МЕЖГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛОРУССКО-РОССИЙСКИЙ УНИВЕРСИТЕТ»

Кафедра «Маркетинг и менеджмент»

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Методические рекомендации к самостоятельной работе для студентов специальности 1-28 01 02 «Электронный маркетинг» заочной формы обучения

Часть 1

Могилев 2021

УДК 004.43
ББК 32.973-018
О75

Рекомендовано к изданию
учебно-методическим отделом
Белорусско-Российского университета

Одобрено кафедрой «Маркетинг и менеджмент» «27» мая 2021 г., протокол № 12.

Составитель канд. физ.-мат. наук, доц. С. Н. Батан

Рецензент канд. техн. наук, доц. В. М. Ковальчук

В методических рекомендациях представлены описание аудиторной контрольной работы, критерии её оценки, изложена последовательность изучения теоретических вопросов, приведены основные термины и понятия, а также примерный перечень вопросов для написания аудиторной контрольной работы.

Учебно-методическое издание

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

Часть 1

Ответственный за выпуск	А. В. Александров
Корректор	Е. А. Галковская
Компьютерная верстка	Н. П. Полевничая

Подписано в печать . Формат 60×84/16. Бумага офсетная. Гарнитура Таймс.
Печать трафаретная. Усл. печ. л. . Уч.-изд. л. . Тираж 31 экз. Заказ №

Издатель и полиграфическое исполнение:
Межгосударственное образовательное учреждение высшего образования
«Белорусско-Российский университет».
Свидетельство о государственной регистрации издателя,
изготовителя, распространителя печатных изданий
№ 1/156 от 07.03.2019.
Пр-т Мира, 43, 212022, г. Могилев.

Содержание

1 Описание аудиторной контрольной работы и критерии её оценки

Аудиторная контрольная работа (АКР) направлена на проверку подготовленности студента по теоретической части дисциплины.

АКР включает два теоретических вопроса из тринадцати тем.

Для получения зачета по АКР необходимо дать исчерпывающий ответ на оба теоретических вопроса.

АКР оценивается исходя из 10 (десяти) баллов. Критерии оценки представлены в таблице 1. АКР считается зачтенной, если сумма полученных баллов составляет не менее 5 (пяти) баллов.

Таблица 1 – Критерии оценки АКР

Задание	Максимальный балл
Вопрос 1	5
Вопрос 2	5
Итого по заданиям	10

2 Содержание учебного материала

2.1 Концепция типов данных

2.1.1 Понятие данных и типов.

Данные – это обобщенное понятие, используемое в программировании для обозначения разнообразных информационных единиц, с которыми оперирует компьютер. На физическом уровне любые данные в памяти компьютера представляют собой последовательности двоичных цифр 0 и 1. Использование языков программирования высокого уровня позволило абстрагироваться от деталей и особенностей представления информации главным образом за счет введения концепции типов данных.

Любой тип данных определяет:

1) множество допустимых значений;

2) набор допустимых операций;

3) формат представления данных в памяти компьютера.

В языке C++ каждой переменной, встречаемой в программе, должен быть определен один и только один тип. Программист имеет возможность либо воспользоваться ***предописанными*** (*заранее определенными в языке*) типами, либо создавать свои ***собственные*** (*пользовательские*) типы.

Типы данных в языке C++ представлены на рисунке 1.

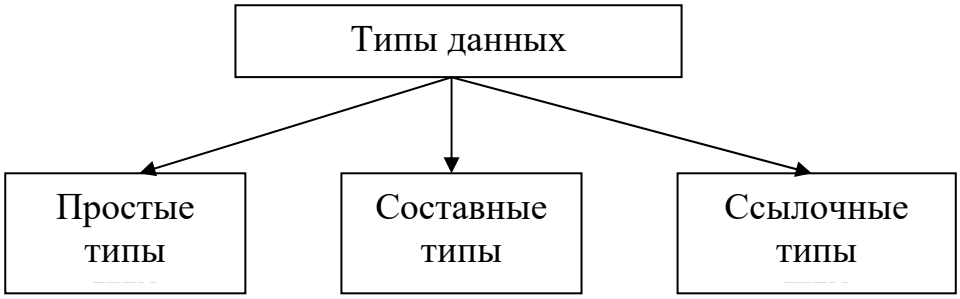


Рисунок 1 – Схема типов данных в языке C++

2.1.2 Простые типы данных.

Простые типы в языке C++ подразделяются на ***ординальные*** и ***вещественные*** типы. ***Ординальный*** (*порядковый*) тип либо определяется программистом (в этом случае это будет *перечислимый* тип), либо обозначается идентификатором одного из предописанных типов (например, логический тип – ***bool***, целочисленный – ***int***, символьный – ***char***).

Схема для простых типов данных изображена на рисунке 2.

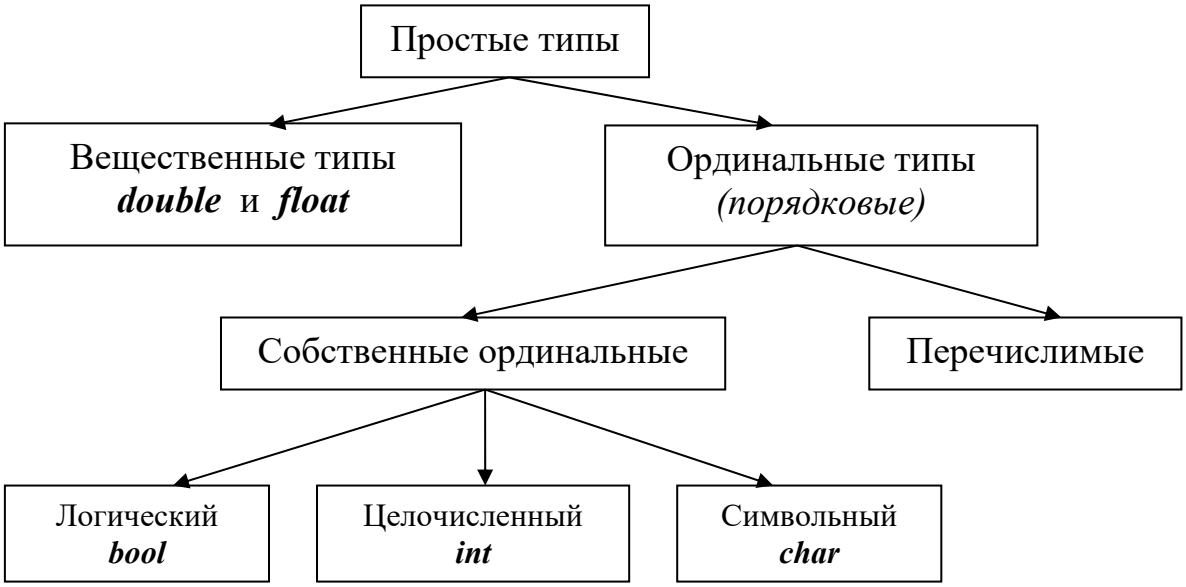


Рисунок 2 – Схема простых типов данных

Ординальный тип данных описывает конечное и упорядоченное множество значений. Эти значения соответственно сопоставляются с последовательностью порядковых номеров от 0 до 255. Исключение сделано для целых чисел, которые отображаются сами на себя.

Для каждого ординального типа определяются минимальное и максимальное значения, при этом, для всех значений, кроме минимального, имеется предыдущее, а для всех, кроме максимального, существует следующее.

В языке C++ к ***простым собственным типам*** относятся: ***bool***, ***double***, ***float***, ***int***, ***char***. Каждый из этих типов имеет определенный набор значений и

внутреннее представление, привязанное к низкоуровневой архитектуре машины, на которой работает компилятор.

Простые типы могут модифицироваться с помощью ключевых слов ***short***, ***long***, ***signed*** и ***unsigned***, и таким образом могут формироваться дополнительные простые типы (рисунок 3).

bool (1 байт)		
char (1 байт)	*signed char* (1)	*unsigned char* (1)
short (2)	*int* (4)	*long* (4)
unsigned short (2)	*unsigned* (4) *unsigned int*	*unsigned long* (4)
float (4)	*double* (8)	*long double* (8)

Рисунок 3 – Простые собственные типы и их модификации

В языке C++ предусмотрен оператор ***sizeof***, с помощью которого можно определить количество байт, необходимое для хранения объекта или типа. Например, можно вывести на экран количество байт для типа ***long double*** с помощью инструкции ***cout << sizeof(long double)***.

2.1.3 Вещественные типы.

Значениями вещественного типа являются элементы определяемого реализацией подмножества вещественных чисел (таблица 2). Все операции над такими величинами выполняются с некоторой степенью точности, которая зависит от конкретной реализации. При условии, что хотя бы один из операндов относится к вещественному типу (*второй операнд может быть и целого типа*), следующие операции в результате выполнения будут формировать вещественное значение: * (умножение); / (деление); + (сложение); – (вычитание).

Таблица 2 – Таблица вещественных типов

Название типа	Диапазон допустимых значений	Количество верных цифр	Размер в байтах
float	±1.175e-38...±3.40e+38	7–8	4
double	±2.23e-308...±1.798e+308	15–16	8

Значащими цифрами числа называют все цифры в его записи, начиная с первой ненулевой слева. ***Значащую цифру*** числа называют ***верной***, если абсолютная погрешность числа не превосходит единицы разряда, соответствующего этой цифре.

2.1.4 Логический тип bool.

Логические переменные типа ***bool*** могут принимать одно из двух значений: ***true*** (истина) или ***false*** (ложь). При этом в языке принята гибкая интерпретация для ***true*** и ***false***. По определению, ***true*** имеет значение 1 при преобразовании к целому типу, а ***false*** – 0. Можно целые значения преобразовать в логические, и

при этом ненулевые целые преобразуются в ***true***, а ноль – в ***false***. Например, допустимы следующие объявления:

bool a = -3; // **a** принимает значение ***true***

int i = false; // **i** принимает значение 0.

В языке С++ предусмотрены следующие три логических (*или булевских*) операции (рисунок 4).

! – логическое отрицание	(NOT)
&& – логическое умножение "И"	(AND)
|| – логическое сложение "ИЛИ"	(OR)

Рисунок 4 – Три логических (*или булевских*) операции

Операция логического отрицания является унарной, т. е. применяется к одному операнду, размещаемому справа от !. Правила выполнения операции представлены на рисунке 5.

! FALSE = TRUE
! TRUE = FALSE

Рисунок 5 – Правила выполнения унарных операций

Правила выполнения бинарных операций && и || приведены на рисунке 6.

Операнд		Результат операции	
X	Y	X && Y	X || Y
FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE

Рисунок 6 – Правила выполнения бинарных операций

Правила выполнения логических операций определены таким образом, что логическое отрицание ! (NOT) имеет, как и всякая унарная операция, наивысший приоритет, следующий приоритет у операции логического умножения && (AND), а самый низкий – у операции логического сложения || (OR). Логические выражения могут содержать не только логические операции, но и сравнения *(отношения типа X<Y)*, которые имеют более низкий приоритет выполнения. В арифметических выражениях логические значения преобразуются в целые типа ***int***.

Например: ***int k = true + true;*** // в результате ***k*** получает значение 2.

Указатель можно неявно преобразовать в ***bool***, при этом ненулевой указатель принимает значение ***true***, а нулевой – ***false***.

2.1.5 Символьные и целочисленные типы в С++.

Для работы с символьными данными в языке С++ предусмотрен специальный тип данных ***char***, значения которого в памяти компьютера занимают 1 байт.

Каждый используемый символ в компьютере имеет уникальный код (*индивидуальный номер*) в кодовой таблице.

Основные характеристики символьного типа описываются в таблице 3.

Таблица 3 – Основные характеристики символьного типа

Название	Диапазон допустимых значений	Размер в байтах
char	0…255	1
signed char	–128…+127	1
unsigned char	0…255	1

Основные характеристики целочисленных типов описываются в таблице 4.

Таблица 4 – Основные характеристики целочисленного типа

Название	Диапазон допустимых значений	Размер в байтах
short	–32768…+32767	2
int	–2147483648…+2147483647	4
unsigned short	0…65535	2
unsigned int	0…+4294967295	4

Типы ***unsigned short*** и ***unsigned int*** используются для представления целых значений без знака, а типы ***short*** и ***int*** – со знаком.

При выполнении действий над целыми операндами следующие арифметические операции вырабатывают целочисленные значения: * – умножение; / – деление; % – остаток от деления целых чисел; + – сложение; – – вычитание.

