократно выполнять одинаковые вычисления с разными исходными данными. Например, составить программу расчета значений функции  $y = \sin x$  при  $x = 0; 0,1; 0,2; \dots; 1$ . Для определения всех значений функции  $y$  необходимо вычислять и выводить 11 раз значение  $\sin x$ , начиная с  $x = 0$  и увеличивая каждый раз аргумент  $x$  на 0,1.

Данный пример является примером *табулирования* функции – определения значений функции при изменении ее аргумента от начального до конечного значения с определенным шагом (приращением). При решении подобных задач целесообразно использовать циклический алгоритм, реализуемый с помощью специальных операторов циклов.

*Цикл* – это конструкция в программировании, позволяющая неоднократно выполнять одну и ту же последовательность операторов программного кода.

Циклические алгоритмы делятся на арифметические и итерационные.

Цикл называется *арифметическим*, если количество его повторений известно или может быть легко вычислено (другие названия арифметического цикла – цикл с известным числом повторений, счетный, цикл с параметром).

Цикл называется *итерационным*, если число его повторений заранее не известно. Заканчивается такой цикл при выполнении или, наоборот, невыполнении некоторого условия.

Любой цикл содержит в себе элементы разветвления, поскольку в цикле всегда нужно проверять условие его окончания.

Для создания работоспособного цикла необходимо корректно задать следующее:

- параметр цикла (управляющую переменную);
- начальное и (или) конечное значения параметра цикла;
- закон изменения параметра цикла (*шаг*) либо параметра, влияющего на условие окончания цикла;
- тело цикла;
- условие окончания (выполнения) цикла.

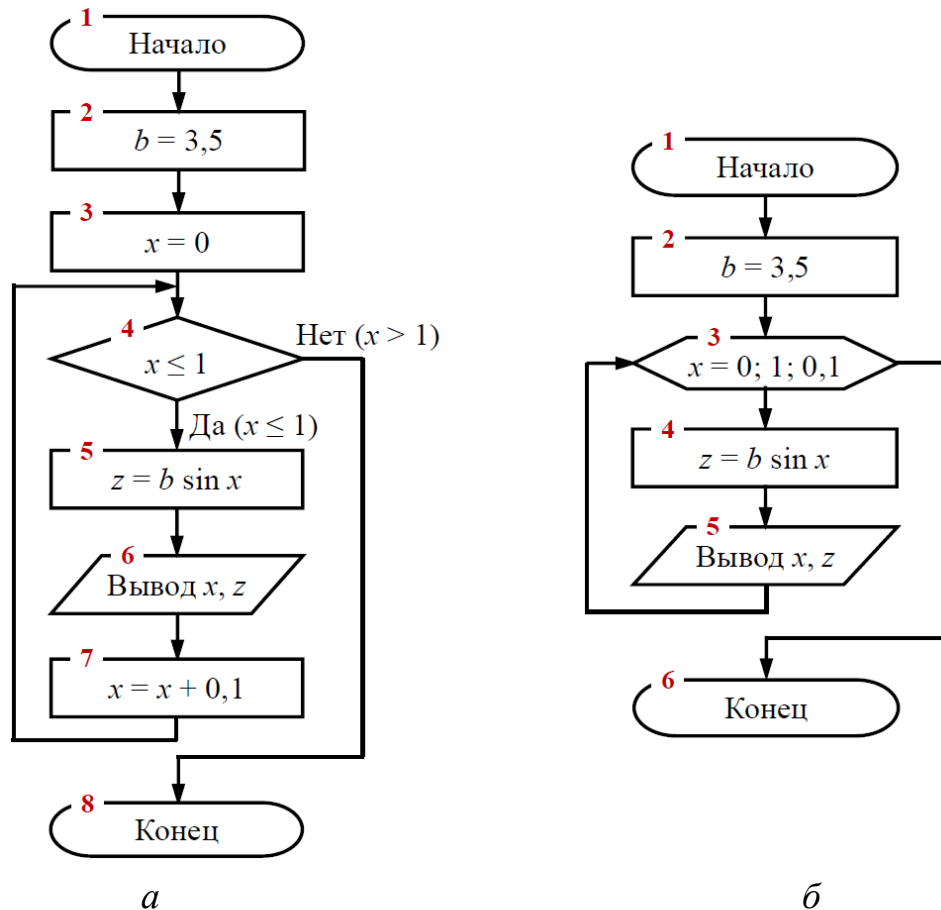
## 8.2. Арифметический цикл

Арифметический цикл предназначен для организации выполнения последовательности операторов заранее заданное число раз.

Рассмотрим реализацию арифметического цикла на [примере 8.1](#). Протабулировать функцию  $z = b \sin x$  в диапазоне изменения аргумента  $x$  от 0 до 1 с шагом  $\Delta x = 0,1$ , если  $b = 3,5$ .

Составим ГСА решения [примера 8.1](#) ([рис. 8.1, а](#)). В блоке 2 задается значение константы ( $b = 3,5$ ), в блоке 3 – начальное значение параметра цикла ( $x = 0$ ), в блоке 4 выполняется сравнение текущего значения параметра цикла с конечным ( $x \leq 1$ ). В случае истинности условия вычисляется  $z$  (блок 5) и выво-

дятся текущие значения  $x$  и  $z$  (блок 6). В блоке 7 значение аргумента  $x$  увеличивается на  $\Delta x$ , выполняется возврат к блоку 4, и весь процесс повторяется до тех пор, пока текущее значение  $x$  не превысит конечное (т. е. условие в блоке 4 перестанет быть истинным).



**Рис. 8.1.** ГСА решения [примера 8.1](#): в полной форме с применением условного блока 4 (а) и в краткой форме с помощью блока модификации 3 (б)

При решении в  $C++$  задач с применением арифметического цикла в ГСА его цикл обычно изображают в компактном виде ([рис. 8.1, б](#)) с использованием блока 3, в котором описывают закон изменения (модификации) параметра цикла (в данном примере  $x$ ), указывая его начальное, конечное значения и шаг изменения. Таким образом, блок 3, реализующий цикл в краткой форме ГСА, объединяет в себе блоки 3, 4 и 7 полной формы ГСА.

Для организации такого цикла в  $C++$  используется оператор *for*, который имеет следующий формат записи:

```
for (тип  $x = x_{\text{нач}}$ ;  $x \leq x_{\text{кон}}$ ;  $x += \Delta x$ ) //Заголовок (начало) цикла
{
    //Начало тела цикла
    ....;
    //Операторы (тело цикла)
}
//Окончание тела цикла
```

В этой записи в элементе «тип  $x = x_{\text{нач}}$ » указываются тип данных параметра цикла, его идентификатор (имя) и начальное значение. Аналогичная запись производится при инициализации переменной. Элемент « $x \leq x_{\text{кон}}$ » приведен для проверки условия выполнения цикла. Пока данное условие истинно, выполняется тело цикла. Элемент « $x += \Delta x$ » записывается для организации увеличения параметра цикла с шагом  $h_x$  для постепенного изменения его  $x_{\text{нач}}$  до  $x_{\text{кон}}$ .

Существуют особенности указания шага изменения параметра цикла:

- если  $x_{\text{нач}}$  меньше, чем  $x_{\text{кон}}$ , то шаг должен быть положительным;
- если  $x_{\text{нач}}$  больше, чем  $x_{\text{кон}}$ , то шаг должен быть отрицательным.

Значения  $x_{\text{нач}}$ ,  $x_{\text{кон}}$  и  $\Delta x$  могут быть константами, переменными или арифметическими выражениями.

В элементе «тип  $x = x_{\text{нач}}$ » тип данных можно не указывать, но тогда переменная должна быть объявлена заранее перед циклом. При использовании записи в заголовке цикла «тип  $x = x_{\text{нач}}$ » переменная будет объявлена только для тела цикла и за его пределами (ниже закрытой операторной скобки «}») будет считаться необъявленной.

При обработке дробных чисел может накапливаться погрешность вычислений, которая приводит к тому, что цикл завершится до достижения последнего значения параметра цикла. Для исключения такой ситуации рекомендуется увеличить конечное значение параметра цикла, например, на десятую долю шага, т. е. вместо  $x_{\text{кон}}$  принимать значение  $x_{\text{кон}} + \Delta x / 10$ .

В теле цикла записывается последовательность действий, которые повторяются для каждого значения параметра цикла. Этот перечень может включать в себя фрагменты линейной структуры, разветвления и (или) циклы, которые в этом случае являются вложенными. После завершения циклического процесса управление передается оператору, следующему за закрытой операторной скобкой «}».

Программный код решения [примера 8.1](#), составленный в соответствии с ГСА (см. [рис. 8.1, б](#)), с использованием оператора *for* приведен на [рис. 8.2](#). При этом в заголовке цикла вместо исходного конечного значения параметра цикла, равного 1, принято значение 1,01. В противном случае значение функции при  $x = 1$  не будет получено за счет погрешности вычислений.

Если необходимо выполнить подсчет количества повторений выполнения тела цикла (итераций) при определенных условиях, то используется цикл со

счетчиком. Счетчик – это целочисленная переменная, в которой накапливается количество итераций цикла, удовлетворяющих некоторому условию.

```
//Арифметический цикл (табуляция функции)
//Объявление переменных
//x – аргумент функции (параметр цикла), z – функция
double x, z;
const double b = 3.5; //Объявление константы b
for (x = 0; x <= 1.01; x += 0.1) //Заголовок цикла с параметром x
{ //Начало тела цикла
    z = b * sin(x); //Вычисление значения z
    cout << "x = " << x << ", z = " << z << endl; //Вывод значений x, z
} //Окончание тела цикла
```

Рис. 8.2. Программный код решения примера 8.1

#### 8.2.1. Цикл со счетчиком

Шаблон программного кода и ГСА для решения задач, связанных с подсчетом количества итераций, представлены на рис. 8.3.

В представленном шаблоне перед условием показан блок операторов, которые необходимы для решения задачи, например, ввод числа с помощью входного потока, формулы для вычислений и т. п.

Рассмотрим пример 8.2. Необходимо подсчитать количество нечетных чисел из вводимых с помощью клавиатуры десяти значений.

Для решения данной задачи необходимо организовать арифметический цикл, тело которого повторится 10 раз (10 итераций). Для этого используется переменная типа *int*, принимающая значения от 1 до 10. В теле цикла выполняется ввод числа с помощью входного потока. По условию задачи каждое введенное число проверяется на нечетность. Чтобы проверить истинность/ложность этого условия, его нужно разделить на два и сравнить остаток от деления с нулем. Если остаток от деления равен единице, то число является нечетным, если остаток от деления равен нулю – четным.

После проверки условия в случае его истинности значение счетчика увеличивается на единицу. После окончания цикла (выполнения всех 10 итераций) на экран с помощью выходного потока выводится итоговое значение счетчика.

Программный код и ГСА решения примера 8.2 представлены на рис. 8.4.