В стандартном заголовочном файле ***limits.h*** определяется диапазон целых значений для данной версии С++. Например, можно найти следующие определения для Visual Studio С++:

```
#define INT_MIN     (-2147483647 - 1) /* minimum (signed) int value */
#define INT_MAX       2147483647    /* maximum (signed) int value */
#define UINT_MAX      0xffffffff    /* maximum unsigned int value */
```

2.1.6 Тип void и перечислимые типы.

В С++ существует специфический тип ***void***. Этот тип используется либо для указания того, что соответствующая функция не возвращает значения, либо в качестве базового типа для указателей на объекты неизвестного типа. Например: ***void*** f(x,y); // функция f не возвращает значение ***void****** pt; // указатель на объект неизвестного типа.

С помощью ключевого слова ***enum*** (*перечисление*) пользователь может определить перечислимый тип. Такой тип задает набор значений, опреде-

ляемый пользователем. Например: ***enum cities { Brest, Grodno, Gomel, Vitebsk, Minsk, Mogilev }.*** Здесь ***Brest, Grodno,…, Mogilev*** являются константами *перечислимого* типа, а их значения равны соответственно 0,1,...,5. Эти значения присваиваются по умолчанию, причем первой перечислимой константе дается значение 0, а каждый следующий элемент списка больше на 1, чем сосед слева.

Допустимо при объявлении перечислимого типа инициализировать перечислимые константы целыми выражениями: ***enum cities*** {***Brest*** =16, ***Grodno***, ***Gomel***=***Grodno***+3, ***Vitebsk***=25, ***Minsk***, ***Mogilev***=***Vitebsk***+5}. В этом случае соответствующие перечисленные значения будут ассоциированы с целочисленными следующим образом: ***Brest*** =16, ***Grodno***=17, ***Gomel***=20, ***Vitebsk***=25, ***Minsk***=26, ***Mogilev***=30. По умолчанию, при выполнении арифметических операций перечисления преобразуются в целые значения.

В языке C++ можно сконструировать следующие типы:

- *тип* «*указатель на*» (*например, указатель на целое – int**);
- *тип массив* (*например, char* [ ]);
- *тип ссылки* (*например, float&*);
- *структуры данных и классы.*

Логические, символьные и целые типы вместе называются ***интегральными типами***. *Интегральные* вместе с типами с плавающей точкой называются ***арифметическими*** типами. *Перечисления, структуры данных и классы* называются ***пользовательскими типами***. Остальные типы называются ***встроенными***.

2.2 Объявления в языке C++

Прежде чем ***идентификатор*** (***имя***) будет использован в программе, он должен быть объявлен. В языке C++ объявление может формироваться из следующих четырех составляющих:

1) *необязательного спецификатора*;
2) *базового типа*;
3) *объявляющей части*;
4) *необязательного инициализатора*.

За исключением определений функций и пространств имен объявление заканчивается точкой с запятой.

Например: ***char**** prog_lang[]={“Pascal”,”C++”,”Fortran”};

Здесь спецификатор отсутствует, базовым типом является ***char***, объявляющей частью – **** prog_lang[]***, а инициализатором – ***={“Pascal”, “C++”, “Fortran”}***. В качестве *спецификаторов* могут использоваться такие ключевые слова, как ***virtual*** и ***extern***. Они описывают характеристики, которые не связаны непосредственно с типом.

Объявляющая часть состоит из имени и может содержать оператор объявления. К операторам объявления, которые используются наиболее часто, относятся: * – указатель; & – ссылка; [] – массив; ( ) – функция.

Когда для имени указан *инициализатор*, то он определяет начальное значение. Если же инициализатор не задан, то глобальным именам, именам из пространства имен и локальным статическим присваивается нулевое значение соответствующего типа.

Например:

int x; // означает фактически ***int x = 0***;
double z; // означает ***double z = 0.0.***

Локальные переменные и создаваемые в динамической Heap-области не инициализируются по умолчанию. В языке C++ большинство объявлений являются еще и ***определениями***, т. е. они определяют некую сущность, которая соответствует имени. Например, запись ***int x*;** означает, что для переменной ***x*** определяется формат представления в памяти и количество байт (4). Бывают объявления, которые не содержат определения сущности, т. е. сущности, на которые они ссылаются, должны быть определены где-то в другом месте. Такие объявления не являются определениями. Примером может быть объявление прототипа функции: ***double f*(*double*);**

По правилам C++ для каждого имени должно быть только одно определение, а вот объявлений может быть несколько. При этом все объявления некой сущности должны согласовываться по ее типу.

Объявления, начинающиеся с ключевого слова ***typedef***, позволяют вводить новое имя для типа, а не для переменной соответствующего типа. Имена, вводимые с помощью ***typedef***, являются антонимами, а не новыми типами, а потому старые типы можно использовать совместно с их синонимами.

***Пример 1 – typedef dbl double; dbl p; double y*;**

Подобные объявления используются, например, для назначения короткого синонима часто применяемым типам.

Остановимся на понятии ***области видимости имен***. Для имени, объявленного в теле функции, область видимости имени ограничивается соответствующим блоком. Блоком в C++ называется фрагмент программы, заключенный в фигурные скобки { }. Отметим, что подобное имя называется ***локальным. Глобальным*** же называется имя, которое объявлено вне любой функции, класса и пространства имен. Область видимости глобальных имен простирается от места их объявления и до конца файла, содержащего это объявление. Объявление имени в блоке может скрыть (заместить) объявление этого имени в охватывающем блоке или может скрыть глобальное имя.

Пример 2

```
1    # include < iostream>
2    using namespace std;
3    double x;  // объявление глобальной переменной
4    void f( )
5    {
6    double x;        // локальная переменная скрывает глобальную
7    x = 17.5;        // присваивает значение локальной переменной
8    {
```

```
9     double x; // это объявление скрывает 1 локальную переменную
10    x = -2.5;   // присваивание 2-й локальной переменной
11    }
12     x = -5.0;  // присваивание 1-й локальной переменной x
13    cout << x  <<  endl;
14    }
15    int main()
16    {
17    x =12.0;
18    f();
19    cout << x  << endl;
20    return 0;
21    }  // на экране в 1-й строке выводится -5.0; а во 2-ой –  +12.0
```

К скрытому глобальному имени можно обратиться с помощью оператора разрешения области видимости **::** . Например, можно заменить 13-ю строку на ***cout << ::x << endl;*** и тогда на экране дважды выведется значение 12.0. Отметим, что не предусмотрен подобный механизм обращения к скрытой локальной переменной.

В C++ допустим такой фрагмент программы:

```
…
double x= 1.5;
void f ()
{
double y = x;     // тут  x – глобальная переменная, а потому y=1.5
double x = -2.5;  // объявляется локальная переменная x
                     // и инициализируется значением –2.5
y = x;               // y получает значение локальной переменной x
}
…
```

Имена аргументов функции определяются в самом внешнем блоке функции, и поэтому в следующем фрагменте содержится ошибка, связанная с тем, что переменная x дважды определяется в пределах одной и той же области видимости:

```
void f ( int x)
{…
int x;  // это повторное определение, что недопустимо!
…
}
```

2.3 Выражения в языке C++

В C++ имеется достаточно много форм представления выражений. Для построения выражений могут использоваться операторы, которых всего в языке

предусмотрено около 70 (по Б. Страуструпу их 68). Отметим, что в С++ присваивание является выражением, а потому допустима следующая запись: ***x = y + (m = j*4-1.5);***

В языке С++, в отличие от большинства других языков программирования, принята терпимая позиция по отношению к смешению типов и автоматическому преобразованию в арифметических выражениях. Допускаются как расширяющие преобразования типов (когда, например, ***int*** может быть расширен до ***double*** при присваивании), так и присваивания с преобразованием к более узкому типу (когда, например, ***double*** может быть присвоен типу ***int*** или ***char***).

Еще одно отличие, характерное для С++, заключается в том, что результат оператора деления (/) зависит от типа аргументов.

Например:

```
double x;
x=7/2; // x присвоится значение 3
x=7/2.0;       // x присвоится значение 3.5
```

Следует понимать, что подобная терпимая позиция С++ может приводить к возникновению ошибок, которые непросто устранять, и потому программист должен стремиться к явному использованию соответствующих типов.

Для осуществления контроля за потоком управления в некоторых инструкциях используются логические значения ***true*** и ***false***. В языке С++ предусмотрены операторы сравнения (**<; >; <=; >=**), операторы равенства (== (равно), **!=** (не равно)), логические операторы (**!** (отрицание), **&&** (логическое И), || (логическое ИЛИ)). Среди этих операторов все ***бинарные***, за исключением оператора **!**, который является ***унарным***. Приоритет **&&** выше, чем ||, но оба оператора имеют более низкий приоритет, чем все унарные, арифметические операторы и операторы сравнения.

В С++ предусмотрен условный оператор ***expr1?expr2:expr3***, который содержит в качестве операндов три выражения. Сначала вычисляется выражение ***expr1***, и если оно ***true***, то вычисляется ***expr2***, а полученное значение становится значением условного выражения в целом. Если же значение ***expr1*** равно ***false***, то вычисляется ***expr3***, и его значение становится значением условного выражения, например: ***max_xy = (x>y) ? x : y.*** Отметим, что использование круглых скобок в данном примере необязательно, т. к. оператор присваивания имеет более низкий приоритет, чем оператор сравнения.

Пример 3 – Вводится значение аргумента **x**. Если **x** – положительное, то **y** получает значение $\mathbf{x}^2$, а иначе $\mathbf{x}^3$.

```
#include <iostream>
using namespace std;
int main()
{double x,y;
 cout << "Input x:";  cin >> x;
 y=(x>=0.0)? x*x : x*x*x;
```

```
  cout << "y=" << y << endl;
  return 0;
}
```

В языке предусмотрены битовые операторы, которые воздействуют на машинно-зависимое битовое представление целых операндов. По своему значению и по приоритетам битовые операторы размещаются следующим образом:

~ побитовое отрицание (1↔0)
<< >> побитовый сдвиг влево и вправо
& побитовое и
^ побитовое исключающее или
| побитовое или

В C++ ***вызов функции*** **( )**, ***индексация массива*** **[ ]**, ***определение адреса*** **&**, ***обращение по адресу*** ***** также рассматриваются как операторы, причем операторы определения адреса & и обращения по адресу * являются ***унарными***. Первый возвращает адрес, по которому хранится объект, а второй применяется к указателям и позволяет получить значения по адресу, на который ссылается указатель.

С помощью оператора ***sizeof*** можно определить, сколько байт потребуется для хранения либо значения некоторого типа, либо конкретного объекта.

Оператор «***запятая***» имеет самый низкий приоритет из всех операторов C++. Это бинарный оператор с выражениями в качестве операндов. В выражении с запятой вида ***expr1,expr2*** сначала вычисляется ***expr1***, а затем ***expr2***. В результате такое выражение имеет значение и тип своего правого операнда.

Например:

```
…
{ int k;   double s;
 k=12,     s=0.5;   // данное выражение с запятой имеет тип double и значение 0.5
 …}
```

Пример 4

```
… x=1; y=5; z=-1;
cout << ( x=2*x, y=x*y, z=2*z) << endl;
```

Такой вид ***cout*** является допустимым и на экран будет выведено –2, поскольку последним в списке стоит присваивание ***z=2*z***.

Остановимся на операторах присваивания, которые предусмотрены в языке. Допускается такой оператор присваивания ***x= i+1;*** , когда в начале вычисляется правая часть присваивания, затем она преобразуется к значению, совместимому с переменной в левой части, и оно присваивается левой части.

В C++ в левой части оператора присваивания может размещаться не просто имя, как в большинстве языков, а так называемое ***именующее***

выражение (***lvalue*** – left value). Фактически ***lvalue*** можно трактовать как простейший объект (место в памяти), в котором значение может храниться и извлекаться. В дополнение к оператору простого присваивания вида ***lvalue=expr*** (где ***expr –*** выражение) предусмотрены еще следующие операторы присваивания:

1) умножение и присваивание ***lvalue*=expr*** (эквивалентно ***lvalue=lvalue*expr***);
2) деление и присваивание ***lvalue/=expr*** (***lvalue=lvalue/expr***);
3) остаток и присваивание ***lvalue%=expr*** (***lvalue=lvalue%expr***);
4) сложение и присваивание ***lvalue+= expr*** (***lvalue=lvalue+expr***);
5) вычитание и присваивание ***lvalue–= expr*** (***lvalue=lvalue–expr***);
6) сдвиг влево и присваивание ***lvalue<<=expr*** (***lvalue=lvalue<<expr***);
7) сдвиг вправо и присваивание ***lvalue>>=expr*** (***lvalue=lvalue>>expr***);
8) побитовое и присваивание ***lvalue&=expr*** (***lvalue= lvalue&expr***);
9) побитовое или и присваивание ***lvalue|=expr*** (***lvalue=lvalue|expr***);
10) исключающее или и присваивание ***lvalue^=expr*** (***lvalue=lvalue^ expr***).

Особенностью C++ является то, что в одной инструкции допускается использование нескольких присваиваний, например: ***x=y+ (c=2*z)+(z=2*y);***

В языке предусмотрены операторы ***инкремента*** ++ (увеличение на 1) и ***декремента*** -- (уменьшение на 1), которые могут представляться в префиксной и постфиксной формах. В префиксной форме оператор инкремента прибавляет 1 к значению lvalue, а декремента – вычитает 1 из value.

++ i; равнозначно i=i+1;
--j; равнозначно j=j–1;

Постфиксная форма предусматривает, что значение ***lvalue*** изменяется после того, как вычислена остальная часть выражения.

j=i++; равнозначно выполнению операторов j=i; i=i+1;
j=++i; равнозначно выполнению операторов i=i+1; j=i;

2.4 Инструкции выбора в языке C++

В языке C++ такие понятия, как операторы безусловного перехода (типа ***goto***), ветвления (типа ***if***), организации циклов (типа ***for***, ***while***) принято называть инструкциями, а операторы C++ были в основном рассмотрены на предыдущей лекции.

Специфика C++ такова, что ***объявление*** является ***инструкцией***, а ***оператор присваивания*** и ***оператор вызова функции*** являются ***выражениями***.

Для организации ветвлений в языке предусмотрены инструкции выбора ***if*** и ***switch***, которые имеют следующий синтаксис:

***if (*условие*)* инструкция**
***if (*условие*)* инструкция *else* инструкция**
***switch (*условие*)* инструкция**

В данном случае в ***if*** и ***switch*** соответствующие инструкции могут быть как простыми, так и составными. ***Составная инструкция*** – это последовательность инструкций, заключенных в фигурные скобки { и }.

При использовании ***if (*условие) инструкция** следует следить за правильным размещением знака **;** (точка с запятой). Например, допустима следующая инструкция: ***if(x>0); x=x+2;*** В этом случае x всегда будет увеличиваться на 2 (*это связано с наличием **;** перед оператором присваивания, и эта **;** трактуется как пустая инструкция*).

Следующая инструкция в C++ будет ошибочной:

if (x>0) x=x+2 else x=x-2; // здесь перед else отсутствует ; (точка с запятой)

if (x>0) x=x+2; else x=x-2; // правильная инструкция

if (x>0) x=x+2;; else x=x-2; // неправильная, лишняя ; (точка с запятой)

Часто удобно вводить переменные в наименьшей возможной области видимости. Например, локальную переменную лучше объявлять в тот момент, когда ей надо присвоить значение, поскольку в этом случае исключаются попытки использования переменной до момента ее инициализации. Можно в C++ объявление разместить внутри условия, например допустима следующая последовательность инструкций:

```
…
   int  y;
   cin >> y;
   if (int j=y)  x=x*j+2;
…
```

В инструкции ***switch (*условие*)* инструкция** обычно используется составная инструкция, содержащая метки ***case*** и необязательную метку ***default***. Каждая метка ***case*** должна быть уникальна, а синтаксис ***case*** таков:

case **целое_постоянное_выражение :**

Обычно действие, выполняемое после каждой метки ***case***, заканчивается инструкций ***break***. Если ***break*** отсутствует, то управление передается или следующей инструкции, идущей за очередным **case,** или ***default***.

Пример 5 *– (фрагмент программы с инструкцией **switch**)*

```
…
int ocenka_examen;
…
switch ( ocenka_examen) {
  case 10: case 9:
cout << “Экзамен сдан отлично!”; break;
   case 8: case 7: case 6:
cout << “Экзамен сдан успешно.”; break;
    case 5: case 4:
cout  << “Экзамен сдан удовлетворительно.”; break;
```

default:
cout << "Экзамен не сдан!";
}...

Правила выполнения ***switch***.

1 Вычисляется выражение в круглых скобках, стоящих за служебным словом ***switch***.

2 Выполняется метка ***case***, совпадающая с тем значением, которое было найдено на предыдущем этапе. Если нет ни одной соответствующей метки ***case***, то выполняется метка ***default***. Если же метки ***default*** нет, то ничего не выполняется, а инструкция ***switch*** заканчивается.

3 Выполнение ***switch*** прерывается, когда встречается инструкция ***break*** или когда достигается конец ***switch***.

}

2.5 Инструкции для организации циклов в языке C++

В языке C++ Для организации циклов предусмотрены инструкции трех видов:

1) ***while*** **(условие) инструкция**
2) ***do*** **инструкция** ***while*** **(выражение)**
3) ***for*** **(инициализир_инструкция [условие] ; [выражение]) инструкция;**

Элементы, заключенные в прямоугольные скобки [], в данном случае являются необязательными.

Каждая из этих трех инструкций будет многократно выполняться до тех пор, пока условие не примет значение ***false***, либо пока не произойдет прерывание цикла каким-либо другим способом, например, с помощью инструкций ***break*** или ***goto***.

При использовании циклов следует иметь в виду, что, во-первых, чтобы цикл когда-нибудь закончился, необходимо в его теле предусматривать влияние на условие цикла, и, во-вторых, условие цикла должно состоять из корректных компонент, определенных еще до первого выполнения тела цикла.

В цикле ***while*** стоящая после условия **инструкция** представляет собой тело цикла. В начале выполнения этого цикла вычисляется значение **условия**. Если оно ***false***, то оператор не выполняется и управление сразу же передается следующему оператору. Если же значение равно ***true***, то выполняется **инструкция**. Отметим что, поскольку перед выполнением цикла предварительно проверяется условие, то цикл называется «***цикл с предусловием»***. Отличительной чертой такого цикла является то, что он может ни разу не выполняться.

Инструкция ***do*** представляет собой вариант инструкции ***while***. Однако вместо проверки логического условия в начале цикла, в инструкции ***do*** она производится в конце. Отсюда очевидно, что цикл ***do*** выполняется хотя бы один раз, тогда как цикл ***while*** может ни разу не выполняться.

Пример 6 *– (фрагмент программы с циклом* ***do****)*

```
…
do
{ cout << “Vvedite veschestvennoe chislo v diapazone ot –20 do +40:”;
  cin >>  x; }
while (!(x>=-20 && x<=40));
…
```

Обратить внимание, что **условие** обязательно записывается в круглых скобках, иначе будет ошибка.

Правила выполнения цикла.

for **(инициализир_инструкция [условие] ; [выражение]) инструкция;** таковы, что сначала выполняется **инициализирующая инструкция**, которая ***обычно*** присваивает в результате некоторое значение управляющей переменной цикла. **Инициализирующей инструкцией** может быть инструкция-выражение или просто объявление (будучи объявленной, управляющая переменная имеет область видимости инструкции ***for***). После выполнения **инициализирующей инструкции** проверяется **условие**. Если оно истинно, то выполняется тело цикла, затем вычисляется выражение, и управление передается обратно в начало цикла ***for*** с той разницей, что **инициализирующая инструкция** больше не выполняется. Это продолжается до тех пор, пока условие не примет значение ***false***.

Допустимо отсутствие условия, и тогда цикл ***for*** выполняется либо бесконечно, либо до тех пор, пока явно не будет прерван (например, с помощью ***break***, ***goto***, ***return***).

В цикле ***for*** может отсутствовать выражение, и тогда необходимо предусматривать обновление переменной цикла в теле цикла. Отметим, что разрешается в качестве **инициализирующей инструкции** использовать т олько ; (точку с запятой), и будет допустимой ***for***-инструкция следующего вида: ***for (; ;) { … }***.

Переменную(ые) можно объявлять в инициализирующей инструкции цикла ***for***. Область видимости такой переменной или переменных до конца ***for***-инструкции. Например, ***for (int*** **i=0; i<max; i++) s=s+i;**

В языке С++ имеются инструкции ***break*** и ***continue***.

Инструкция ***break*** может встречаться только внутри тела циклов ***for***, ***while***, ***do*** или внутри инструкции ***switch***. Она прерывает выполнение соответствующей инструкции и передает управление следующей в программе.

Инструкция ***continue*** вызывает остановку выполнения очередного шага цикла и производит переход на начало очередного (следующего) шага цикла. ***Continue*** может использоваться только внутри инструкций ***for***, ***while*** или ***do***.

В языке С++ имеется инструкция безусловного перехода ***goto*** **метка**; **Меткой** в языке является идентификатор. Следует иметь в виду, что и инструкция ***goto***, и соответствующая ей помеченная инструкция должны находиться в теле одной функции. Обратим внимание на то, что использование

инструкции ***goto*** может оказаться эффективным для организации выхода из вложенных циклов или вложенных ***switch***-инструкций.

2.6 Функции в языке C++

В языке C++ функция фактически представляет собой подпрограмму, которая осуществляет обработку данных и может возвращать некоторое значение. Любая программа на C++ обязательно содержит функцию со стандартным именем ***main***. Эта функция вызывается операционной системой при запуске программы, а при завершении выполнения программы функция ***main*** возвращает управление операционной системе.

В C++ функции подразделяют на два вида: 1) ***встроенные***, которые являются составной частью компилятора и поставляются вместе с ним; 2) ***нестандартные***, которые разрабатываются самим программистом.

По правилам языка перед использованием функции в программе требуется сначала объявить ее, а затем и определить. При объявлении функции компилятору сообщается тип возвращаемого функцией значения, ее имя и список параметров.

Объявление функции можно реализовать с помощью так называемого ***прототипа функции***, который состоит из типа возвращаемого значения функции и сигнатуры. В свою очередь под ***сигнатурой*** функции подразумевается ее ***имя*** и ***список формальных параметров***. Таким образом, прототип функции имеет следующий синтаксис:

тип имя_ функции ([список_формальных_параметров]);

Список формальных параметров может быть пустым, содержать одно или несколько объявлений параметра, разделенных запятыми. Если функция параметров не имеет, то вместо списка формальных параметров разрешается использовать ключевое слово ***void***. При записи объявления параметра можно указать тип параметра и его имя или можно ограничиться только типом, например:

double max (***double*** x, ***double*** y);
double max (***double***, ***double***);

Теперь перейдем к определению функции. По правилам языка оно состоит из ***заголовка функции*** и ***тела функции***, которое представляет собой набор соответствующих инструкций.

Заголовок подобен ***прототипу*** функции за исключением того, что, во-первых, объявление параметра обязательно содержит тип и имя, во-вторых, в конце заголовка точка с запятой не ставится.

Синтаксис определения функции следующий:

заголовок функции
{
***инструкции тела функции*;**
}

При вызове функции ее выполнение начинается с первой инструкции, стоящей за открывающейся фигурной скобкой. Внутри функции можно задавать вызов других функций, и даже организовать рекурсию, т.е. допускается вызов функции самой себя.

Пример 7 *– Демонстрируется программа, содержащая две функции.*

Рассмотрим пример программы, в котором вводятся координаты трех вершин треугольника и выводятся значения длин его сторон. Подсчет длины стороны треугольника производится с использованием функции dlina.

```
# include < iostream>
# include < math.h> // для того, чтобы использовать функцию sqrt,
                    // которая вычисляет значения корня квадратного
using namespace std;

double dlina (double, double, double, double); // прототип ф-ии dlina

int main ()
{       double xA,xB,xC,yA,yB,yC;

cout << "VVedite koordinaty vershiny A(xA,yA):";    cin>>xA>>yA;
        cout << "VVedite koordinaty vershiny B(xB,yB):";    cin>>xB>>yB;
        cout << " VVedite koordinaty vershiny C(xC,yC):";    cin>>xC>>yC;

cout << "Dlina AB=" << dlina(xA,yA,xB,yB) << endl
<< "Dlina AC=" << dlina(xA,yA,xC,yC)<< endl
<< "Dlina CB=" << dlina(xC,yC,xB,yB) << endl;
return 0;
}
double dlina(double x1, double y1, double x2, double y2) // заголовок
{ return sqrt (( x1-x2)*(x1-x2)+(y1-y2)*(y1-y2)); }          // тело функции
```

Обратим внимание на то, что параметры x1, y1, x2, y2 в определении функции являются ***формальными параметрами***. Эти параметры, во-первых, используются для задания правил вычисления (задают формулу вычислений, а потому **форм(у)альные**), во-вторых, будут замещены ***фактическими значениями***, которые передаются функции в момент вызова.