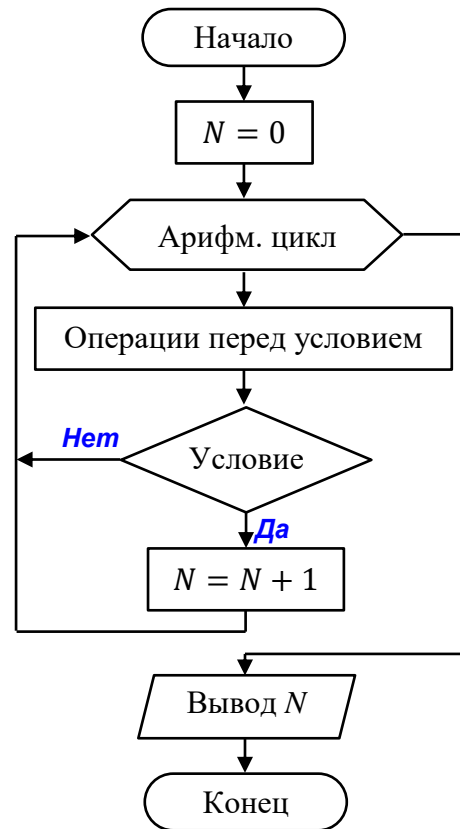


```

#include "stdafx.h"
#include <iostream>
#include <math.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[ ])
{
    int N=0; //Начальное значение счетчика
    for(...) //Задание арифметического цикла
    {
        ...; //Операторы перед условием
        if(...) //Условие для подсчета
            N++; //Прибавление
                //к счетчику единицы
    }
    cout<<N<<endl; //Вывод количества
    return 0;
}

```

*а*



*б*

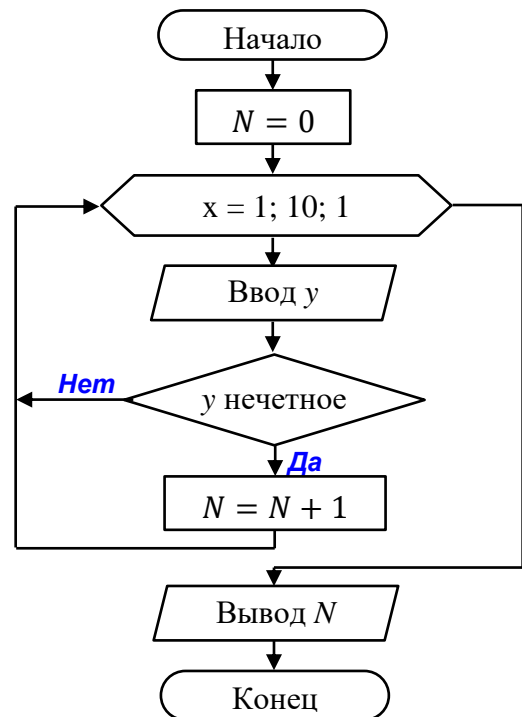
**Рис. 8.3.** Шаблон программного кода (*а*) и ГСА (*б*) для решения задач, связанных с подсчетом количества (счетчиком)

```

#include "stdafx.h"
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[ ])
{
    int x, y, N=0;
    for(x = 1; x <= 10; x++)
    {
        cin>>y;
        if(y%2 == 1)
            N++;
    }
    cout<<N<<endl;
    return 0;
}

```

*а*



*б*

**Рис. 8.4.** Шаблон программного кода (*а*) и ГСА (*б*) для решения [примера 8.2](#)

Переменная  $x$  в решении (см. [рис. 8.4](#)) является порядковым номером вводимого с помощью клавиатуры числа.

### 8.2.2. Вычисление максимума и минимума

Среди разных типов вычислительных задач особое место занимает определение экстремума – максимального и (или) минимального значения функции на рассматриваемом интервале изменения ее аргумента. Решение такой задачи основано на последовательном сравнении каждого предыдущего значения функции с текущим значением и выбором из сравниваемой пары наибольшего/наименьшего.

В программировании для осуществления таких действий организуют цикл, в который вложена разветвляющаяся структура, реализующая сравнение и выбор экстремального значения.

Рассмотрим [пример 8.3](#). Необходимо вычислить максимальное и минимальное значения функции  $z = b \sin^2 x^3$  в диапазоне изменения аргумента  $x$  от 2 до 5 с шагом  $\Delta x = 0,2$ , если  $b = 0,13$ .

Вычисление значений  $z$  в этой задаче аналогично решению [примера 8.1](#). Дополнительно вместе с расчетом функции  $z$  выполняется определение экстремумов функции (ее максимального и минимального значений). ГСА решения [примера 8.3](#) приведена на [рис. 8.5](#).

Процесс поиска значений экстремумов функции  $z$  заключается в следующем. Предусматриваются отдельные переменные (блок 3):

для хранения максимума –  $z_{max}$ , в качестве ее начального значения принимается любое число, которое заведомо меньше хотя бы одного из анализируемых значений функции (в идеальном случае это должно быть минимально возможное из области допустимых значений  $z$ , определяемых типом). Примем в качестве начального значения  $z_{max}$  очень малое число, например,  $-1E25$ ;

для хранения минимума –  $z_{min}$ , в качестве ее начального значения принимается любое число, большее хотя бы одного из анализируемых значений функции (максимально возможное из области допустимых значений  $z$ , определяемых типом). Примем в качестве начального значения  $z_{min}$  очень большое число, например,  $1E25$ .

После вычисления и вывода текущего значения  $z$  (блоки 5 и 6) выполняется его сравнение со значением  $z_{max}$  (блок 7). Если оно окажется больше, то переменной  $z_{max}$  присваивается вычисленное значение  $z$  (блок 8). Аналогично каждое текущее значение  $z$  сравнивается со значением  $z_{min}$  (блок 9), и если оно окажется меньше, то переменной  $z_{min}$  присваивается вычисленное на этом шаге значение  $z$  (блок 10).

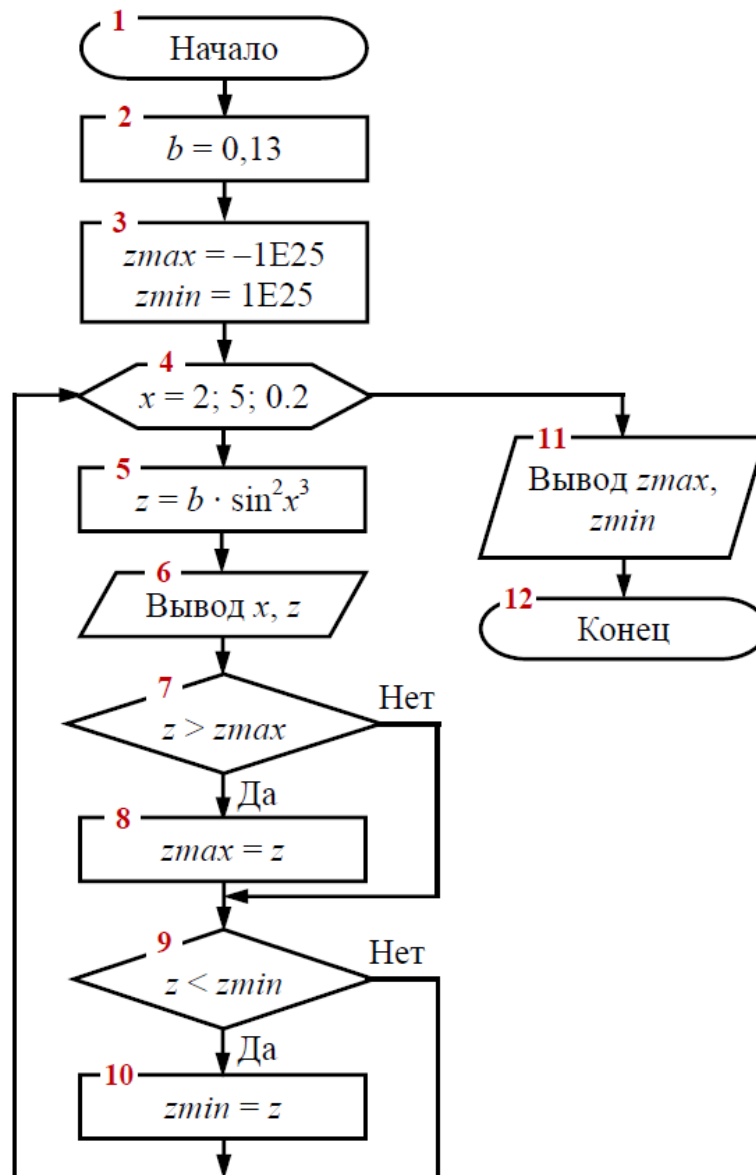


Рис. 8.5. ГСА решения примера 8.3

По окончании цикла, выполнения вычислений  $z$  и замены значений  $z_{max}$  и  $z_{min}$  они содержат экстремумы функции  $z$ , для вывода которых служит соответствующий блок в ГСА (блок 11).

Программный код решения примера 8.3, составленный в соответствии с ГСА (см. рис. 8.5), приведен на рис. 8.6.

```

double x, z, zmax, zmin;
const double b = 0.13;
zmax = -1e25;
zmin = 1e25;
for (x = 2; x <= 5; x += 0.2)
{
    z = b * pow(sin(pow(x,3)),2);
    cout<<"x = "<<x<<" , z = "<<z<<endl;
    if (z > zmax) //условие для нахождения максимума
        zmax = z;
    if (z < zmin) //условие для нахождения минимума
        zmin = z;
}
cout<<"zmax = "<<zmax<<" , zmin = "<<zmin<<endl;

```

**Рис. 8.6.** Программный код решения [примера 8.3](#)

Таким образом, можно сформулировать общий порядок вычисления максимума и (или) минимума:

- определить отдельную переменную для хранения значения, например:  
для максимума – *zmax*, для минимума – *zmin*;
- до цикла задать начальное значение:  
для максимума – *zmax* = -1E25, для минимума – *zmin* = 1E25;
- в цикле записать проверку условия и вычисление текущего значения:  
для максимума – *z* > *zmax*; для минимума – *z* < *zmin*;
- после соответствующей проверки в случае истинности условия заменить значение *zmax* или *zmin* на текущее значение *z*;
- после выхода из цикла вывести результирующее значение переменных *zmax* и *zmin*.

### 8.2.3. Цикл с разветвлением

При решении некоторых математических задач встречаются случаи, когда на рассматриваемом интервале изменения аргумента функции в зависимости от ограничений значения функции требуется определять по разным формулам. Решение подобных задач осуществляется с помощью организации цикла, в который вложена разветвляющаяся структура. Элементы данной структуры уже рассматривались в п. 8.2.1.

Рассмотрим [пример 8.4](#). Протабулировать разрывную функцию  $z(x)$ , изменяя аргумент  $x$  с шагом  $\Delta x = 0,2$ , если  $b = -4,5$ :

$$z = \begin{cases} b \sin x & \text{при } x < 3; \\ \sqrt{x} & \text{при } 3 \leq x \leq 5; \\ \cos x & \text{при } x > 5. \end{cases}$$

Заданная функция  $z$  состоит из трех частей и рассчитывается по разным формулам в зависимости от диапазона изменения аргумента  $x$ . Поскольку общий интервал изменения  $x$  в задании не указан, определим его исходя из равенства длин всех трех диапазонов.