Инструкция ***return*** передает управление обратно в вызывающую функцию, либо если она размещена внутри функции ***main***, то управление передается операционной системе. Синтаксис инструкции return следующий:

return [***выражение***];

Если в инструкции предусмотрено выражение, то при выполнении оно вычисляется и передается либо вызывающей функции, либо операционной системе. Отметим, что использование ***return*** не является обязательным, но рекомендуемым по правилам «хорошего тона».

В C++ ***формальному параметру*** может задаваться ***аргумент по умолчанию***, т. е. значение по умолчанию. Если для функции нескольким формальным параметрам задаются аргументы по умолчанию, то в списке эти параметры размещаются правее всех остальных. В том случае, когда для функции используется прототип функции, необходимо в нем определить аргументы по умолчанию.

Пример 8

```
double real_of_power (double n, int m=2) // m=2 по умолчанию
{
if (m==2) return (n*n);
else
{ double s; s=n*n; for (int i=3;  i<=m; i++) s=s*n;
return s;}
}
```

В этом примере предполагается, что чаще всего функция ***real_of_power*** применяется для вычисления n^2, но может использоваться и для вычисления более высоких степеней ***n***.

В языке C++ предусмотрена возможность перегрузки функций, которая связана с использованием нескольких одноименных функций, т. е. в языке допускается использование несколько функций с одинаковыми именами, но при этом перегруженные функции должны отличаться между собой ***списками параметров*** (***сигнатурой***). Таким образом, сигнатуры двух функций с одинаковыми именами должны отличаться либо типом одного или нескольких параметров, либо различным количеством параметров, либо и тем, и другим одновременно. При этом типы возвращаемых значений перегруженных функций могут быть или одинаковыми, или разными.

Следует иметь в виду, что недопустимо использование двух функций с одинаковыми именами и сигнатурами, но с разными типами возвращаемых значений.

При наличии перегруженных функций в программе компилятор выбирает соответствующую функцию в согласовании с типами аргументов и их количеством. Правило, по которому производится такой выбор, называется алгоритмом соответствия сигнатуре.

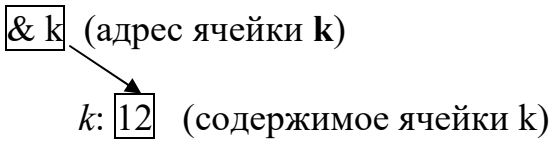
2.7 Указатели и ссылки в языке C++

Указатели в языке C++ используются для связи переменных с машинными адресами. Принято, что для некоторого типа **T** тип **T*** является «указателем на **T**», т. е. переменная типа **T*** содержит адрес объекта типа **T**.

Например:

```
int  k=12;
int* p= &k; // указатель p содержит адрес переменной k.
```

Основной операцией над указателями является операция получения значения, хранящегося по адресу, записанному в указателе



В данном случае **p** является указателем на целочисленное значение типа ***int***.

В языке C++ операция получения объекта, на который указывает указатель, называют ***разыменованием*** или ***косвенным обращением***. Оператором разыменования является ***префиксный унарный оператор*** *.

Например:

int **k**=12; // определение целочисленной переменной k
int* **p**=&**k** // **p** содержит ***адрес переменной*** **k** с начальным
// значением, хранящимся по адресу, записанному в **p**.

Итак, если **p** является указателем, то ***p** – значение переменной, на которую указывает **p**. Прямое значение **p** – это адрес памяти, тогда как ***p** – это значение, находящееся по адресу, записанному в **p**.

На основе указателей можно организовать эффективный вызов по ссылке при использовании функций. В данном случае адреса переменных используются в качестве аргументов функций, а в результате значения переменных могут изменяться в вызывающем окружении. Указатели используются в списке параметров для определения адресов переменных, значения которых могут изменяться. При обращении к функции адреса соответствующих переменных должны передаваться как аргументы.

```
// Пример организации вызова функции по ссылке на основе
// использования указателей.
// В программе вводятся два числа типа double, а в результате
// они выводятся по возрастанию
#include <iostream>
using namespace std;
void min_max (double* f1, double* f2)
{
double x;
if (*f1>*f2)  { x=*f1;  *f1= *f2; *f2=x;}
}
int main()
{     double d1,d2;
      cout << " Vvedite veschestvennue  d1 i d2 : " ;  cin >>  d1 >> d2;
cout << " Ishodnaya para chisel : "  << d1 << " ; " << d2 << endl;
min_max(&d1,&d2);
cout << "Rezultat : " << d1 << " ; " << d2 << endl;
```

```
return 0;
}
```

В программе внутри функции ***main()*** объявляются две вещественные переменные *d1,d2*. Затем на экран выводится запрос на ввод двух вещественных чисел, вводятся два числа (*cin >> d1 >> d2*) и происходит обращение к функции ***min_max***(). В качестве фактических значений аргументов передаются адреса *d1* и *d2*. Эти адреса передаются по значению. Отметим, что после вызова функции значение аргумента, соответствующее формальному параметру, используется в теле выполняемой функции.

В языке C++ такие параметры являются ***вызываемыми по значению***. Когда происходит вызов по значению, то переменные передаются функции как аргументы, их значения копируются в соответствующие параметры функции, а сами переменные не изменяются в вызывающем окружении. По своей сути, вызываемые по значению параметры являются локальными и им могут передаваться выражения, значения которых присваиваются этим локальным переменным.

Возвращаясь к нашему примеру, обратим внимание на то, что адреса *&d1* и *&d2* передаются по значению, а потому не могут быть изменены в вызывающем окружении. А вот значения *d1* и *d2* могут изменяться в вызывающем окружении. В конце программы выводится результат выполнения программы.

В языке C++, используя ***void,*** можно объявлять тип обобщенного указателя: ***void*** **ptr**. Такому указателю можно присваивать адрес переменной любого типа, однако он не может быть ***разыменован***, поскольку неизвестно, какого типа информация хранится по адресу указателя. Любой тип указателя может быть преобразован к обобщенному указателю типа ***void****.

Например:

```
void* ptr;      // объявление обобщенного указателя
float* fp;      // объявление указателя на float
bool* bp;       //  объявление указателя на  bool;
ptr=fp;//  допустимое преобразование
```

Для указателей допустимыми являются следующие преобразования: *нулевое значение указателя может быть преобразовано к любому типу; имя массива является указателем на его базовый адрес; тип* «*функция, возвращающая T*» *может быть преобразован в указатель на функцию, возвращающую T.*

В языке C++ предусмотрена возможность использования ***ссылок***. Объявление ссылки, которое является ее определением, обязательно должно содержать инициализатор.

Например:

```
double x;              // определяется вещественная переменная x
double& y=x;       // y – альтернативное имя для x
```

В данном примере имена ***x*** и ***y*** ссылаются на одно и то же значение в памяти, а потому изменение ***y*** равносильно изменению ***x*** и наоборот.

Использование ссылок позволяет применять более простую форму параметров при вызове функции по ссылке, когда аргументы применяются напрямую без разыменования:

```
// Программа с примером организации вызова функции по ссылке
// Вводятся два числа типа double,
// и в результате они выводятся по возрастанию
# include < iostream>
using namespace std;
void min_max (double& f1, double& f2)
{     double x;
      if  (f1>f2)
     { x=f1; f1=f2; f2=x;}
}
int main ()
{     double d1, d2;
      cout << " Vvedite veschestvennue  d1 i d2 : " ;  cin >>  d1 >> d2;
      cout << " Ishodnaya para chisel :  "  << d1 << " ;  " << d2 << endl;
      min_max(d1,d2);
      cout << "Rezultat :  " << d1 << " ;  " << d2 << endl;
      return 0;
}
```

Здесь в функции ***main()*** при вызове функции ***min_max()*** используются имена переменных, а не их адреса. Кроме того, упростилось внутреннее содержание функции ***min_max()***, поскольку отпала необходимость в разыменовании.

В программировании на языке С++ рекомендуют отдавать предпочтение ссылкам, а не указателям, поскольку ссылки проще использовать. Но следует помнить, что ссылки нельзя переназначать, а поэтому если в программе возникает необходимость сначала указывать на один объект, а затем на другой, то приходится использовать указатель.

Пример 9 – (*демонстрируется использование* ***указателя на целочисленное*** *значение*). *Рассмотрим пример программы, в которой объявляются* ***целочисленная переменная i*** *и* ***указатель на целочисленное значение iPointer.*** *Затем демонстрируется механизм изменения значение переменной* ***i*** *через использование указателя* ***iPointer***.

```
#include <iostream>
using namespace std;
int  main()
{// объявляем целочисленную переменную i и указатель iPointer
int i;    int *iPointer;
```

```
// инициализируем объявленные переменные
i = 225;        iPointer = &i;
// выводим значение переменной i
cout <<  “i=” << i << endl;
// далее изменим значение i через iPointer
*iPointer = -121;
// убедимся что переменная i изменила свое значение
// в результате предыдущего действия,
// выведя значение переменной ещё раз
cout <<  “i=” << i << endl;
return 0;
}
```

2.8 Массивы в языке C++

Под массивом в языке C++ подразумевается тип данных, который применяется для представления ***однородных*** (***однотипных***) ***значений***. Для обращения к элементам массива используются индексы, нумерация которых начинается с 0.

При объявлении массива указывается его размер в квадратных скобках после имени массива. Массивы и обычные переменные разрешается объявлять вместе, например:

double x1, mas[5], p1; //здесь объявлены вещественные переменные ***x1, p1***
// и пятиэлементный массив вещественных чисел
// ***mas[0],…, mas[4]***

По правилам языка индекс элемента массива не обязательно должен быть целой константой. Допустимо, чтобы он задавался выражением, значением которого являются числа от 0 до числа, на единицу меньше размера массива.

При использовании массивов необходимо четко следить за корректностью значений индексов элементов, т. к. выход за границы допустимых значений обычно вызывает ошибку выполнения (**!!! *в общем случае результат возникновения такой ошибки зависит от системы и непредсказуем***).

При объявлении массива можно инициализировать его элементы. В этом случае инициализирующие значения заключаются в фигурные скобки и разделяются запятыми, например,

double **m[3]={12.6,-14.1,7.05}; // m[0]=12.6 m[1]=-14.1…**

Если в инициализаторе задано меньше значений, чем количество элементов массива, то начальным элементам присваиваются указанные значения, а оставшиеся элементы массива инициализируются 0. Если же в инициализаторе значений больше, чем элементов массива, то фиксируется ошибка.

Неинициализированные глобальные и статические массивы автоматически получают нулевые значения для своих элементов. Для элементов локальных массивов начальные значения неопределенны.

Массиву, объявленному с явным списком инициализаторов, но без указания его размера, определяется размер, соответствующий количеству значений в инициализаторе:

int **m[ ]={2,16,-8}; // эквивалентно** ***int*** **m[3]={2,16,-8};**

Оператор ***sizeof***(**mas**) возвращает количество байт, занимаемое элементами массива в памяти.

2.8.1 Массив символов в языке С++.

В С++ можно создавать и обрабатывать массив символов. Рассмотрим пример программы, в которой демонстрируются некоторые особенности обработки символьного массива.

В рассмотренной программе функция ***kl_cifr()*** определяется перед функцией ***main()***, а функция ***kl_bukv()*** – после и потому в тексте программы перед определением функции размещен прототип функции ***kl_bukv().*** Следует обратить внимание на то, что в качестве параметра в функции ***kl_cifr()*** используется символьный массив, а в функции ***kl_bukv()*** – указатель на ***char***.

2.8.2 Массив строк в языке С++.

В языке С++ можно создавать и обрабатывать массив строк. Набор строк можно рассматривать как двухмерный массив символов, в котором каждая строка представляет собой одномерный массив символов. Например, если в программе написать *char s[4][7]*, то можно считать, что в памяти резервируется символьная матрица, содержащая 4 строки по 7 символов. Если далее предусмотреть ввод 4 слов, соответственно «**Язык**», «**Pascal**», «**C++**» и «**C#**», то в памяти получится следующая матрица (рисунок 7).

				0		
						0
			0			
		0				

Рисунок 7 – Матрица

Отметим, что каждая символьная строка заканчивается символом ‘\0’.

Рассмотрим пример программы, в которой демонстрируются основные особенности обработки массива строк.

В программе предусмотрены две функции – ***length()*** *и* ***main()****. Функция*

length() *в качестве параметра на вход получает указатель на начало строки и вычисляет ее длину.*

В начале выполнения программы резервируется место для размещения до max_kl_strok=4, каждая из которых может содержать до max_kl_stolb=7 символов. Далее с клавиатуры вводится количество строк в допустимом диапазоне. Затем в цикле вводятся соответствующие строки символов. В конце программы в цикле выводятся длины введенных строк.

```
#include <iostream>
using namespace std;
// Определение вспомогательных констант:
// max_kl_strok для резервирования максимального числа строк
// max_kl_stolb для резервирования максимального числа столбцов
const int max_kl_strok=4, max_kl_stolb=7;
// Определение функции length() для вычисления длины строки
// символов
int length(char* stroka)
{int kl=0;
  while (stroka[kl]!='\0') kl++;
return kl;
}
int main()
{
// Определение набора строк, каждая из которых может содержать
// до 6 полезных символов, а 7 – зарезервирован под '\0'
   char ns[max_kl_strok][max_kl_stolb];
// Цикл для обеспечения ввода корректного значения числа строк
   int kol_strk;
  do {
    cout << "Vvedite kolichestvo strok ne bolee " << max_kl_strok << " : ";
     cin >> kol_strk;
     if (kol_strk < 1 || kol_strk > max_kl_strok) cout << "ERROR !!!" << endl;
     }
  while (!(kol_strk > 0 && kol_strk <= max_kl_strok));
// Цикл для ввода строк
for(int i=0; i<kol_strk; i++)
  { cout << "Vvedite " << i << " stroku:";
    cin >> ns[i]; };
// Цикл для вывода длин строк
for(int i=0; i<kol_strk; i++)
          cout << "Dlina " << i << " stroki = " << length(ns[i]) << endl;
return 0;
}
```

Разберем, почему в результате выполнения последней программы на экране было получены отображаемые в консольном окне результаты (рисунок 8).

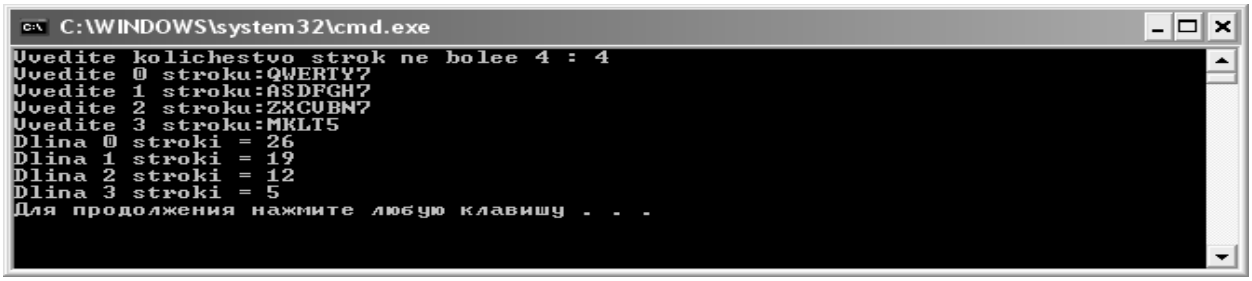


Рисунок 8 – Консольное окно с результатами

Первая вводимая строка содержит 7 символов «QWERTY7» и после нажатия клавиши <Enter> восьмым будет записан нулевой символ ‘\0’. После ввода следующей строки «ASDFGH7» этот символ будет замещен первой буквой, т. е. A. Соответственно начальный символ третьей строки Z заместит нулевой символ второй строки, а начальный символ четвертой строки M заместит нулевой символ третьей строки. В итоге при выводе длина первой строки будет вычисляться от начала строки и до тех пор, пока не встретиться нулевой символ, т. е. длина будет 7 + 7 + 7 + 5 = 26 и т. д. для других строк.

Рассмотрим пример программы, в которой демонстрируется использование генератора случайных чисел для формирования символьных строк.

В программе предусмотрено использование функции ***strlen(),*** *которая вычисляет длину строки и размещается в подключаемом файле string.*

В начале выполнения программы резервируется место для размещения до max_kl_strok=7, каждая из которых может содержать до max_kl_stolb=51 символов. Далее с использованием генератора случайных чисел (ГСЧ) формируется количество строк в диапазоне [1, max_kl_strok]. Затем в цикле с помощью ГСЧ формируются строки символов допустимой длины в диапазоне [1, max_kl_stolb]. В конце программы в цикле сначала выводятся сформированные строки, а затем также в цикле выводятся длины этих строк.

```
#include <iostream>
using namespace std;
#include <string>   // для подключения функции strlen()
#include <stdlib.h>// для подключения функцй srand() и rand()
#include <time.h>  // для подключения функции time()
const int max_kl_strok=7, max_kl_stolb=51;
int main()
{char matrix_char[max_kl_strok][max_kl_stolb];
  srand((unsigned) time(NULL)); // инициализация генератора ГСЧ
// Объявление целочисленной переменной для размещения
// количества строк и инициализация ее случайным значением
int kol_strk = rand()%max_kl_strok+1;
// Вложенные циклы для формирования набора строк с помощью ГСЧ
```

```
for(int kls,i=0; i<kol_strk; i++)
        { kls=rand()%max_kl_stolb+1;
          for(int j=0; j<kls; j++) matrix_char[i][j]=rand()%40+48;
          matrix_char[i][kls]='\0';  };
// Цикл для вывода сформированных строк
 for(int i=0; i<kol_strk; i++)
        cout << "Stroka " << i << " : " << matrix_char[i] << endl;
// Цикл для вывода длин строк
 cout << endl;
 for(int i=0; i<kol_strk; i++)
 cout << "Dlina " << i << " stroki = " << strlen(matrix_char[i]) << endl;
return 0;
}
```

2.9 Многомерные массивы в языке C++

В языке C++ допустимо создавать и обрабатывать многомерные массивы. Предусмотрена возможность инициализации таких массивов. При этом каждая строка задается своим списком в фигурных скобках. Например, двухмерный массив (матрицу) целых чисел можно определить следующим образом:

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

Рассмотрим пример программы, в которой *вводится с клавиатуры квадратная матрица целочисленных значений размерности* ***n x n*** *(где* ***n****<=9). В начале программы вводится размерность матрицы, затем предусматриваются два вложенных цикла для ввода значений элементов матрицы с клавиатуры. Далее матрица выводится на экран и в цикле предусматривается вывод значений сумм элементов по строкам. В программе для вычисления суммы элементов строки матрицы предусмотрена функция* ***sum_row()****.*

```
# include <iostream>
using namespace std;
const int razm=9;
int n,int_matr[razm][razm];
// Функция  sum_row() на вход получает номер строки
// целочисленной матрицы int_matr  и вычисляет
// сумму элементов этой строки матрицы
int sum_row(int k)
{int sum=0;
for(int j=0; j<n; j++) sum+=int_matr[k][j];
return sum;
}
int main()
{int i,j;
```

```
//Ввод числа строк и столбцов в квадратной матрице
do {      cout << "Vvedite razmernost matritsy n<=" << razm << " -> ";
cin >> n;}
  while (n<=0 || n>razm);
// Для построчного ввода элементов матрицы с клавиатуры можно
// использовать вложенные циклы for
  for(i=0; i<n; i++)                  // внешний цикл
   for(j=0; j<n; j++)                 // вложенный цикл
{
  cout << "Vvedite znachenie elementa matr[" << i << "]["
  << j << "] -> ";  cin >> int_matr[i][j];
}
// Вывод двухмерного массива на экран в матричном виде
  cout << endl << "Iskhodnaya matrix :\n";
  for(i=0; i<n; i++) {
                  for(j=0; j<n; j++)  cout << '\t' << int_matr[i][j];
cout << endl;         }
// Цикл для вывода значений сумм элементов в строках матрицы
for(j=0; j<n; j++) cout << "V stroke " << j << " summa = "
<< sum_row(j) << endl;
return 0;
}
```

Отметим, что в рассмотренной программе квадратная матрица *int_matr* и ее размерность *n* определены как ***глобальные*** переменные, а потому они «видны» (доступны) и из функции *main(),* и из функции *sum_row()*.

Теперь остановимся на особенностях представления массивов в языке С++. Имя массива само по себе является ***адресом*** или ***значением указателя***, а потому массивы и указатели очень похожи по организации доступа к памяти. Различие заключается в том, что ***указатель*** – это переменная, принимающая в качестве значения адрес, а ***имя массива*** представляет конкретный фиксированный адрес, который называют ***базовым адресом***.

Таким образом, ***базовый адрес массива можно считать постоянным указателем на нулевой элемент массива***.

Функция может иметь формальный параметр типа массив, и при обращении к функции соответствующим аргументом будет весь массив. Однако формальный параметр для массива передается не по значению и не по ссылке, а реализован новый вид формального параметра, называемый ***параметром массивом***.

При подстановке аргумента массива вместо параметра массива, функции передается только адрес первого элемента с индексом 0. Аргумент массива не передает функции значение размера массива, а потому необходимо предусматривать еще один формальный целочисленный параметр, указывающий размер массива. Таким образом, в языке С++ функция с ***формальным параметром***

массивом может вызываться с ***фактическим аргументом массивом*** любого размера при условии, что массив имеет правильный базовый тип.

При использовании аргумента массива следует учитывать то, что если в теле функции значение параметра массива изменяется, то при обращении к функции будет изменено значение аргумента массива, т. е. функция может изменять значения элементов массива.

Иногда возникает ситуация, когда необходимо исключить возможность нежелательных изменений внутри функции, и тогда перед параметром массива нужно писать модификатор ***const***. Параметр массива, объявленный с модификатором ***const***, называется ***константным параметром массива***.

Пример 10

```
int sum (const int a[n], int k) // здесь k – размер массива
 {
int s=0;
for (int i=0; i<k; ++i) s=s+a[i];
return s;
 }
```

Отметим, что, если предлагается использование модификатора ***const***, то его необходимо указывать и в прототипе, и в определении функции. В общем случае модификатор ***const*** может использоваться с любыми параметрами, однако обычно он используется с параметрами массивов и передаваемыми по ссылке параметрами типа класса.

Рассмотрим пример программы, в которой *формируется с помощью генератора случайных чисел прямоугольная матрица вещественных значений размерности* ***n x m*** *(где* ***n***<=6, ***m***<=5). *В начале программы вводятся размерности матрицы, затем предусматриваются два вложенных цикла для формирования значений элементов матрицы в диапазоне [0,1] с помощью ГСЧ. Далее матрица выводится на экран и в цикле предусматривается вывод значений сумм элементов и средних арифметических по столбцам. В программе для вычисления суммы элементов столбца матрицы предусмотрена функция* ***summa_stolb()****, а для вычисления среднего арифметического –* ***sarifm_stolb()***.

```
#include <iostream>
using namespace std;
#include <stdlib.h>
#include <time.h>
const int str=6,stolb=5;
// Функция  summa_stolb() на вход получает 3 параметра:
// 1) матрицу, 2)количество строк в ней, 3)номер столбца
// вещественной матрицы dbl_matr и вычисляет сумму
// элементов в этом столбце матрицы
double summa_stolb(double m[str][stolb], int nstr, int k)
```

```
{double sum=0.0;
 for(int j=0; j<nstr; j++) sum+=m[j][k];
return sum;
}
// Функция  sarifm_stolb() на вход получает 3 параметра:
// 1) матрицу, 2)количество строк в ней, 3)номер столбца
// вещественной матрицы dbl_matr и вычисляет среднее
// арифметическое элементов в этом столбце матрицы
double sarifm_stolb(double m[str][stolb], int nstr, int k)
{
 return summa_stolb(m,nstr,k)/nstr;
}
int main()
{int n,m;       double dbl_matr[str][stolb];
 do{ cout << "Vvedite razmernost matritsy: \n Kolichestvo strok n<="
         << str << " -> "; cin >> n;     //Ввод числа строк в матрице
         cout << "Kolichestvo stolbtsov m<="
                 << stolb << " -> "; cin >> m;}
 while ((n<=0 || n>str) || (m<=0 || m>stolb));
// Для формирования по столбцам элементов матрицы
// с помощью ГСЧ воспользуемся вложенными циклами for
 srand((unsigned)time(NULL));
for(int j=0; j<m; j++)              // внешний цикл
         for(int i=0; i<n; i++)             // вложенный цикл
                 dbl_matr[i][j]=(double)rand()/RAND_MAX;
// Вывод двухмерного массива на экран в матричном виде
 cout << endl << "Iskhodnaya matrix :\n";
 for(int i=0; i<n; i++)     {
         for(int j=0; j<m; j++)  cout << '\t' << dbl_matr[i][j];
         cout << endl;       }
// Цикл для вывода эначений суммы элементов и средних
// арифметических в столбцах матрицы
 for(int j=0; j<m; j++) cout << "V stolbtse " << j
                 << " summa elementov = " << summa_stolb(dbl_matr,n,j)
                  << " srednee arifm. = " << sarifm_stolb(dbl_matr,n,j)
                  << endl;
return 0;
}
```

2.10 Структуры

Предположим, что необходимо сохранять информацию об автомобилях, учитывая следующие данные:

1) марка автомобиля (*15 символов*);

2) цена в долларах (*целое число*);

3) расход топлива на 100 км пробега (*вещественное с 1 знаком после десятичной точки*).