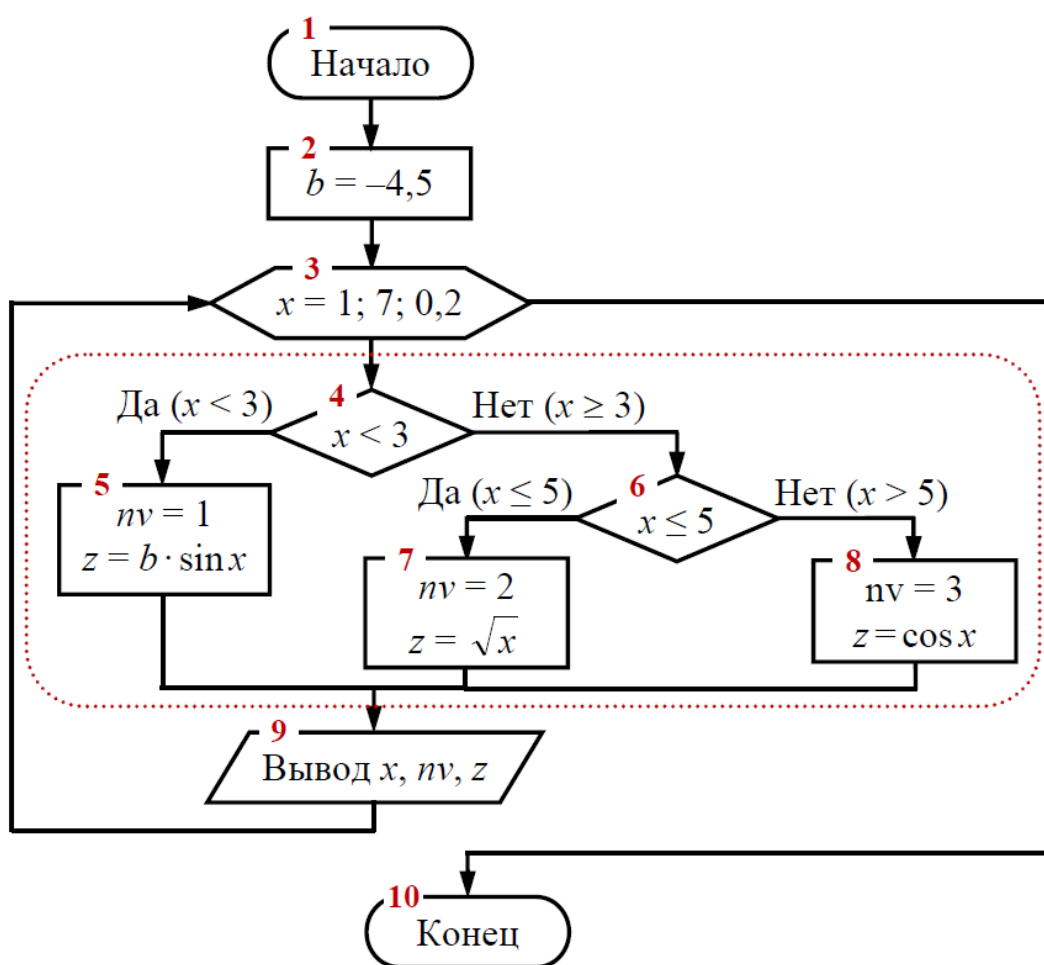


Рис. 8.7. ГСА решения [примера 8.4](#)

По условию задачи известны начальное и конечное значения среднего диапазона, что позволяет определить его длину:  $|5 - 3| = 2$ . Тогда начальное значение общего интервала изменения аргумента  $x$  равно  $3 - 2 = 1$ , а конечное значение общего интервала изменения аргумента  $x$  равно  $5 + 2 = 7$ .

Таким образом, примем общий интервал изменения  $x$  от 1 до 7 с шагом  $\Delta x = 0,2$ , что позволяет решить задачу аналогично примеру 8.1. Отличие заключается в том, что в теле цикла необходимо предусмотреть проверку условий изменения значения  $x$  с целью выбора формулы для расчета значения функции  $z$ .

ГСА решения [примера 8.4](#) приведена на [рис. 8.7](#), программный код – на [рис. 8.8](#). Пунктиром в ГСА и программном коде выделена структура, реализующая разветвление на три ветви.

```
double x, z;
int nv;
const double b = -4.5;
for (x = 1; x <= 7; x += 0.2)
{
    if (x < 3)           //условие 1
    {
        nv = 1;         //номер формулы – 1
        z = b * sin(x); //расчет значения
    }
    else
    {
        if (x <= 5)      //условие 2 без сравнения с числом 3,
        {               //так как оно уже учтено в условии 1
            nv = 2;      //номер формулы – 2
            z = pow(x, 1./2); //расчет значения
        }
        else
        {               //условия 3 нет
            nv = 3;      //номер формулы – 3
            z = cos(x);  //расчет значения
        }
    }
}
cout<<"x = "<<x<<" , nv = "<<nv<<" , z = "<<z<<endl;
}
```

**Рис. 8.8.** Программный код решения [примера 8.4](#)

### 8.3. Вычисление сумм и произведений

Задачи накопления сумм (математические выражения вида  $\sum_{i=n}^k f_i$ ) или накопления произведений ( $\prod_{i=n}^k f_i$ ) представляют собой большую группу вычислительных задач.

Нахождение суммы заключается в циклическом вычислении промежуточных сумм  $S_i$  посредством определения очередного слагаемого  $f_i$  и добавления его к значению суммы предыдущих слагаемых  $S_{i-1}$ . Таким образом, накопление суммы можно представить в виде последовательности однотипных вычислений (для наглядности примем  $n = 1$ ):

$$\begin{aligned} S_1 &= S_0 + f_1 = f_1; \\ S_2 &= S_1 + f_2 = (f_1) + f_2; \\ S_3 &= S_2 + f_3 = (f_1 + f_2) + f_3; \\ &\dots \\ S_k &= S_{k-1} + f_k = \sum_{i=1}^k f_i. \end{aligned} \quad (8.1)$$

Поскольку в памяти компьютера обычно не требуется сохранять значения всех слагаемых и промежуточных сумм, то их можно представить простыми переменными, т. е. хранить в памяти ЭВМ в одних и тех же ячейках. Тогда общая формула для накопления суммы будет иметь вид:

$$S = S + f_i. \quad (8.2)$$

Характерной особенностью формулы (8.2) является то, что и в правой, и в левой ее частях фигурирует одна и та же переменная  $S$ . Суть такой формулы заключается в том, что к вычисленному ранее промежуточному значению суммы  $S$  (в правой части формулы) прибавляется очередное слагаемое  $f_i$  и полученное текущее значение суммы присваивается этой же переменной  $S$  (в левой части). Чтобы обобщенную формулу (8.2) можно было использовать и для расчета первого промежуточного значения суммы (равного первому ее слагаемому), следует принять начальное значение суммы  $S$  равным нулю.

Рассмотрим **пример 8.5**. Вычислить  $S = \sum_{t=1}^4 (t \cdot \sin x)$  для произвольного  $x$ .

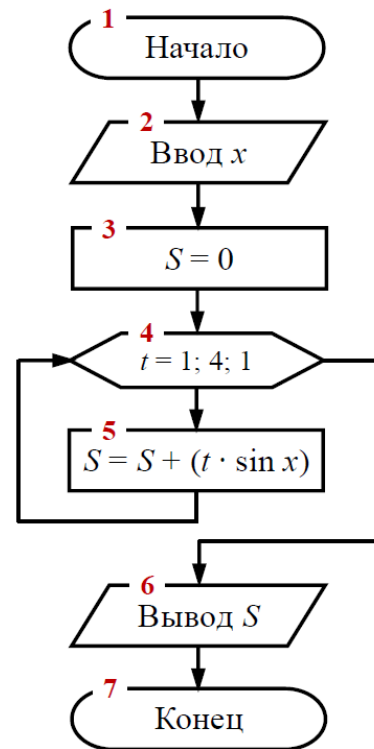
Общая формула накопления суммы для заданного выражения имеет вид:  $S = S + (t \cdot \sin x)$ . ГСА решения **примера 8.5** приведена на **рис. 8.9, б**. В блоке 3 в ГСА переменной  $S$  присваивается начальное нулевое значение. Затем организуется цикл (блок 4 в ГСА), в котором на каждом шаге к значению переменной  $S$ , полученному на предыдущем шаге, добавляется очередное слагаемое  $t \cdot \sin x$  (блок 5 в ГСА). После завершения цикла выводится результат накопления  $S$  (блок 6 в ГСА).

Программный код решения [примера 8.5](#), составленной в соответствии с ГСА (см. [рис. 8.9, б](#)), приведен на [рис. 8.9, а](#).

Аналогичным образом вычисляется произведение  $P = \prod_{i=n}^k f_i$ . Отличие от расчета суммы состоит лишь в том, что общая формула имеет вид  $P = P \cdot f_i$ , а начальное значение произведения, которое задается перед циклом, должно быть равно единице.

```
double x, S;
int t;
cin >> x; //Ввод значения x
S = 0; //Начальное значение суммы
for (t = 1; t <= 4; t++)
{
    S += t * sin(x); //Накопление суммы
}
cout << "S = " << S << endl;
```

*а*



*б*

**Рис. 8.9.** Решение [примера 8.5](#): программный код (*а*) и ГСА (*б*)

Рассмотрим [пример 8.6](#). Вычислить  $W = x^2 + \prod_{t=1}^4 (t \cdot \sin x)$  для произвольного  $x$ . Вывести результат пошагового накопления произведения.

Обозначим  $P = \prod_{t=1}^4 (t \cdot \sin x)$ , тогда  $W = x^2 + P$ . По аналогии с [примером 8.5](#) общая формула накопления произведения имеет вид:  $P = P \cdot (t \cdot \sin x)$ . ГСА решения [примера 8.6](#) приведена на [рис. 8.10, б](#). Отличие от [примера 8.5](#) состоит в том, что начальное значение  $P = 1$  (блок 2), на каждом шаге цикла значение переменной  $P$ , полученное на предыдущем шаге, умножается на  $t \cdot \sin x$  (блок 5) и выводится текущее промежуточное значение  $P$  (блок 6). После цикла вычисляется значение выражения  $W$  (блок 7) и выводится его результат (блок 8).



Программный код решения [примера 8.6](#), составленный в соответствии с ГСА (см. [рис. 8.10, б](#)), приведен на [рис. 8.10, а](#).

Таким образом, можно сформулировать общий порядок вычисления суммы  $S = \sum_{i=n}^k f_i$  или произведения  $P = \prod_{i=n}^k f_i$ :

– задать перед циклом начальное значение накапливаемой переменной.

Если не указано иначе, то в качестве него обычно принимают:

при накоплении суммы:  $S = 0$ ;

при накоплении произведения:  $P = 1$ ;

при вычислении количества (счетчик):  $N = 0$ ;

– организовать арифметический цикл;

– записать в теле цикла закон изменения переменной:

при накоплении суммы:  $S += f_i$ ;

при накоплении произведения:  $P *= f_i$ ;

при вычислении количества:  $N++$ ;

– вывести после цикла итоговое значение накапливаемой переменной.

*double* x, P, W;

*int* t;

*cin*>>x;

*P* = 1; //Начальное значение произведения

*for* (*t* = 1; *t* <= 4; *t*++)

{

*P* \*= *t* \* *sin*(*x*); //Накопление произведения

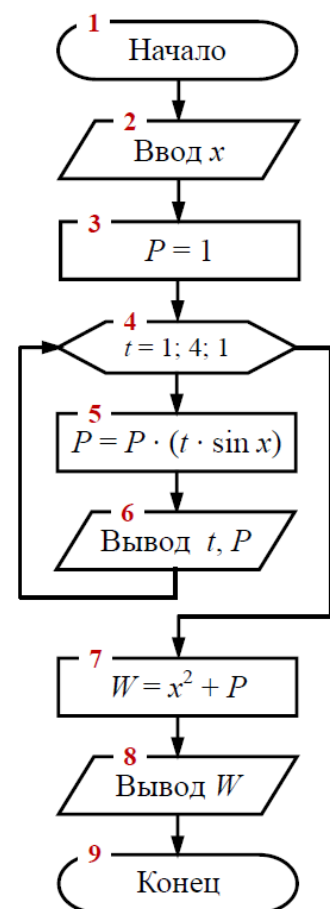
*cout*<<"*t* = "<<*t*<<" , *P* = "<<*P*<<*endl*;

}

*W* = *pow*(*x*,2) + *P*; //Вычисление значения *W*

*cout*<<"*W* = "<<*W*<<*endl*;

*а*



*б*

**Рис. 8.10.** Решение [примера 8.6](#): программный код (*а*) и ГСА (*б*)

## 8.4. Итерационные циклы

Если число повторений цикла заранее не известно, а решение о его завершении принимается на основе анализа заданного условия, то такой повторяющийся вычислительный процесс называется итерационным циклом.

Итерационные циклы в C++ организуются с помощью операторов *while* или *do ... while*.

### 8.4.1. Оператор цикла *while*

Шаблон программного кода и ГСА для организации итерационного цикла с помощью оператора *while* представлены на [рис. 8.13](#). Данный вид цикла называется *циклом с предусловием*.



**Рис. 8.11.** Шаблон оператора *while* для реализации цикла с предусловием: программный код (*а*) и ГСА (*б*)

Когда условие, записанное справа в скобках от ключевого слова *while*, перестанет быть истинным, осуществляется выход из цикла, т. е. переход к оператору, следующему за закрытой операторной скобкой «*}*».

В ГСА итерационный цикл изображается только в полной форме.

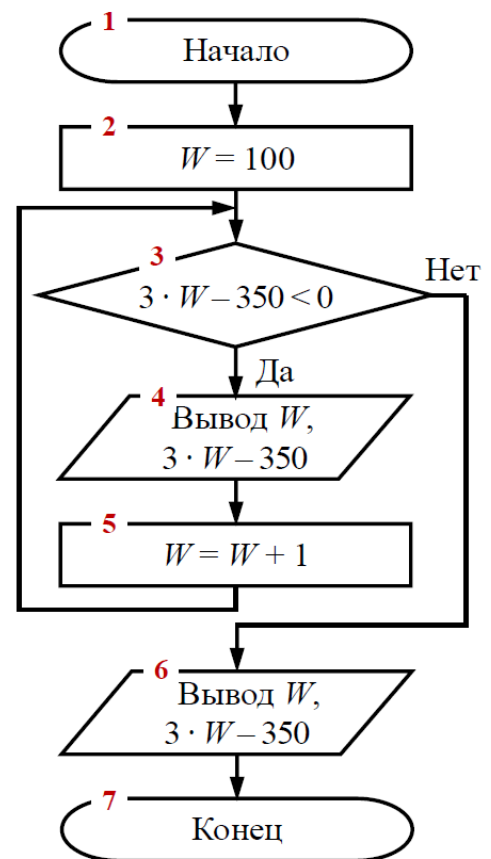
Рассмотрим [пример 8.7](#). Найти наименьшее положительное целое трехзначное число  $W$ , при котором значение выражения  $3 \cdot W - 350$  становится неотрицательным числом. ГСА решения [примера 8.7](#) приведена на [рис. 8.12, б](#). Так как в задаче требуется найти трехзначное число, то начальное значение  $W$  принимается равным 100 (блок 2). Пока значение выражения  $3 \cdot W - 350$  отри-

цательно (т. е. меньше нуля, блок 3), выводится результат вычисления выражения (блок 4) и определяется следующее число  $W$  (блок 5). Действия повторяются до тех пор, пока значение выражения  $3 \cdot W - 350$  станет неотрицательным (т. е. больше нуля или равно нулю), а значит, искомое число  $W$  найдено (блок 6).

Программный код решения [примера 8.7](#), составленный в соответствии с ГСА (см. [рис. 8.12, б](#)), приведен на [рис. 8.12, а](#).

```
int W;
W = 100; //Начальное значение W
while (3 * W - 350 < 0)
{
    //Вывод текущего числа и значения выражения
    cout<<"W = "<<W;
    cout<<", 3 * W - 350 = "<<3 * W - 350<<endl;
    W++; //получение следующего значения W
}
//Вывод найденного числа и значения выражения
cout<<"W = "<<W;
cout<<", 3 * W - 350 = "<<3 * W - 350<<endl;
```

*а*



*б*

**Рис. 8.12.** Решение [примера 8.7](#): программный код (*а*) и ГСА (*б*)

#### 8.4.2. Оператор цикла *do...while*

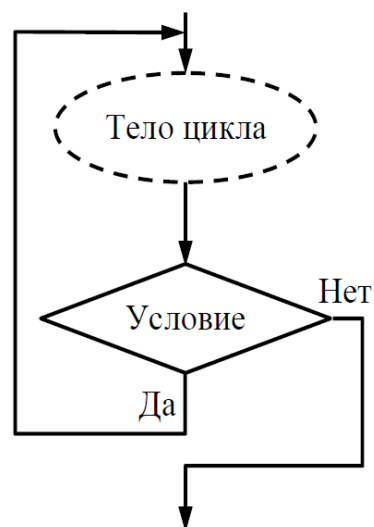
Другим способом реализации итерационного цикла является использование оператора *do...while*. Шаблон программного кода и ГСА для организации итерационного цикла с помощью оператора *do...while* представлены на [рис. 8.13](#). Данный вид цикла называется *циклом с постусловием*.

Форма записи оператора *do...while* полностью аналогична рассмотренному выше оператору *while*. В данном случае вместо ключевого слова *while* за-

писывается *do*, а после тела цикла записывается *while* вместе с условием. В данном случае в строке с оператором *while* в конце ставится точка с запятой.

```
do
{
...; //Операторы, которые выполняются,
...; //пока Условие истинно (тело цикла)
}
while (Условие);
```

*а*



*б*

**Рис. 8.13.** Шаблон оператора *do...while* для реализации цикла с постусловием: программный код (*а*) и ГСА (*б*)

Основным отличием цикла с постусловием от цикла с предусловием заключается в том, что тело цикла выполнится хотя бы один раз, даже если условие в скобках у *while* ложно. В случае цикла с предусловием при ложности условия тело цикла не выполнится ни разу.

## 8.5. Вложенные циклы

Если в теле цикла (внешнего) организовать еще один цикл (внутренний), то такой цикл будет называться вложенным. При одном значении управляющей переменной внешнего цикла во вложенном (внутреннем) управляющая переменная принимает все свои заданные значения (от начального до конечного).

Рассмотрим [пример 8.8](#). Необходимо рассчитать и вывести на экран таблицу умножения.

Для решения задачи используем две переменные целого типа *i* (номер строки) и *j* (номер столбца), которые могут принимать значения от 1 до 9.

На первой итерации внешнего цикла переменная *i* примет значение 1, а во вложенном цикле переменная *j* будет принимать значения от 1 до 9. Путем перемножения будут получены числа для первой строки таблицы умножения. Затем во внешнем цикле переменная *i* примет значение 2 и вложенный цикл

повторит свою последовательность, сформировав вторую строку таблицы умножения.

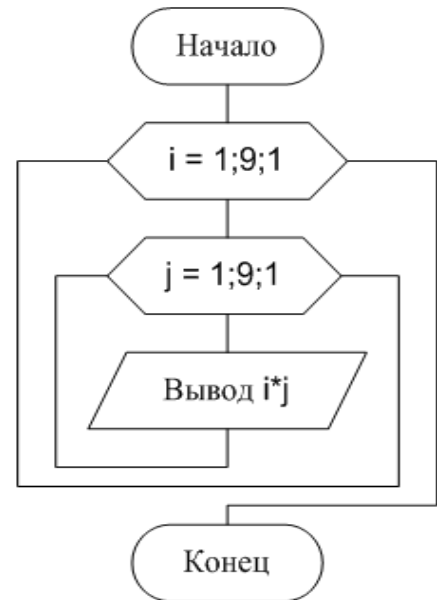
Программный код и ГСА для решения данного примера представлены на рис. 8.14.

```

int i, j;
//Внешний цикл
for (i = 1; i <= 9; i++)
{
    //Вложенный цикл
    for (j = 1; j <= 9; j++)
    {
        cout<< i * j<< '\t';
    }
    cout<<endl; //Переход на новую строку
}

```

*a*



*б*

Рис. 8.14. Решение примера 8.8: программный код (a) и ГСА (б)

Результат выполнения представленного на рис. 8.14 программного кода представлен на рис. 8.15.

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Для продолжения нажмите любую клавишу . . .

Рис. 8.15. Результат выполнения программного кода, представленного на рис. 8.14

## 9. ФУНКЦИИ В C++

Очень часто в программировании необходимо выполнять одни и те же действия, изменяя при этом только входные значения. Это увеличивает размер программного кода, затрудняет его чтение и повышает возможность возникновения ошибок.

Одним из способов минимизации перечисленных сложностей является разделение программного кода и оформление его части в виде особо оформленных самостоятельных программ (функций). В C++ реализован механизм создания программистом собственных функций, вызов которых может осуществляться многократно из разных точек программного кода.

Использование функций позволяет выполнять отладку программы по частям (например, коллективом программистов), уменьшает время перекомпиляции, облегчает процесс отладки программы.

### 9.1. Понятие функции

В C++ различают два вида функций: пользовательские и встроенные. Встроенные функции были рассмотрены в предыдущих разделах данного пособия, поэтому далее будут рассмотрены пользовательские функции, разрабатываемые самостоятельно программистом и являющиеся неотъемлемой частью программного кода при решении задач.

*Функция* – это именованная последовательность входных данных и операторов, выполняющая заданные программистом действия и возвращающая результат своей работы в виде значений или действий (например, заливка экрана цветом). Обращение (вызов) к функции может выполняться из любого места программы, вызов осуществляется при указании имени функции. Вызов функции в программном коде может быть выполнен неограниченное число раз.

Функции в C++ могут быть реализованы в виде процедур-*функций* или процедур-*подпрограмм*. Различия между процедурами-*функциями* и процедурами-*подпрограммами* заключаются в следующем:

- 1) процедура-*функция*, как правило, возвращает результат (значение). Например, возвращение значения математической функцией  $\cos(x)$ ;
- 2) процедура-*подпрограмма* возвращает действие. Например, вывод на экран элементов псевдографики, изменение цвета экрана и т. п.

### 9.1.1. Создание функции

Использование функции в программном коде требует ее объявления и определения.

Объявление функции сообщает компилятору *имя функции*, ее *тип* (определяет тип возвращаемого функцией значения) и *параметры* (количество и тип передаваемых в функцию значений). Все эти компоненты составляют заголовок функции.