Отметим, что массив в данном случае использовать не удастся, поскольку, хотя массив и может содержать несколько элементов, все элементы его должны быть **однотипными**.

В языке С++ предусмотрен **агрегатный** тип ***struct***, используя который можно объединить разнотипные компоненты в переменную с одним именем. Все поименованные компоненты **структуры** принято называть ***членами***.

В общем случае структура объявляется с помощью ключевого слова ***struct***, за которым обычно записывается имя типа (*теговое имя*), а затем в фигурных скобках идет список объявлений членов структуры.

Допускается, чтобы теговое имя отсутствовало, и тогда объявление является безымянным. Оно может использоваться только для немедленного объявления переменных соответствующего типа.

Например: ***struct { int a; double x; } massiv [2]={ {5,6.2},{-7,8.9} };***

Пример объявления структурного типа:

```
struct  avto {                    // информация по автомобилям
char  marka [15];
int    cena;
double  rashod;
};
```

Отметим, что при объявлении структуры обязательно необходимо писать ; (точка с запятой) после закрывающейся фигурной скобки.

Переменные типа ***struct*** можно объявлять как и обычные переменные, а к конкретным членам можно обращаться используя оператор **.** (точка).

Например:

```
avto x1,x2;
x1. cena=22100;     // цена в $
x2. rashod= 6.4;    // расход бензина на 100 км. пробега
```

Доступ к членам структуры можно производить и через использование оператора указателя структуры -> (двухсимвольный). Если переменной-указателю присвоен адрес структуры, то доступ к члену структуры может быть осуществлен с помощью конструкций вида ***p->m***, где ***p*** - указатель, ***m*** - имя члена-переменной. Отметим, что запись ***p->m*** эквивалентна ***(*p).m.*** Переменные типа структуры можно присваивать, передавать в качестве аргументов и возвращать в качестве значений функций.

С помощью оператора ***sizeof*** можно узнавать размер объекта типа ***struct***. Следует учитывать, что значение размера может несколько превышать суммарное значение длин входящих в структуру компонентов, т. к. для некоторых типов предусматривается выравнивание на границу аппаратного слова.

```
// Пример программы, в которой определяется структурный тип
// avto, содержащий три поля. Демонстрируются способы обращения
// к полям и механизм передачи значений структурного типа при
// использовании функций.
#include <iostream>
using namespace std;
struct avto {
                char marka[15];
                int cena;
                double rashod;
                };
avto vvod_info()
{avto x;
  cout << "Vvedite marku avtomobilya :"; cin >> x.marka;
  cout << "Vvedite cenu avtomobilya :"; cin >> x.cena;
  cout << "Vvedite rashod na 100 km :"; cin >> x.rashod;
return x;
}
void VYVOD(avto s)
{cout << "MARKA AVTO :" << s.marka << endl;
  cout << "CENA AVTO :" << s.cena << endl;
  cout << "RASHOD NA 100 km. :" << s.rashod << endl;
  cout << "_____________" << endl;
}
int main()
{avto A1;
  A1=vvod_info();
  VYVOD(A1);
  avto* A2;
  A2=&(vvod_info()); // !!! Так можно
  cout << "MARKA AVTO :" << A2->marka << endl;
  cout << "CENA AVTO :" << (*A2).cena << endl;
  cout << "RASHOD NA 100 km. :" << A2->rashod << endl;
  cout << "_____________" << endl;
  cout << "RAZMER A1 = " << sizeof(A1) << endl;
  cout << "RAZMER A2 = " << sizeof(A2) << endl;
return 0;
}
```

В языке C++ не разрешается использовать имя структуры для объявления других объектов в том случае пока сама структура полностью не объявлена. Например, недопустимо следующее рекурсивное определение:

```
struct Demo_1 {
...;
```

```
Demo_1 component_k;
...
                    };
```

В данном случае компилятор не может определить размер типа ***Demo_1***, а потому фиксируется ошибка.

В том случае, когда два или более объектов типа ***struct*** ссылались друг на друга, разрешается использовать предварительное объявление только имени типа.

Например:

```
// В примере сначала объявляется структурный тип  st_1 (см.3
// строку текста программы ниже).  При объявлении типа st_2
// допускается использование указателя st_1*, но недопустимо
// использование просто st_1.
#include <iostream>
using namespace std;
int main()
{
struct st_1; // это только объявление типа
struct st_2 { int h1; st_1* h2; double h3; };
struct st_1 {char* c1; float c2; st_2 c3; };
st_1 x;
x.c1="0123456789"; x.c3.h1=-1245; x.c2=-12.75;
cout << x.c1 << endl << x.c2 << endl << x.c3.h1 << endl;
st_2 y;
y.h2=new st_1; // необходимое действие
y.h1=9876;  y.h2->c1="9876543210";  y.h3=-78.77;
cout << y.h2->c1 << endl << y.h1 << endl<< y.h3 << endl;
return 0;
}
```

Отметим, что при определении структурного типа ***st_1*** первая компонента ***c1*** объявлена как имеющая тип ***char****, что позволяет передавать строку символов.

При использовании структур необходимо учитывать, что две структуры являются разными типами, даже если у них одинаковые члены.

Например:

```
struct  st_1 { int i;};
struct  st_2 { int i;}; // здесь st_1 и st_2 – разные типы
st_1 comp1;
st_2 comp2=comp1; // ошибка несоответствия типов
```

2.11 Массивы структур

В языке С++ можно создавать массивы структур. Отметим, что в предыдущем параграфе было показано, что поле структуры может быть массивом. Сейчас же рассмотрим пример программы, в которой создается и обрабатывается массив структур.

```
// Пример программы, в которой определяется структурный тип
// avto, содержащий три поля. На основе типа avto создается
// и инициализируется одномерный массив из трех элементов.
// Демонстрируются механизм обращения к полям элементов
// массива и механизм передачи значений структурного типа
// при использовании функции.
#include <iostream>
using namespace std;
struct avto {  char marka[15];
               int cena;
               double rashod;
               };
void VYVOD(avto s)
{cout << "MARKA AVTO :" << s.marka << endl;
 cout << "CENA AVTO :" << s.cena << endl;
 cout << "RASHOD NA 100 km. :" << s.rashod << endl;
 cout << "_____________" << endl;
}
int main()
{avto x[3] = {       {"Ford", 18400, 8.6},
                    {"Audi", 23500, 7.5},
                    {"Renault", 21300, 8.0}
                 };
 for (int i=0; i<3; i++) VYVOD(x[i]);

 cout << "RAZMER MASSIVA x = " << sizeof(x) << endl;
 cout << "RAZMER ELEMENTA x[0] = " << sizeof(x[0]) << endl;
 cout << "RAZMER x[1].marka = " << sizeof(x[1].marka)
<< endl;
 cout << "Cena avtomobilya " << x[2].marka << " = "
<< x[2].cena << endl;
 cout << "Poslednij simvol v nazvanii marki poslednego"
         "avtomobilya = " << x[2].marka[6] << endl;
return 0;
}
```

2.12 Понятие объединения union

В языке C++ можно создавать ***объединения*** **(*union*)**, которые фактически представляют собой подвид ***структур***. Так же, как и ***структура***, ***объединение*** состоит из полей, но отличительной чертой ***union*** является то, что это ***объединение*** занимает в памяти столько места, сколько необходимо для размещения наиболее объемного из всех полей. В каждый конкретный момент времени объединение может хранить значение только одного из членов.

```
// Пример программы, в которой определяется объединение
// Type_Cena и структурный тип avto, содержащий три поля (одно
// из которых типа union). В программе определяется массив из
// трех элементов. Вводятся три записи, и далее демонстрируются
// механизм обращения  к полям элементов массива и механизм
// передачи значений структурного типа при использовании
// функции.
#include <iostream>
using namespace std;

union Type_Cena { int icena;
                  float fcena;  };

struct avto {   char marka[15];
                  Type_Cena cena;
                  double rashod;      };
void VYVOD(avto s, int i)
{cout << "MARKA AVTO :" << s.marka << endl;
  if (i<2) cout << "CENA AVTO :" << s.cena.icena << endl;
         else cout << "CENA AVTO :" << s.cena.fcena << endl;
  cout << "RASHOD NA 100 km. :" << s.rashod << endl;
  cout << "_______________" << endl;
}

int main()
{avto A1[3];
cin >> A1[0].marka >> A1[0].cena.icena >> A1[0].rashod;
cin >> A1[1].marka >> A1[1].cena.icena >> A1[1].rashod;
cin >> A1[2].marka >> A1[2].cena.fcena >> A1[2].rashod;
for (int i=0; i<3; i++) VYVOD(A1[i],i);
  cout << "RAZMER MASSIVA A1 = " << sizeof(A1) << endl;
  cout << "RAZMER ELEMENTA A1[0] = " << sizeof(A1[0]) << endl;
  cout << "RAZMER ELEMENTA A1[2] = " << sizeof(A1[2]) << endl;
  cout << "Cena" << A1[1].marka << " = "<< A1[1].cena.icena << endl;
  cout << "Cena" << A1[2].marka << " = " << A1[2].cena.fcena << endl;
return 0;
}
```

2.13 Понятие потоков в языке C++

Для обмена данными между программой и внешними устройствами в языках программирования предусматриваются операции ***ввода-вывода***. ***Типичным внешним устройством ввода-вывода*** является ***консоль***, представляющая собой комплект устройств (дисплей, клавиатура, мышь), присоединённых к компьютеру непосредственно, а не через сеть. На консоль можно вывести информацию, можно ввести информацию с клавиатуры или мыши.

Другими типичными устройствами ввода-вывода являются жесткий или гибкий диск, на котором расположены файлы. Во время выполнения программы можно создавать файлы, в которых будет храниться информация, а другая или эта же программа может читать информацию из этого файла.

В языке C++ нет специальных операторов для выполнения операций ввода или вывода данных. Вместо этого имеется стандартная библиотека ***<iostream>***, стандартно поставляемая вместе с компилятором, с помощью классов которой и реализуются основные операции ввода-вывода (*I/O – Input and Output*). Такой подход обеспечил эффект "***независимости от платформы***".

Механизм для обеспечения выполнения операций ввода-вывода в C++ называется ***потоком***. Название произошло оттого, что данные вводятся и выводятся в виде ***потока*** байтов.

Операции «*запись данных на внешнее устройство* (*диск*)» и «*считывание их с внешнего устройства* (*диска*)» являются относительно медленными, а потому могут существенно притормаживать выполнение программы. Для повышения скорости обмена данными предусматривается механизм ***буферизации***. В этом случае данные сначала записываются в ***буфер потока***, а после его наполнения все содержимое записывается на диск.

Реализация потоков и буферов в языке C++ построена на объектно-ориентированном подходе:

– класс ***streambuf*** управляет буфером (с помощью соответствующих методов буфер может, например, наполняться, очищаться, сбрасываться);

– класс ***ios*** является базовым для классов потоков ввода/вывода и внутри него в качестве переменной-члена предусмотрен объект ***streambuf***;

– классы ***istream*** и ***ostream*** являются производными от класса ***ios*** и используются, соответственно, для обеспечения потокового ввода и вывода данных;

– класс ***iostream*** является производным от классов ***istream*** и ***ostream*** и используется для обеспечения ввода с клавиатуры и вывода на экран;

– класс ***fstream*** используются для операций ввода и вывода из файлов.

Классы библиотеки ***<iostream>*** рассматривают данные, выводимые на экран, как побитовый ***поток*** данных. Когда данные выводятся (*записываются*) в файл или на экран дисплея, то источник потока содержится в программе. Если же поток вводится в программу из внешних источников, то данные могут поступать с клавиатуры или файла на диске, и они заносятся в переменные.