После заголовка в операторных скобках `{ }` указывается тело функции – правильная последовательность операторов для решения поставленной задачи по требуемому алгоритму. Перед закрывающей операторной скобкой `}` указывается ключевое слово *return*. После *return* указывается идентификатор, содержащий значение, вычисленное в данной функции, или выражение.

Синтаксис определения функции приведен на [рис. 9.1](#).

Тип функции (тип возвращаемого результата)	Имя функции	Типы и имена аргументов
<i>double</i>	<i>SomeFunction</i>	<i>(int z, double d)</i>
<pre>{     Программный код;     return &lt;выражение&gt;; }</pre>		

[Рис. 9.1](#). Синтаксис определения функции

На [рис. 9.1](#) показаны:

*возвращаемый тип* – любой из доступных типов C++, представляет собой тип результата;

*имя функции* – правильный идентификатор C++;

*тип аргумента* – тип передаваемых функции аргументов (*параметров*);

*имя аргумента* – правильный идентификатор C++.

*Тело функции* – последовательность объявлений типов идентификаторов и операторов, описывающих алгоритм решения задачи.

Важным оператором тела функции является оператор *return <выражение>*. Оператор *return* обеспечивает немедленный возврат в вызывающую функцию (в строку, следующую за строкой вызова), а также используется для передачи значения, вычисленного функцией в вызывающую программу.

В теле функции может быть несколько операторов *return*, но в некоторых случаях он вообще может не указываться. В последнем случае возврат в вызы-

вающую программу происходит после выполнения последнего оператора тела функции.

### 9.1.2. Определение функции и ее вызов

Существует несколько способов описания функций. Один из способов позволяет описать функцию в любой части программного кода:

```
int _tmain(int argc, _TCHAR* argv[ ])
{
    ...
    SomeFunction (1, 2) //вызов функции по ее имени
    ...
    double SomeFunction (int z, double d) /*заголовок функции и ее тело
        {
            ...
        }
    ...
    return 0;
}
```

Рассмотрим этот же пример, но с использованием *прототипа* функции, т. е. с описанием заголовка функции и аргументов. Прототип указывается в разделе объявлений (см. подразд. 5.2), т. е. перед функцией *\_tmain* (в конце прототипа ставится «;»):

```
double SomeFunction (int z, double d);
int _tmain(int argc, _TCHAR* argv[ ])
{
    ...
    SomeFunction (1, 2) //вызов функции по ее имени
    ...
    return 0;
}
double SomeFunction (int z, double d) /*заголовок функции и ее тело описаны
после операторных скобок функции _tmain*/
{
    ...
}
```

Функция, возвращающая значение, является выражением, поэтому с ней можно обращаться так же, как и с любым другим выражением, например:

$y = \text{SomeFunction} (1, 2).$



## 9.2. Области видимости переменных в программе

Каждая переменная в программном коде обладает *областью видимости*, т. е. областью программного кода, где к ней возможно обращение. Переменные принадлежат тому блоку кода, в котором они объявлены.

По области видимости различают *глобальные* и *локальные* переменные. *Глобальные* переменные доступны во всем программном коде. *Локальные* переменные объявляются в теле функции и доступны только в ее теле. По завершении работы функции локальные переменные удаляются из памяти.

Пример объявления глобальных и локальных переменных показан в следующем фрагменте программного кода:

```
...
int _tmain(int argc, _TCHAR* argv[ ])
{
    double x, y; //глобальные переменные x и y
    ...
    double SomeFunction (int z, double d) //заголовок функции
    {
        int z, double d; //локальные переменные z и d
        ...
        return z;
    }
    cout<<x<<y; //вывод на печать глобальных переменных
    cout<<z<<d; /*НЕВЕРНО!!! локальные переменные z и d вне функции не
существуют*/
    return 0;
}
```

Таким образом, возможно объявление переменной с одним именем как глобальной, так и локальной:

```
...
int _tmain(int argc, _TCHAR* argv[ ])
{
    double a=2.65; //глобальная переменная a
    ...
    double S_Func (int a) //заголовок функции
    {
        int a=567; //локальная переменная a
        ...
        return 0;
    }
}
```

```

    cout<<a; //вывод на печать глобальной переменной a
    return 0;
}

```

На экран будет выведено значение 2.65, значение  $a = 567$  доступно только внутри функции *S\_Func* и удалено из памяти после окончания работы функции *S\_Func*.

### 9.3. Возвращаемое значение

Механизм возврата из функции значения в вызвавшую ее главную функцию реализуется оператором *return <выражение>*. В качестве выражения могут быть использованы константы, идентификаторы (локальные) или выражения:

```

return 5; //возврат функцией константы
return z; //возврат функцией значения идентификатора
return (x>10); //возврат функцией логического значения ИСТИНА или ЛОЖЬ.

```

Допустимо использование нескольких операторов *return* в функции:

```

double Accumulation (double a, double b) //заголовок функции
{
    cin>>a>>b;
    ...
    if (a>0)
        return a;
    else
        return b;
}

```

Значение выражения (идентификатора), указанного после *return*, преобразуется к типу функции и передается в точку программного кода, следующую за вызовом функции.

### 9.4. Аргументы функции

*Аргументами* функции называются значения, передаваемые функции во время ее вызова. Количество и типы аргументов определяются требованиями к передаваемым значениям и выполняемым внутри функции действиям. Аргументы функции позволяют установить с ней двустороннюю связь – через передаваемые параметры и возвращаемые значения.

В C++ различают аргументы функций двух типов: *фактические* и *формальные*. При указании аргументов необходимо указать *тип* каждого из них. Функция может иметь несколько аргументов, которые разделяются запятой.

#### 9.4.1. Фактические аргументы

Фактические аргументы – значения глобальных переменных, определенных в функции *\_tmain*, передаваемые функции при ее вызове. Например:

```
double x = 10.1;
```

```
double y = MyFunction(x);
```

В качестве фактического аргумента могут использоваться:

- значения (числа, строки символов);
- идентификаторы (переменные, константы);
- арифметические выражения (включая встроенные функции).

При вызове функции должны использоваться только фактические аргументы.

#### 9.4.2. Формальные аргументы

Формальные аргументы (как и фактические) указываются в заголовке функции, используются в теле функции, а также в ее прототипе.

Типы формальных аргументов (ФА) должны совпадать с типом фактических аргументов. Значения формальных аргументов не задаются, ФА используются для описания операций в теле функции и для возврата значения.

Пример процедуры-функции с формальным аргументом, возвращающей действие:

```
int print (int j); //прототип функции с формальным аргументом  
int _tmain(int argc, _TCHAR* argv[ ])  
{  
int a=265; //глобальная переменная a  
int PRINT = print (a) //вызов функции с фактическим аргументом  
return 0;  
}  
int print (int j) // заголовок функции с формальным аргументом  
{  
cout<<"j="<<j<<endl;  
return 0;  
}
```

Пример функции-подпрограммы с формальными аргументами, которая возвращает вещественное значение:

```

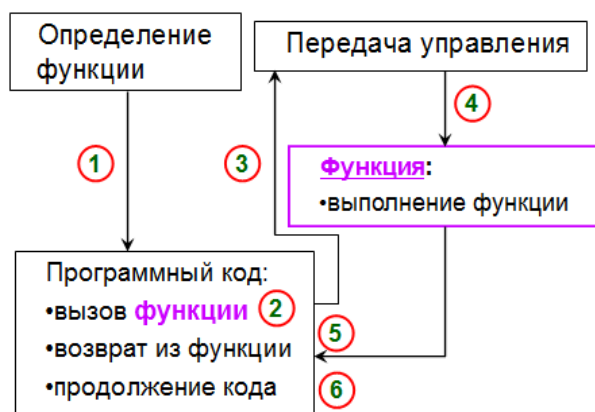
double summa (double a, double b); //прототип функции с формальными
аргументами
int _tmain(int argc, _TCHAR* argv[ ])
{
    double x=1.87; //глобальная переменная x
    double y=2.99; //глобальная переменная y
    double SUM = summa (x, y) /*вызов функции с фактическими аргу-
        ментами*/
    ...
    return 0;
}
double summa (double a, double b) /*заголовок функции с формальными ар-
гументами*/
{
    double c = a + b;
    return c; //возврат результата вычислений
}

```

## 9.5. Алгоритм вызова функции

На [рис. 9.2](#) приведен алгоритм вызова функции:

- 1 – в программном коде определяется функция – задаются ее тип, имя, аргументы (и их типы) и набор операторов;
- 2 – из функции `_tmain` (или любой другой функции) производится вызов функции с передачей в нее фактических аргументов;
- 3 – управление передается непосредственно вызванной функции;
- 4 – осуществляется выполнение вызванной функции с фактическими аргументами;



**Рис. 9.2.** Алгоритм вызова функции

5 – выполняется возврат из функции значения, определенного в операторе *return*;

6 – продолжается выполнение остального программного кода, следующего за вызовом функции на шаге 2.

## 9.6. Примеры использования функций

**Пример Ф1.** Создать функцию (процедуру-подпрограмму) *privet()*, которая будет выводить фразу «*Hello, world!*». Функция не содержит аргументов, т. е. при ее вызове будут выполнены только действия, описанные в функции:

```
int privet() //объявление функции
{
    cout << " Hello, world!";
    return 0;
}
int _tmain(int argc, _TCHAR* argv[ ])
{
    ...
    privet(); //вызов функции
    ...
    return 0;
}
```

**Пример Ф2.** Вычислить значение выражения *w* по приведенной ниже формуле:

$$w = \sqrt{x^2 + y^2 + \sin^2 xy} + \sqrt{y^2 + z^2 + \sin^2 yz} + \sqrt{z^2 + x^2 + \sin^2 zx}.$$

При анализе выражения очевидно, что необходимо вычислять значения трех слагаемых, которые отличаются только аргументами. Запишем одно из слагаемых в общем виде и определим функцию с формальными параметрами:

$$F(a, b) = \sqrt{a^2 + b^2 + \sin^2 ab}.$$

В результате получим итоговое выражение:

$$w = F(x, y) + F(y, z) + F(z, x).$$

При объявлении функции первым способом программный код имеет вид:

```
double F(double a, double b) // заголовок функции
{
    double w = sqrt(pow(a, 2)+pow(b, 2)+pow(sin(a*b), 2));
    return w;
}
int _tmain(int argc, _TCHAR* argv[ ])
{
    double x, y, z;
    cin>>x>>y>>z;
    double w1 = F(x, y) + F(y, z) + F(z, x); /*вызов функции
    с фактическими аргументами*/
    cout<<w1<<endl;
    return 0;
}
```

Если использовать способ с объявлением прототипа, то программный код будет таким:

```
.....
double F(double a, double b); // прототип функции с формальными аргументами
int _tmain(int argc, _TCHAR* argv[ ])
{
    double x, y, z;
    cin>>x>>y>>z;
    double w1 = F(x, y) + F(y, z) + F(z, x);
    cout<<w1<<endl;
    return 0;
}

double F(double a, double b)
{
    double w = sqrt(pow(a, 2)+pow(b, 2)+pow(sin(a*b), 2));
    return w;
}
```

**Пример Ф3.** Вычислить значение функции у:

$$y = e^{z^2+z^4} - e^{z^1+z^3},$$

где  $z_1, z_2$  – действительные корни квадратного уравнения  $az^2 + bz + c = 0$ ;

$z_3, z_4$  – действительные корни квадратного уравнения  $wz^2 + nz + m = 0$ .

В данной задаче требуется вычислить корни квадратного уравнения, поэтому создадим две функции – *koren1* и *koren2*, каждая из которых позволит определить наибольшее (*koren1*) и наименьшее (*koren2*) значения корней.

Программный код:

```
double koren1 (double p, double r, double q) /*уравнение  $px^2 + rx + q = 0$ ,
                                              функция вычисления большего корня*/
{
    double x1;
    double d = pow(r,2) - 4*p*q; //расчет дискриминанта
    if (d < 0)
    {
        cout<< "net korney";
        return 0;
    }
    else
    {
        x1 = (- r + sqrt(d)) / (2 * p);
        return x1;
    }
}

double koren2 (double p, double r, double q) //уравнение  $px^2 + rx + q = 0$ , функция
                                              // вычисления меньшего корня
{
    double x2;
    double d = pow(r,2) - 4*p*q; //расчет дискриминанта
    if (d < 0)
    {
        cout<< "net korney";
        return 0;
    }
    else
    {
        x2 = (- r - sqrt(d)) / (2 * p);
        return x2;
    }
}

int _tmain(int argc, _TCHAR* argv[ ])
{
    double y, z1, z2, z3, z4, a, b, c, w, n, m;
```

```

cin >> a >> b >> c; // ввод коэффициентов первого уравнения;
cin >> w >> n >> m; // ввод коэффициентов второго уравнения;
z1 = koren1(a, b, c); //вызов функции для вычисления большего корня
                        //уравнения  $az^2 + bz + c = 0$ 
z2 = koren2(a, b, c); //вызов функции для вычисления меньшего корня
                        //уравнения  $az^2 + bz + c = 0$ 
z3 = koren1(w, n, m); //вызов функции для вычисления большего корня
                        //уравнения  $wz^2 + nz + m = 0$ 
z4 = koren2(w, n, m); // вызов функции для вычисления меньшего корня
                        // уравнения  $wz^2 + nz + m = 0$ 
y = exp(z2 + z4) + exp(z1 + z3); // расчет значения y
cout << "y=" << y;
return 0;
}

```

**Пример Ф4.** Вычислить значение функции  $z$ :

$$Z = \frac{\arcsin a^2 - \sin ab}{2 \arcsin \sqrt{b + 3,2}},$$

где  $a = 0,123$ ;  $b = -2,4$ .

В данном случае в функции целесообразно вычислять повторяющуюся часть выражения – арксинус.

Программный код:

```

double R (double x); // прототип функции с формальным аргументом
int _tmain(int argc, _TCHAR* argv[ ])
{
    double a = 0.123, b = -2.4, c, d;
    c = pow(a,2);
    d = sqrt(b+3.2);
    double z = (R(c) - sin(a*b))/(2*R(d)); //вызов функции с фактическими
                                           //аргументами

    cout<<z<<endl;
    return 0;
}
double R (double x) // определение функции
{
    double s = asin(x);
    return s;
}

```



## 10. МАССИВЫ

### 10.1. Понятие массива и его инициализация

При использовании в программе переменных для каждой из них выделяется соответствующая область памяти (ячейка), предназначенная для хранения в ней данных. Каждая ячейка с объявленным именем может одновременно хранить только одно значение.

Многие языки программирования позволяют под одним именем определить некоторое количество ячеек памяти, доступ к каждой из которых осуществляется по порядковому номеру ячейки. C++ не является исключением и тоже позволяет выполнять операции с группами ячеек памяти, объединенными одним именем. Ограничением является использование в этих ячейках данных только одинакового для них типа. Такая структура называется *массивом* данных, а ячейка – элементом массива.

*Массив* – упорядоченный набор данных одного типа, имеющих общее имя. Доступ к любому элементу массива осуществляется с помощью *индекса* (порядкового номера) элемента.

Для того чтобы объявить массив, необходимо указать его тип (аналогично любому идентификатору) и указать количество элементов (размер массива). Пример объявления вещественного массива из десяти элементов:

```
float Mass[10].
```

Порядковые номера элементов массива начинаются с нуля. Это является особенностью языка C++, изменить эту нумерацию нельзя. В качестве индексов элементов массива могут использоваться только положительные целые значения (или значения любого перечислимого типа). Для объявленного выше массива доступ к любому его элементу можно получить при указании его индекса, например, первый его элемент – *Mass[0]*.

Еще один вариант обращения к элементу объявленного массива:

```
int i = 5; //инициализация переменной целого типа
```

```
Mass[i]; //обращение к элементу массива с порядковым номером 5.
```

Объявленный массив необходимо инициализировать, т. е. присвоить значения каждому элементу. Одним из вариантов инициализации массива является использование арифметического цикла.

Например, объявить вещественный массив из десяти элементов, значения ввести с клавиатуры:

```
double x[10]; //объявление вещественного массива из 10 элементов
for (int i=0; i<10; i++) //цикл, значение параметра которого используется
    //в качестве индексов элементов массива
    //значения i изменяются от 0 до 9
{
    cin>>x[i]; //ввод значений элементов с клавиатуры
    cout<<x[i]; //вывод значений элементов
}
```

Еще один способ инициализации элементов массива:

```
double y[5] = {8.6, 5.2, 9.1, 4.9, 5.1}.
```

Значения указываются в фигурных скобках после символа присваивания, перечисление значений выполняется через запятую.

Возможна инициализация массива без указания его размера:

```
int x[] = {2, 6, 9, 4}.
```

В этом случае компилятор самостоятельно определяет размер массива в соответствии с количеством перечисленных в { } значений. Однако такой способ инициализации затрудняет последующие действия при работе с элементами массива, так как явно не задано их количество.

## 10.2. Примеры решения задач по обработке массивов

Обработка массивов осуществляется *поэлементно*. При использовании в программном коде нескольких массивов одинакового размера наиболее рациональным способом является инициализация каждого в отдельном цикле. Это позволяет снизить количество ошибок при вводе значений.

**Пример М1.** Вычислить среднее арифметическое элементов вещественного массива  $x[i]$ .

При определении среднего арифметического элементов массива  $x$  необходимо создать цикл накопления суммы значений элементов и разделить ее на их количество (рис. 10.1).

**Пример М2.** Сформировать одномерный массив  $B$ , элементы которого связаны с элементами массива  $A$  зависимостью

$$b_i = \frac{\sin a_i}{4 + i}.$$

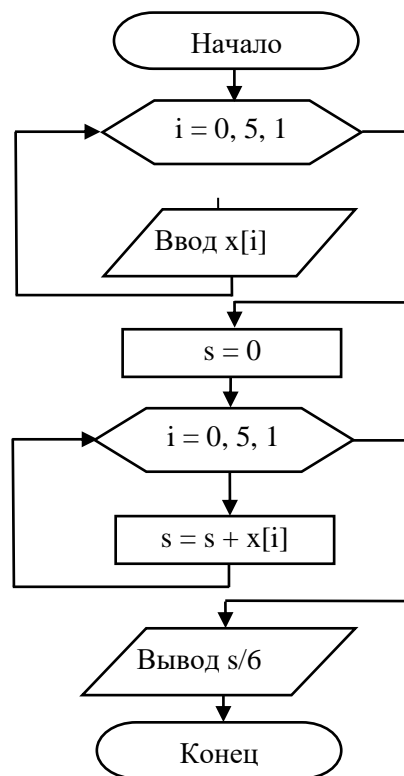
Программный код и графическая схема алгоритма для решения этой задачи приведены на рис. 10.2.

```

int _tmain(int argc, _TCHAR* argv[ ])
{
    double x [6];
    for (int i = 0; i <=5; i ++)
        cin>>x [i];
    double s = 0;
    for (int i = 0; i <=5; i ++)
        s = s + x [i];
    cout<<s/6;
    return 0;
}
.....

```

а



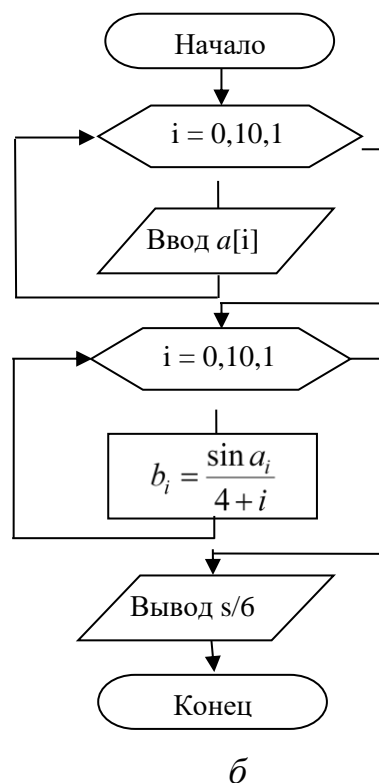
**Рис. 10.1.** Программный код (а) и ГСА (б) вычисления среднего арифметического элементов массива

```

int _tmain(int argc, _TCHAR* argv[ ])
double a[10],b[10];
for (int i=0; i<10; i++)
{
    cin>>a[i];
}
cout<<endl; //для отделения выводимых значений второго массива
for (int i=0; i<10; i++)
{
    b[i]=sin(a[i])/(4+i);
    cout<<b[i]<<endl;
}

```

а



**Рис. 10.2.** Программный код (а) и ГСА (б) формирования нового массива, связанного с исходным заданным соотношением

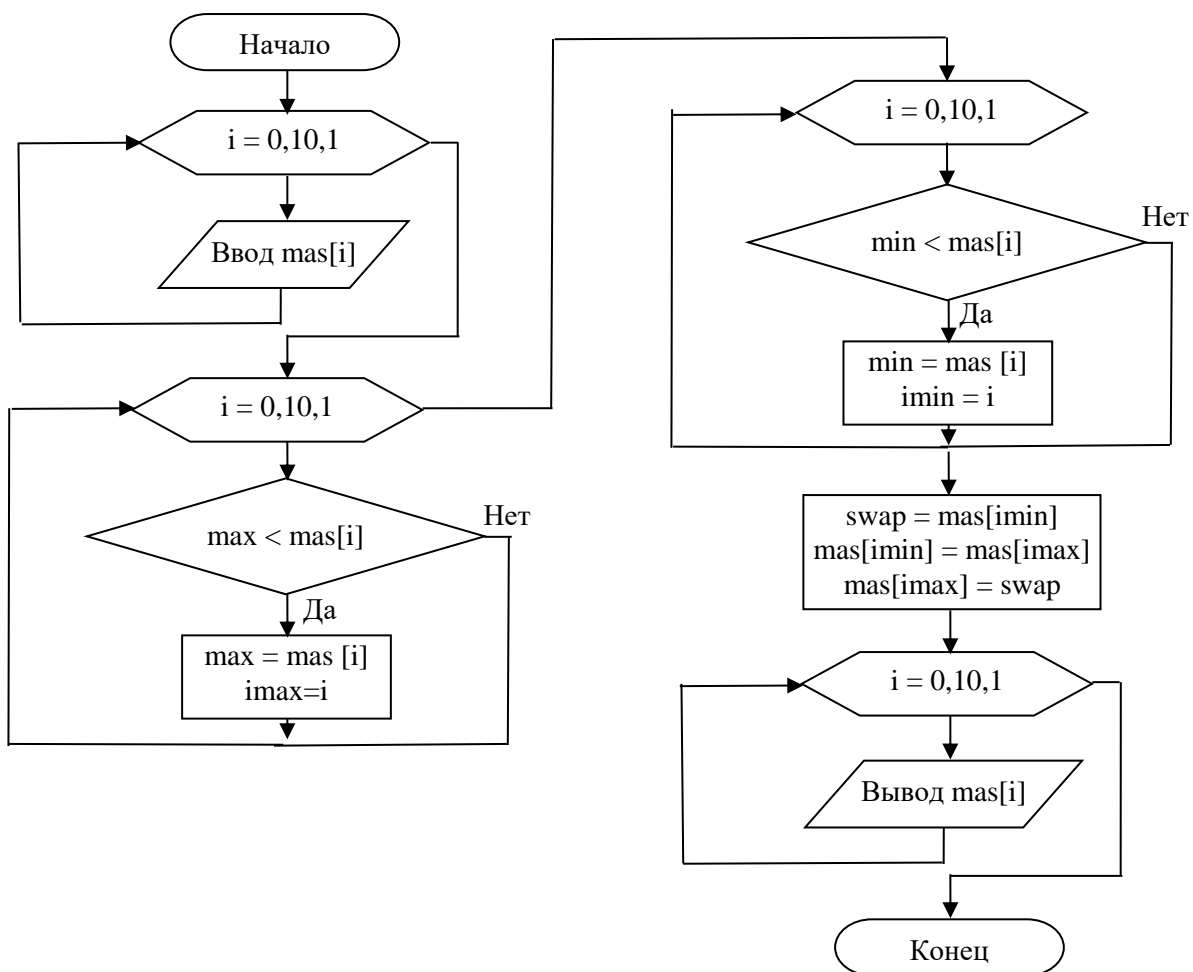
**Пример М3.** Найти в массиве, состоящем из 10 элементов  $mas[10]$ , максимальный и минимальный элементы и поменять их местами.