Класс ***istream*** реализует поток ввода, класс ***ostream*** – поток вывода. Эти классы определены в библиотеке <***iostream***>. Библиотека потоков ввода-вывода определяет три глобальных объекта: ***cin***, ***cout***, и ***cerr***. Объект ***cin*** отвечает за ввод данных, объект ***cout*** является стандартным выводом, а ***cerr*** является потоком сообщений об ошибках. Объекты ***cout*** и ***cerr*** являются экземплярами класса ***ostream***, а ***cin*** – объект класса ***istream***.

2.14 Ввод данных с помощью объекта cin в языке C++

Ввод данных с помощью объекта ***cin*** может осуществляться с использованием оператора >>. Например, для ввода целочисленного значения с клавиатуры можно записать следующий программный фрагмент:

… int **n;** **cin >> n; …**

Объект ***cin*** включает перегруженный оператор ввода >>, который записывает данные из буфера в локальную переменную **n.** Оператор ввода перегружен таким образом, что предусматривает ввод данных основных базовых типов, включая int&, short&, long&, double&, float&, char&, char*. Когда в программе компилятор встречает выражение **cin >>** …, то вызывается вариант оператора ввода, соответствующий типу переменной. Поскольку параметр передается как ссылка, то оператор ввода способен изменять исходную переменную.

Объект ***cin*** может принимать в качестве аргумента указатель на строку символов типа ***char****, а потому можно с его помощью вводить массив символов, т. е. строку. Рассмотрим фрагмент кода:

```
… char STROKA[9];
 cout << “Введите строку:”;
 cin >> STROKA;  …
```

Если ввести слово ЯЗЫК, то массив STROKA будет заполнен слева 5 символами: Я, З, Ы, К, \0. Если же попробовать ввести фразу ЯЗЫК С++, то окажется, что введены будут только символы, стоящие до *пробела*, т. к. *пробел* выступает в роли заданного по умолчанию разделителя строк.

Отметим, что в языке С++, кроме перегружаемого оператора, >> объект ***cin*** имеет ряд встроенных методов (getline(), get() и др.), которые позволяют обеспечивать более строгий контроль при выполнении ввода данных.

Например, для ввода строки символов можно использовать метод getline(). Рассмотрим пример простой программы, в которой определяются два массива типа ***char*** и ввод фразы ЯЗЫК С++ осуществляется с использованием метода getline()и оператора >>.

```
#include <iostream>
using namespace std;
```

```
int main()
{char STROKA_1[9], STROKA_2[9];

cout << "Vvedite 1 stroku:"; cin.getline(STROKA_1,9);
  cout << "Vvedite 2 stroku:"; cin >> STROKA_2;
  cout << endl << "STROKA_1:" << STROKA_1;
  cout << endl << "STROKA_2:" << STROKA_2 << endl;
return 0;
}
```

Отметим, что использование метода getline()позволило ввести полностью всю фразу, тогда как оператор >> обеспечил только ввод части фразы до первого слева *пробела*.

При последовательном комбинированном вызове методов get(), getline() и оператора >> необходимо следить за правильным использованием буфера ввода.

2.15 Вывод данных с помощью объекта cout в языке C++

Вывод данных с помощью объекта ***cout*** может осуществляться с использованием оператора <<. Причем этот объект позволяет осуществлять:

- *форматирование выводимых данных;*
- *выравнивание столбцов;*
- *вывод числовых значений не только в десятичном, но в восьмеричном и шестнадцатеричном представлении.*

По умолчанию ширина поля вывода устанавливается автоматически таким образом, чтобы вместить все символы из буфера вывода. Используя метод ***width()***, можно точно определять значение ширины ***только следующего*** поля вывода. Отметим, что объект ***cout*** по умолчанию заполняет пробелами пустые позиции поля, однако можно с помощью метода ***fill()*** установить другой символ заполнения.

```
#include <iostream>
using namespace std;

int main()
{
  cout << "->"; cout << 123456 << ';' << endl;
  cout << "->"; cout.width(9);   cout << 789 << ';' << endl;
  cout << "->"; cout.width(3);   cout << 123456 << ';' << endl;
  cout << "->"; cout << 123456 << ';' << endl;
  cout << "->"; cout.width(9); cout.fill('*');
  cout << 789 << ';' << endl;
  cout << "->";   cout.width(19); cout << 123456 << ';' << endl;
return 0;
}
```

В языке C++ предусмотрен набор манипуляторов для управления правилами выполнения операций вывода. Примером такого манипулятора является ***endl***, который вставляет символ новой строки и очищает буфер вывода. Отметим, что для использования части манипуляторов в программе необходимо предусматривать подключение файла ***<iomanip>.***

Набор манипуляторов, не требующих включения ***iomanip*** (таблица 5).

Таблица 5 – Набор манипуляторов, не требующих включения **iomanip**

Манипулятор	Действие
flush	Очищает буфер вывода
endl	Вставляет символ новой строки и очищает буфер вывода
oct	Устанавливает восьмеричное основание для выводимых чисел
dec	Устанавливает десятичное основание для выводимых чисел
hex	Устанавливает шестнадцатеричное основание для вывода чисел
left	Выравнивает выводимое значение по левому краю поля
right	Выравнивает выводимое значение по правому краю поля
showpos	Добавляет знак + перед положительным десятичным числом
noshowpos	Отменяет вывод + перед положительным десятичным числом
uppercase	Отображает шестнадцатеричные и экспоненциальные значения в верхнем регистре
nouppercase	Отображает шестнадцатеричные и экспоненциальные значения в нижнем регистре
scientific	Отображает числа с плавающей запятой в экспоненциальном представлении
fixed	Отображает числа с фиксированной запятой

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.       cout << showpos << "10 -> 8 : ";
6.       cout << 123456 << '=' << oct << 123456 << ';' << endl;
7.       cout << "8 -> 16 : ";
8.       cout << 123456 << '=' << hex << 123456 << ';' << endl;
9.       cout << "16 -> 8 : " << uppercase;
10.      cout << 123456 << '=' << oct << 123456 << ";\n" << endl;

11.      cout << "->";  cout.width(20); cout << 789 << ';' << endl;
12.      cout << "->";   cout << left << 789 << ';' << dec << endl;
13.      cout << "->" ;  cout.width(20);
14.      cout << 789 << ';' << endl;
15.      cout << "->" << noshowpos;   cout.width(20);
16.      cout << right << scientific << nouppercase << 789.3
<< ';' << endl;
17.      cout << "->" << left << fixed << 3.4567e+2 << ';' << endl;
```

```
18.     return 0;
19.     }
```

В строке 5 модификатор **showpos** включает режим отображения знака + перед выводом положительных десятичных чисел, а потому при выводе числа 123456 перед ним появится + (см. 6 строку). Модификатор **oct** (см. 6 строку) устанавливает восьмеричное основание для выводимых чисел, а **hex** – шестнадцатеричное основание (см. 8 строку).

В строке 9 модификатор **uppercase** включает режим вывода, при котором латинские буквы в шестнадцатеричных числах и в числах с плавающей запятой будут прописными (*большими*).

В 11 строке устанавливается ширина поля вывода 20 (**cout.width(20);**) и далее десятичное число 789 выводится в виде восьмеричного 1425 с выравниванием по правому краю.

В 12 строке модификатором **left** устанавливается режим выравнивания по левому краю, а модификатор **dec** вызывает переход к десятичному основанию для вывода целых чисел.

В 15 строке модификатор **noshowpos** подавляет вывод знака + перед положительными числами.

В 16 строке модификатором **right** устанавливается режим выравнивания по правому краю, модификатор **scientific** включает режим вывода вещественных значений в формате с плавающей точкой, а **nouppercase** устанавливает вывод латинских букв в шестнадцатеричных числах и в числах с плавающей запятой будут строчными (*малыми*).

В 17 строке модификатор **fixed** определяет режим вывода вещественных значений с фиксированной точкой.

Набор манипуляторов, для которых требуется подключение ***iomanip*** (таблица 6).

Таблица 6 – Набор манипуляторов, требующих включения **iomanip**

Манипулятор	Действие
setw()	Устанавливает ширину поля вывода
setprecision()	Точность вывода – количество цифр в дробной части
setfill()	Определяет символ заполнения при выводе

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
  cout << showpos << setw(15) << setfill('*') << 12.12345 << ';' << endl;
  cout << noshowpos << scientific << uppercase << setfill(' ')
  << setw(15) << setprecision(4) << 12.12345 << ';' << endl;
cout << setw(15) << left << fixed << 3.12345e+2 << ';' << endl;
```

```
return 0;
}
```

2.16 Файловые потоки в языке C++

Под ***файлом*** подразумевается поименованная совокупность данных на внешнем носителе, например, на винчестере. Логически файл может рассматриваться как *конечная последовательность байтов*, а потому такие устройства, как *дисплей*, *клавиатуру* и *принтер,* также можно рассматривать как частные случаи файлов.

По способу доступа файлы можно разделить на последовательные, чтение и запись в которых производятся с начала байт за байтом, и файлы с произвольным доступом, допускающие чтение и запись в указанную позицию.

В стандартной библиотеке <***fstream***> языка C++ предусмотрены три класса для работы с файлами:

1) ***ifstream*** – класс входных файловых потоков;
2) ***ofstream*** – класс выходных файловых потоков;
3) ***fstream*** – класс двунаправленных файловых потоков.

Эти три класса являются производными от классов ***istream***, ***ostream*** и ***iostream*** соответственно, поэтому они наследуют перегруженные операции << и >>, флаги форматирования, манипуляторы, методы.

При работе с файлами в программе могут выполняться следующие операции:

- *создание потока;*
- *открытие потока и связывание его с файлом;*
- *ввод/вывод данных;*
- *удаление потока;*
- *закрытие файла.*

2.17 Основы работы с текстовыми файлами в языке C++

Текстовый файл в общем случае содержит последовательность строк, каждая из которых завершается специальным символом – *признаком конца строки*. За последней строкой размещается специальный признак конца файла – ***e****nd_****o****f_****f****ile*.

Рассмотрим пример программы, в которой создается текстовый файл, имя которого вводится с клавиатуры. В этот файл записываются девять строк, и он закрывается. Далее из созданного файла читаются строки и выводятся на экран дисплея.

```
1. #include <fstream>
2. #include <iostream>
3. using namespace std;
4. int main()
5. {     char FileName[20];
6.       char X[10];
```

```
7.      cout << "Vvedite imya FILE : "; cin >> FileName;
8.      ofstream F(FileName);
9.      for (int i=0; i<3; i++)
10.       { cout << "Vvedite " << i << " stroku : ";  cin >> X;
11.             F << X << endl << i << endl << 1.0/(i+1) << endl; };
12.     F.close();
13.     cout << endl << "FILE " << FileName << " : \n";
14.     ifstream P(FileName);
15.     while (! P.eof())
16.     {
17.           P >> X;
18.           cout << X << endl;
19.     }
20.     P.close();
21. return 0;
22. }
```

В строке 1 подключается библиотека <***fstream***>, внутри которой определены три класса: ***ifstream***, ***ofstream*** и ***fstream*** для работы с файлами.

В строке 5 определяется переменная FileName для размещения имени текстового файла на внешнем устройстве.

В строке 6 определяется переменная X, которая фактически будет играть роль логического буфера для обеспечения выполнения операций ввода-вывода при работе с файлами.

В строке 7 выдается запрос на ввод имени файла, и осуществляется ввод имени с клавиатуры.

В строке 8 создается объект F класса ofstream, который ассоциируется с текстовым файлом на внешнем устройстве. Соответствующее имя файла на внешнем устройстве содержится в переменной FileName.

В строках 9–11 реализован цикл, при выполнении каждого из три шагов которого вводится строка символов, и далее в файл записывается эта введенная строка, номер шага цикла и вещественное значение, вычисленное по формуле 1.0/(i+1).

В строке 12 закрывается файл путем вызова метода ***close()***. Отметим, что когда файл закрывается, то все данные, которые программа писала в него, сбрасываются на диск, и обновляется запись каталога для этого файла. В результате в текущей папке создается текстовый файл.

В строке 14 создается объект P класса ifstream, который ассоциируется с текстовым файлом на внешнем устройстве, имя которого задается переменной FileName.

В строках с 15 по 19 включительно организован цикл, в котором считываются все строки файла и выводятся на экран. Цикл завершается при достижении признака конца файла ***eof***. В строке 20 закрывается файл. Отметим, что при достижении конца программы автоматически закрываются все открытые в программе файлы.