В качестве начальных значений минимального (максимального) элемента принимается первый элемент  $max = mas[0]$ ,  $min = mas[0]$ , их порядковые номера соответственно  $imax = 0$ ,  $imin = 0$ .

Затем организуется цикл, в котором происходит поочередное сравнение текущего элемента массива с минимальным (максимальным). Если значение очередного элемента массива оказывается меньше (больше)  $min$  или  $max$ , то выполняется замена значения переменной  $max$  или  $min$  «новым» минимальным (максимальным) значением.

Для перестановки элементов массива вводится вспомогательная переменная  $swap$ .

На **рис. 10.3** представлена графическая схема алгоритма для решения задачи.



**Рис. 10.3.** Графическая схема алгоритма поиска минимального и максимального значений в массиве и обмена их местами

Ниже приведен программный код алгоритма, представленного на [рис. 10.3](#):

<pre>double mas [10]; double swap, min, max; int imin = 0, imax = 0;</pre>	
<pre>for (int i = 0; i &lt; 10; i++)     cin &gt;&gt; mas [i];</pre>	<i>Ввод элементов массива mas</i>
<pre>min = mas [0]; max = mas [0];</pre>	<i>Задание начальных значений для min и max</i>
<pre>for (int i = 0; i &lt; 10; i++) {     if (max &lt; mas [i])     {         max = mas [i];         imax = i;     } }</pre>	<i>Поиск значения и места положения максимального элемента массива</i>
<pre>for (int i = 0; i &lt; 10; i++) {     if (min &gt; mas [i])     {         min = mas [i];         imin = i;     } }</pre>	<i>Поиск значения и места положения минимального элемента массива</i>
<pre>swap = mas [imin]; mas [imin] = mas [imax]; mas [imax] = swap;</pre>	<i>Запись максимального значения на место минимального и наоборот</i>
<pre>for (int i = 0; i &lt; 10; i++){     cout &lt;&lt; mas [i] &lt;&lt; 't'; }</pre>	<i>Вывод результирующего (измененного) массива</i>

#### 10.2.1. Вставка и удаление элемента массива

**Пример М4.** В произвольном одномерном массиве  $x[5]$  удалить элемент с  $k$ -й позиции, например:  $k = 2$ .

Процесс удаления  $k$ -го элемента массива с указанной позиции осуществляется по следующему алгоритму:

1 – все элементы до удаляемого остаются на своих местах;

2 – на место удаляемого  $k$ -го элемента ( $k = 2$ ) записывается следующий за ним  $(k+1)$ -й элемент, т. е. каждый элемент после удаляемого смещается влево на одну позицию;

3 – выводится массив размерности  $N - 1 = 4$ .

На [рис. 10.4](#) проиллюстрирован смысл удаления элемента из массива и сдвига остальной части массива влево.

Следует учесть, что при удалении элемента из массива размер массива *не изменится*. Рассматриваемые в данном пособии массивы относятся к категории *статических*, размер которых *задан* изначально и в ходе программирования *не меняется*.

$i$	0	1	2	3	4
Исходный массив	24	-3	15	31	-7
Измененный массив	24	-3	31	-7	-7

[Рис. 10.4.](#) Удаление элемента массива

На [рис. 10.5](#) представлены программный код (*а*) и графическая схема алгоритма (*б*) удаления элемента массива с индексом  $k$ .

**Пример М5.** В одномерном массиве  $x[5]$  вставить элемент в  $k$ -ю позицию, например, для  $k = 2$ ,  $x[2] = 0$ .

Процесс вставки элемента на нужную позицию осуществляется по следующему принципу:

1 – объявляется массив размерности  $N + 1 = 6$  (так как будет вставлен один элемент);

2 – элементы до  $k$ -й позиции остаются без изменения;

3 – выполняется перестановка элементов массива, начиная с последнего в соседнюю справа позицию, т. е. выполняется действие  $x[i+1] = x[i]$ ;

4 – элементу с индексом  $k$  присваивается заданное в условии задачи значение  $x[2] = 0$ .

На [рис. 10.6](#) проиллюстрирована вставка элемента в массив.

.....

`int k;`

`cin>>k;`

`double x[5];`

`for (int i = 0; i <= 4; i++)`

`cin>>x[i]; // ввод элементов массива`

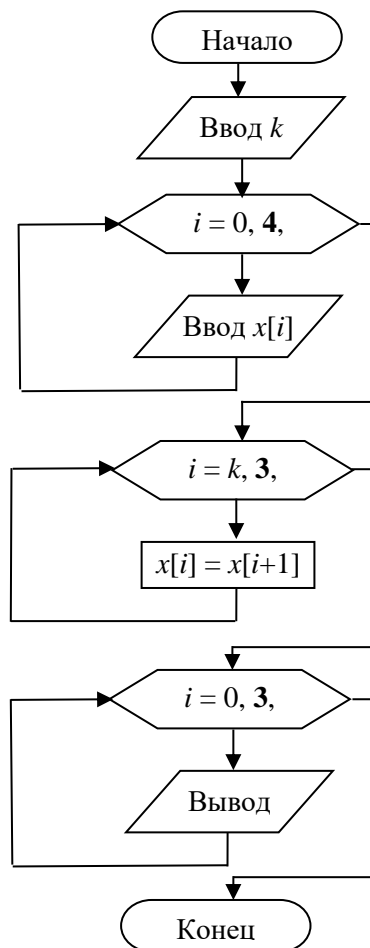
`for (int i = k; i <= 3; i++)`

`x[i]=x[i+1];`

`for (int i = 0; i <= 3; i++)`

`cout<<x[i]<<'t'; // вывод массива`

.....



*a*

*б*

**Рис. 10.5.** Программный код (а) и графическая схема (б) алгоритма удаления элемента массива с индексом  $k$

$i$	0	1	2	3	4	5
Исходный массив	24	-3	15	31	-7	
Измененный массив	24	-3	0	15	31	-7

**Рис. 10.6.** Вставка значения на место с позицией  $k$

На **рис. 10.7** приведены программный код и графическая схема алгоритма вставки элемента в  $k$ -ю позицию:

– инициализируется (задается тип и значение) целочисленная переменная  $k$ , определяющая номер удаляемого элемента в массиве целого типа  $x$ , состоящего из шести элементов;

– массив  $x$  вводится с клавиатуры;

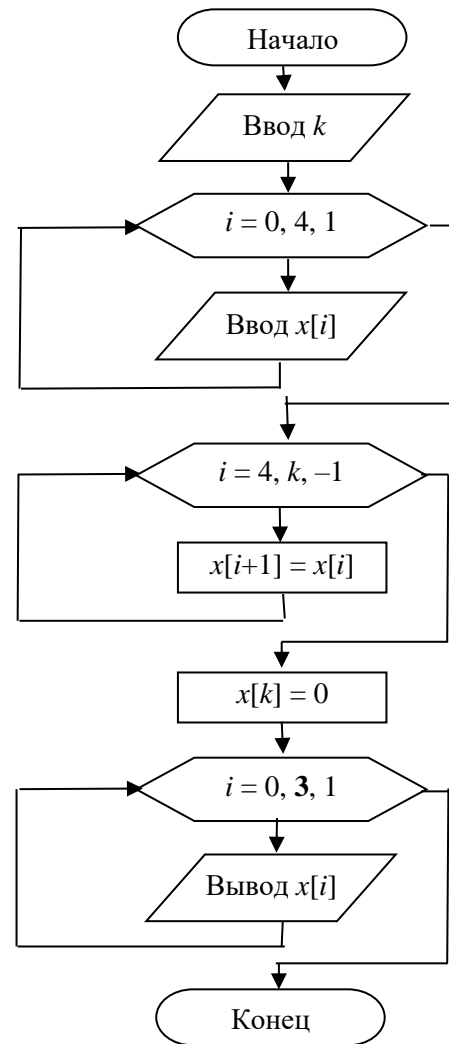
- в счетном цикле *for*, организованном с последнего (с индексом  $i = 4$ ) до  $k$ -го элемента, выполняется сдвиг элементов на следующую позицию;
- элементу, находящемуся на  $k$ -й позиции, присваивается значение 0, заданное в условии;
- производится вывод результирующего массива.

```

.....
int k;
cin >> k;
int x[6];
for (int i = 0; i <= 4; i++)
    cin >> x[i];
for (int i = 4; i >= k; i--)
    x[i+1] = x[i];
x[k] = 0;
for (int i = 0; i <= 5; i++)
    cout << x[i] << 't';
.....

```

*a*



*б*

**Рис. 10.7.** Программный код (*a*) и графическая схема алгоритма (*б*) вставки значения в  $k$ -ю позицию

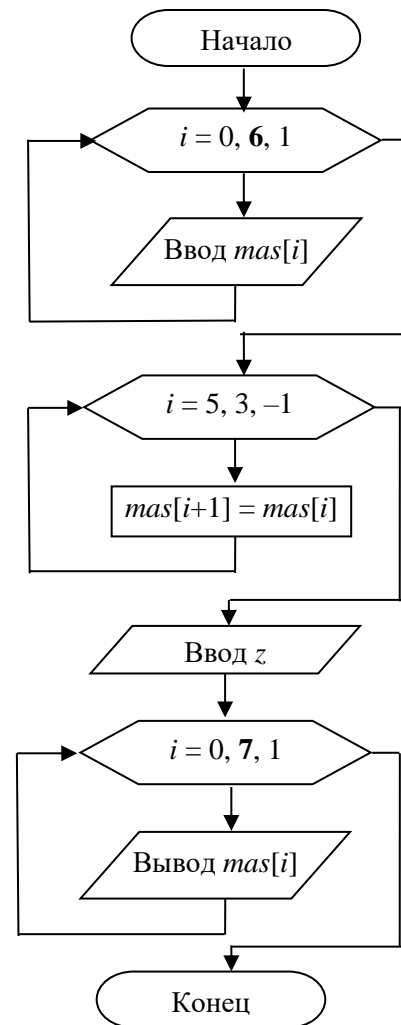
**Пример М6.** Между третьим и четвертым элементами массива вставить произвольное число.

Задача решается аналогично примеру М5 (рис. 10.8). Необходимо на позицию с номером  $k = 4$  вставить произвольное число.



.....

```
int i;  
double mas[7]={1,2,3,4,5,6}, z;  
for (i=5;i>3;i-=1)  
{  
    mas[i+1]=mas[i];  
}  
cin >> z;  
mas[4]=z;  
for (i=0;i<7;i++)  
{  
    cout<<"mas["<<i<<"]="<<mas[i]<<"\t";  
}  
cout<<endl;  
.....
```



*a*

*б*

**Рис. 10.8.** Программный код (*a*) и графическая схема алгоритма (*б*) вставки произвольного числа между третьим и четвертым элементами массива

### 10.2.2. Сортировка массива

Сортировкой называется упорядочивание элементов массива по возрастанию или убыванию. Осуществляется сортировка по следующему алгоритму:

1 – за начальное значение максимального значения *max* – принимается первый элемент исходного массива; за начальное значение порядкового номера (индекса) *imax* – принимается индекс первого элемента массива;

2 – осуществляется поиск максимального значения *max* в массиве и определение его индекса (см. [пример M2](#));

3 – производится перестановка местами первого элемента и максимального элемента, равного *max* с индексом *imax*;

4 – повторяются п. 1 – 3 для массива, начинающегося со второго элемента, и т. д.

**Пример М7.** Выполнить сортировку элементов массива по убыванию.

На [рис. 10.9](#) приведена иллюстрация описанного метода сортировки для массива  $x = \{5, 2, 3, 8, 4\}$ .



**Рис. 10.9.** Сортировка массива по убыванию

На [рис. 10.10](#) представлена графическая схема алгоритма сортировки массива. Ниже представлен программный код для решения указанной задачи:

```

.....
int imax, max;
int x[10];
for (int i = 0; i <= 9; i++)
    cin >> x[i]; // Ввод элементов массива
for (int i = 0; i <= 9; i++) // Внешний цикл, значение параметра кото-
    рого используется для задания начального значения max
    {
        max = x[i]; imax = i;
        for (int j = i + 1; j <= 9; j++) // Цикл поиска максимума в укороченной на
            один элемент части массива
            {
                if (x[j] > max)
                {
                    max = x[j];
                    imax = j;
                }
            }
    }

```

```

    }
}
    x[imax]=x[i]; // Перестановка первого
    x[i]=max;     // и максимального элементов
}
for (int i = 0; i <=9; i++)
    cout<<x[i]<<'t'; // Вывод упорядоченного массива в строку
.....

```

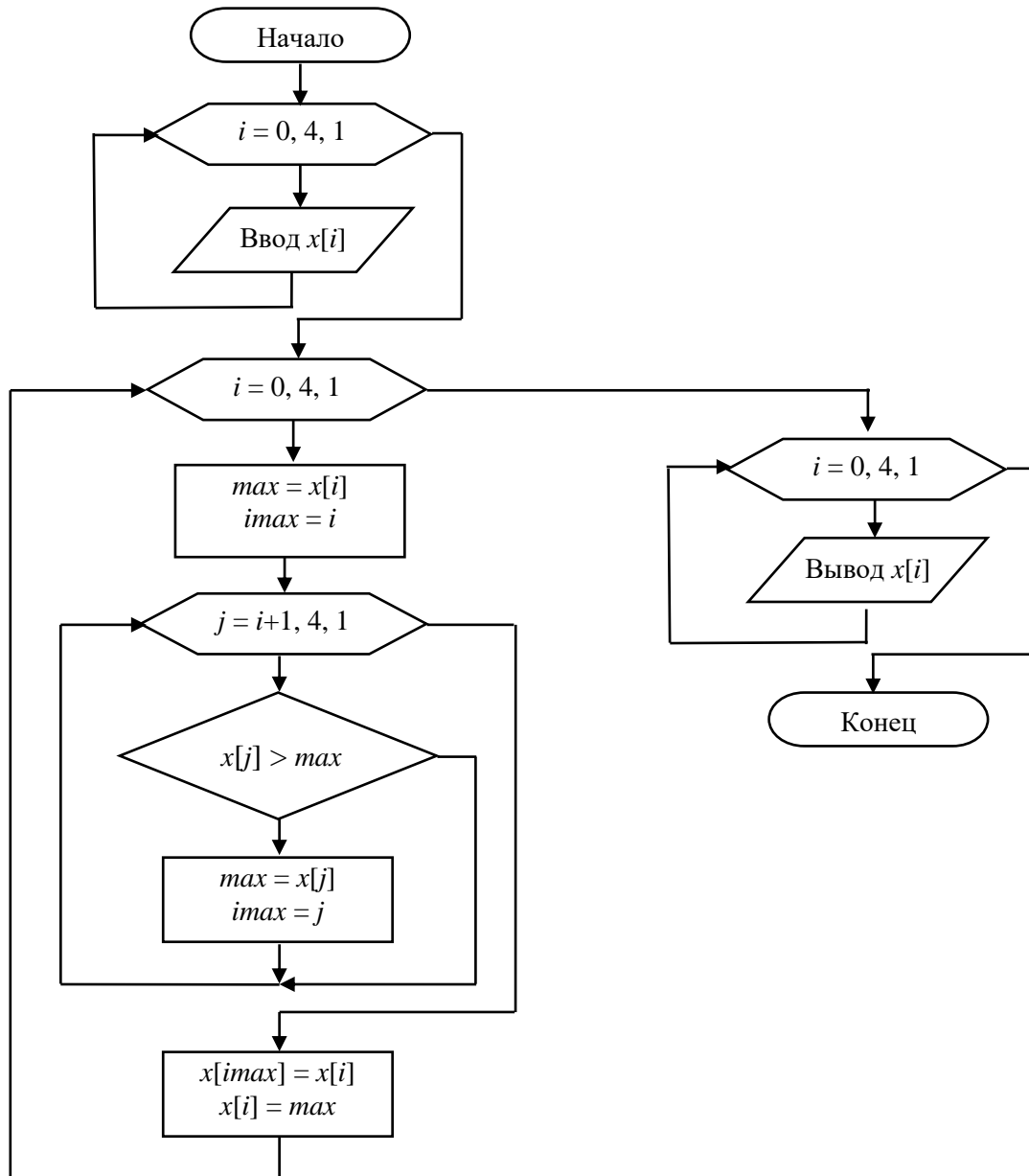


Рис. 10.10. Алгоритм сортировки массива по убыванию

### 10.3. Многомерные массивы

Кроме одномерных массивов в C++ можно создавать и другие, например, двумерный массив определяется следующим образом:

```
int A[n][m].
```

Адреса элементов данного массива определяются двумя индексами – строки и столбца. Для объявленного выше массива  $n$  – номер строки,  $m$  – номер столбца. Таким образом, двумерные массивы можно представлять в виде таблицы. Например, массив  $A[3][4]$  состоит из 12 элементов, и его можно записать в виде:

```
A[0][0] A[0][1] A[0][2] A[0][3]
A[1][0] A[1][1] A[1][2] A[1][3]
A[2][0] A[2][1] A[2][2] A[2][3]
```

Для инициализации, обработки и вывода двумерных массивов необходимо использовать вложенные циклы.

Внешний цикл позволяет изменять номера строк, внутренний цикл – номера столбцов. Например, вывести на экран двумерный массив в виде таблицы, разделяя элементы в строке одним пробелом, можно следующим образом:

```
int A[n][m];
for (int i = 0; i < n; ++i)
{
    for (int j = 0; j < m; ++j)
    {
        cout << A[i][j] << " "; // Вывод на экран элементов массива i-й строки
    }
    cout << endl; // Перевод курсора на новую строку
}
```

### 11. КОНТРОЛЬНЫЕ ВОПРОСЫ

- 1) Какие способы использования комментариев в программном коде на языке C++ существуют?
- 2) Какие символы разрешается использовать в именах переменных?
- 3) Какие символы запрещено использовать в начале имен переменных?
- 4) Каким символом обозначается логическая операция И? Приведите таблицу истинности этой операции.

- 5) Каким символом обозначается логическая операция ИЛИ? Приведите таблицу истинности этой операции.
- 6) Запишите операцию логическое НЕ для переменной *X*.
- 7) Запишите условный оператор *if* в полной и краткой форме.
- 8) Как записывается логическое равенство в операторе *if*?
- 9) В каких случаях следует использовать оператор *switch*?
- 10) Приведите обозначение логического знака «не равно».
- 11) Поясните понятие вложенного цикла.
- 12) Объявите прототип функции с двумя аргументами целого типа, которая возвращает вещественный результат.
- 13) Какая из функций будет вызвана в строке программного кода `float a = abs(-6.0)`, если объявлены прототипы следующих функций: `int abs(int x)`; `float abs(int x)`; `int abs(double x)`?
- 14) Для чего используется ключевое слово `const` в языке C++?
- 15) Каким образом задаются массивы в языке C++?
- 16) Запишите массив целых чисел известными способами.
- 17) Сформулируйте идею алгоритма упорядочивания элементов массива по возрастанию (убыванию).

#### Библиографический список

1. Сидорова, Е. А. Основы программирования на языке VBA / Е. А. Сидорова, С. П. Железняк. – Омск : Омский гос. ун-т путей сообщения, 2021. – 118 с. – Текст : непосредственный.
2. Основы алгоритмизации / Е. А. Сидорова, С. П. Железняк [и др.]. – Омск : Омский гос. ун-т путей сообщения, 2020. – 35 с. – Текст : непосредственный.
3. Либерти, Д. Освой самостоятельно C++ за 21 день / Д. Либерти. – Москва : Вильямс, 2015. – 784 с. – Текст: непосредственный.
4. Дэвис, С. C++ для «чайников» / Дэвис С. – Москва : Вильямс, 2010. – 336 с. – Текст : непосредственный.
5. ГОСТ 19.701–90 (ИСО 5807–85). Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения. – Москва : Изд-во стандартов, 1990. – 36 с. – Текст : непосредственный.

*Учебное издание*

ДАВЫДОВ Алексей Игоревич, КАЛИНИНА Екатерина Сергеевна,  
САЛЯ Илья Леонидович, СТУПАКОВ Сергей Анатольевич

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Учебное пособие

---

Редактор Н. А. Майорова

\*\*\*

Подписано в печать 16.03.2022. Формат  $60 \times 84 \frac{1}{16}$ .  
Офсетная печать. Бумага офсетная. Усл. печ. л. 5,5. Уч.-изд. л. 6,2.  
Тираж 50 экз. Заказ .

\*\*

Редакционно-издательский отдел ОмГУПСа  
Типография ОмГУПСа

\*

644046, г. Омск, пр. Маркса, 35