Если в рассмотренной программе заменить 8 строку **ofstream F(FileName);** на строку **ofstream F(FileName, ios::app);**, то при первом запуске программы будет создаваться текстовый файл, а при следующем обращении к уже существующему файлу будут добавляться новые строки.

Если в рассмотренной программе заменить строки: 8. **ofstream F(FileName);** на строку **fstream F(FileName);** 14. **ifstream P(FileName);** на строку **fstream P(FileName, ios::in);** то программа будет работать.

Если в рассмотренной программе заменить 8 строку **ofstream F(FileName);** на строку **ofstream F(FileName, ios::binary);**, то при запуске программы будет создаваться текстовый файл, содержащий 1 строку.

В рассмотренной выше программе были использованы два потока **F** и **P**. Приведем пример программы, которая вырабатывает абсолютно тот же результат, но при этом мы обойдемся только одним потоком, используя метод ***open()*** – *открыть файл*.

```
1. #include <fstream>
2. #include <iostream>
3. using namespace std;
4. int main()
5. {     char FileName[20];
6.       char X[10];
7.       cout << "Vvedite imya FILE : "; cin >> FileName;
8.       fstream F;
9.       F.open(FileName,ios::out);
10.      for (int i=0; i<3; i++)
11.        { cout << "Vvedite " << i << " stroku : ";  cin >> X;
12.              F << X << endl << i << endl << 1.0/(i+1) << endl; };
13.      F.close();
14.      cout << endl << "FILE " << FileName << " : \n";
15.      F.open(FileName, ios::in);
16.      while (! F.eof())
17.      {
18.            F >> X;
19.            cout << X << endl;
20.      }
21.      F.close();
22.  return 0;
23.  }
```

В строке 8 создается поток **F** типа **fstream**. В строке 9 с помощью метода **open(FileName,ios::out)** поток **F** связывается с текстовым файлом на внешнем устройстве и устанавливается режим *записи данных* в этот файл. В 10–12 строках в цикле формируется 9 строк текстового файла, а в 13 строке он закрывается.

В 15 строке поток **F** связывается с текстовым файлом на внешнем устройстве и устанавливается режим *чтения данных* из этого файла.

В 16–20 строках программы организован цикл, в котором последовательно (до тех пор, пока не будет прочитан признак конца файла ***eof***) происходит считывание строк и их вывод на консоль. Затем в 21 строке файл закрывается и далее программа завершается.

2.18 Обработка текстовых файлов на основе встроенных методов

Текстовый файл можно сформировать и прочитать посимвольно, используя для этого методы ***put()*** и ***get()*** соответственно для *записи символа в файл* и *чтения символа из файла*.

В текстовый файл можно записать и можно из него прочитать данные, используя для этого методы соответственно ***write()*** и ***read()***.

2.19 Основы работы с двоичными файлами в языке C++

Сохранять данные в файле в языке C++ можно не только в текстовом представлении, но и в двоичном формате. Текстовая форма означает, что все данные, включая числовые, сохраняются в виде ***текста***, который можно легко просмотреть с помощью обычного текстового редактора.

Например, сохранение целочисленного значения 1234567890 в текстовом виде означает сохранение 10 символов-цифр, из которых состоит данное число. Если же сохранить это число в ***двоичном формате***, то оно будет размещаться во внутреннем представлении, т. е. вместо символов сохраняется 32-битное представление числа типа ***int***.

Для значения типа ***char*** двоичное представление совпадает с его текстовым – двоичным представлением (например, ASCII-код символа). Для чисел же двоичное представление кардинально отличается от их текстового представления.

У каждого формата хранения данных в файле имеются свои достоинства. Так, текстовые файлы, во-первых, легко читать и редактировать с помощью обычного текстового редактора, во-вторых, можно относительно легко переносить с одной системы на другую.

В двоичном формате вещественные числа сохраняются более точно, поскольку они сохраняются во внутреннем представлении. Обычно сохранение данных в двоичном формате происходит быстрее, поскольку при этом не выполняются соответствующие преобразования, и требуется меньше пространства для хранения (в зависимости от природы данных). Однако при переносе данных с одной систему на другую могут возникать проблемы, в случае если новая система использует другое внутреннее представление данных. При такой ситуации необходимо будет писать программу, которая будет обеспечивать требуемые преобразования данных.

Чтобы сохранить данные в двоичном файле можно воспользоваться методом ***write().*** Этот метод копирует определенное число байтов из памяти в файл, причем он копирует любой тип данных байт в байт, не производя преобразования. Например, если ему передать адрес переменной типа ***int*** и указать скопировать 4 байта, то данный метод скопирует 4 байта, передав значение типа ***int*** и не производя его преобразования.

Чтобы прочесть информацию из двоичного файла нужно воспользоваться методом ***read()*** класса ***ifstream***.

3 Примерный перечень вопросов для написания аудиторной контрольной работы

1 Состав языка C++. Алфавит языка. Ключевые слова. Знаки операций. Константы. Комментарии. Инструкции. Ключевые слова. Идентификаторы. Разделяющие знаки.

2 Переменные. Объявление и определение переменных. Область видимости переменных.

3 Типы данных. Целый тип. Символьный тип. Расширенный символьный тип. Логический тип.

4 Типы с плавающей точкой. Тип void. Представление целочисленного типа и типа с плавающей точкой в памяти компьютера. Структура программы.

5 Ввод и вывод данных в программе. Переменные. Операции. Унарные, бинарные и тернарные. Унарные операции инкремента и декремента, операции отрицания.

6 Бинарные операции: аддитивные (+ -), мультипликативные (* / %), сдвигов (<< >>) , поразрядные (& | ^), операции отношений (> < >= <= == !=;), логические (&& ||), присваивания (= *= /= %/ += -= <<= >>= &= |= ^=) , операция «запятая», скобки в качестве операций. Тернарная операция ?: Инкрементация ++ и декрементация --. Префиксная и постфиксная формы инкрементации ++ и декрементации --. Приоритеты операций.

7 Выражения. Старшинство и порядок вычисления. Математические функции. Схема иерархии типов операндов. Явные преобразования типов.

8 Стандартные потоки ввода и вывода. Вывод с использованием cout. Ввод с использованием cin.

9 Парадигмы программирования. Процедурное программирование. ООП. Структурное программирование.

10 Простейшая программа на C++. Пространство имен.

11 Базовые конструкции структурного программирования. Операция присваивания. Множественное присваивание. Следование, ветвление, цикл. Операторы if, switch, for, while, do while, их синтаксис и особенности использования. Операторы goto, continue, break, return.

12 Строки в языке программирования C++. Строка. Описание строк. Ввод-вывод строк в C и C++.

13 Операции над строками в языке C++. Копирование, добавление, сравнения, поиска, длина строки, разбиение строки на лексемы.

14 Массивы. Одномерные и многомерные массивы. Способы инициализации. Ввод и вывод массивов.

15 Структуры. Определение структур. Выделение памяти под структуры. Инициализация структур.

16 Основные алгоритмы обработки данных, организованных в массив: поиск максимального и минимального элемента, определение среднего арифме-

тического членов массива.

17 Основные алгоритмы обработки данных, организованных в массив: суммирование строк (столбцов) двумерного массива, перестановка элементов в массиве, вставка строки (столбца) в массив.

18 Основные алгоритмы обработки данных, организованных в массив: преобразование двумерного массива в одномерный.

19 Основные алгоритмы обработки данных, организованных в массив: сумма двух массивов.

20 Алгоритмы поиска в линейных массивах. Линейный поиск.

21 Алгоритмы поиска в линейных массивах. Двоичный поиск.

22 Некоторые методы сортировки массивов. Убывающий, возрастающий, неубывающий, невозрастающий массивы. Сортировка выбором.

23 Сортировка обменом.

24 Сортировка вставками.

25 Указатели в языке C++. Понятие указателя. Объявление указателя. Операции над указателями. Арифметические операции над указателями.

26 Взаимодействие между указателями и массивами. Указатели и многомерные массивы. Массивы указателей.

27 Динамические массивы. Операторы new, delete. Организация динамических массивов.

28 Массивы и структуры в качестве элементов структур. Массивы структур.

29 Функции в языке программирования C++. Объявление и определение функций. Глобальные переменные. Возвращаемое значение. Параметры функции. Передача параметров по ссылке и по значению.

30 Передача массивов в качестве параметров. Строки как параметры функций. Структуры и функции.

31 Параметры функции по умолчанию. Функции с переменным числом параметров. Рекурсивные функции.

32 Перегрузка функций. Шаблоны функций. Функция main().

Список литературы

1 **Батан, С. Н.** Сборник задач по программированию: учебно-методические материалы / С. Н. Батан, Н. В. Кожуренко . – Могилев: МГУ имени А. А. Кулешова, 2015. – 83 с.

2 **Гавриков, М. М.** Теоретические основы разработки и реализации языков программирования: учебное пособие / М. М. Гавриков, А. Н. Иванченко, Д. В. Гринченков; под ред. А. Н. Иванченко . – Москва: Кнорус, 2016. – 177 с.

3 **Гагарина, Л. Г.** Введение в теорию алгоритмических языков и компиляторов: учебное пособие / Л. Г. Гагарина, Е. В. Кокорева . – Москва: Форум, 2018. – 175 с.

4 **Иванова, Г. С.** Технология программирования: учебник / Г. С. Иванова. – 3-е изд., стер. – Москва: Кнорус, 2018. – 333 с.

5 **Мороз, Л. А.** Технологии программирования и методы алгоритмизации: контрольные задания / Л. А. Мороз. – Могилев: МГУ им. А. А. Кулешова,

2018. – 66 с.

6 **Орлов, С. А.** Теория и практика языков программирования: учебник по направлению «Информатика и вычислительная техника» / С. А. Орлов. – Санкт-Петербург: ПИТЕР, 2013. – 688 с.

7 **Хорев, П. Б.** Объектно-ориентированное программирование с примерами на C# : учебное пособие / П. Б. Хорев . – Москва: Форум, 2018. – 197 с.

8 **Батан, С. Н.** Контрольные задания по курсу «Методы программирования и информатика» / С. Н. Батан. – Могилев: МГУ им. А. А. Кулешова, 2008. – 40 с.

9 **Давыдов, В. Г.** Программирование и основы алгоритмизации: учебное пособие / В. Г. Давыдов. – 2-е изд., стер. – Москва: Высшая школа, 2005. – 447 с.

10 **Давыдов, В. Г.** Технологии программирования C++: учебное пособие / В. Г. Давыдов. – Санкт-Петербург: BHV-Санкт-Петербург, 2005. – 672 с.

11 **Ишкова, Э. А.** C#. Начало программирования / Э. А. Ишкова. – Москва: БИНОМ, 2011. – 333 с.

12 **Канцедал, С. А.** Алгоритмизация и программирование: учебное пособие / С. А. Канцедал. – Москва: ФОРУМ; ИНФРА-М, 2019. – 352 с.

13 **Коплиен, Д.** Программирование на C++: пер. с англ. / Д. Коплиен . – Санкт-Петербург: ПИТЕР, 2005. – 479 с.

14 **Павловская, Т. А.** C/C++. Структурное программирование: практикум / Т. А. Павловская, Ю. А. Щупак. – Санкт-Петербург: ПИТЕР, 2005. – 239 с.

15 **Павловская, Т. А.** C#. Программирование на языке высокого уровня / Т. А. Павловская. – Санкт-Петербург: ПИТЕР, 2009. – 432 с.

16 **Савич, У.** Программирование на C++ / У. Савич. – 4-е изд. – Санкт-Петербург: BHV-Санкт-Петербург, 2004. – 781 с.

17 **Страуструп, Б.** Язык программирования C++ / Б. Страуструп. – спец. изд . – Москва: БИНОМ, 2008. – 1104 с.

18 **Окулов, С. М.** Программирование в алгоритмах / С. М. Окулов. – 3-е изд. – Москва: БИНОМ, 2007. – 383 с.

19 **Франка, П.** C++: учебный курс: пер. с англ. / П. Франка. – Санкт-Петербург: ПИТЕР, 2001. – 528 с.

20 **Шилдт, Г.** C++: руководство для начинающих: пер. с англ. / Г. Шилдт. – 2-е изд. – Москва; Санкт-Петербург; Киев: Вильямс, 2005. – 672 с.

21 Язык C++: учебное пособие / И. Ф. Астахова [и др.] – Минск: Новое знание, 2003. – 203 с.