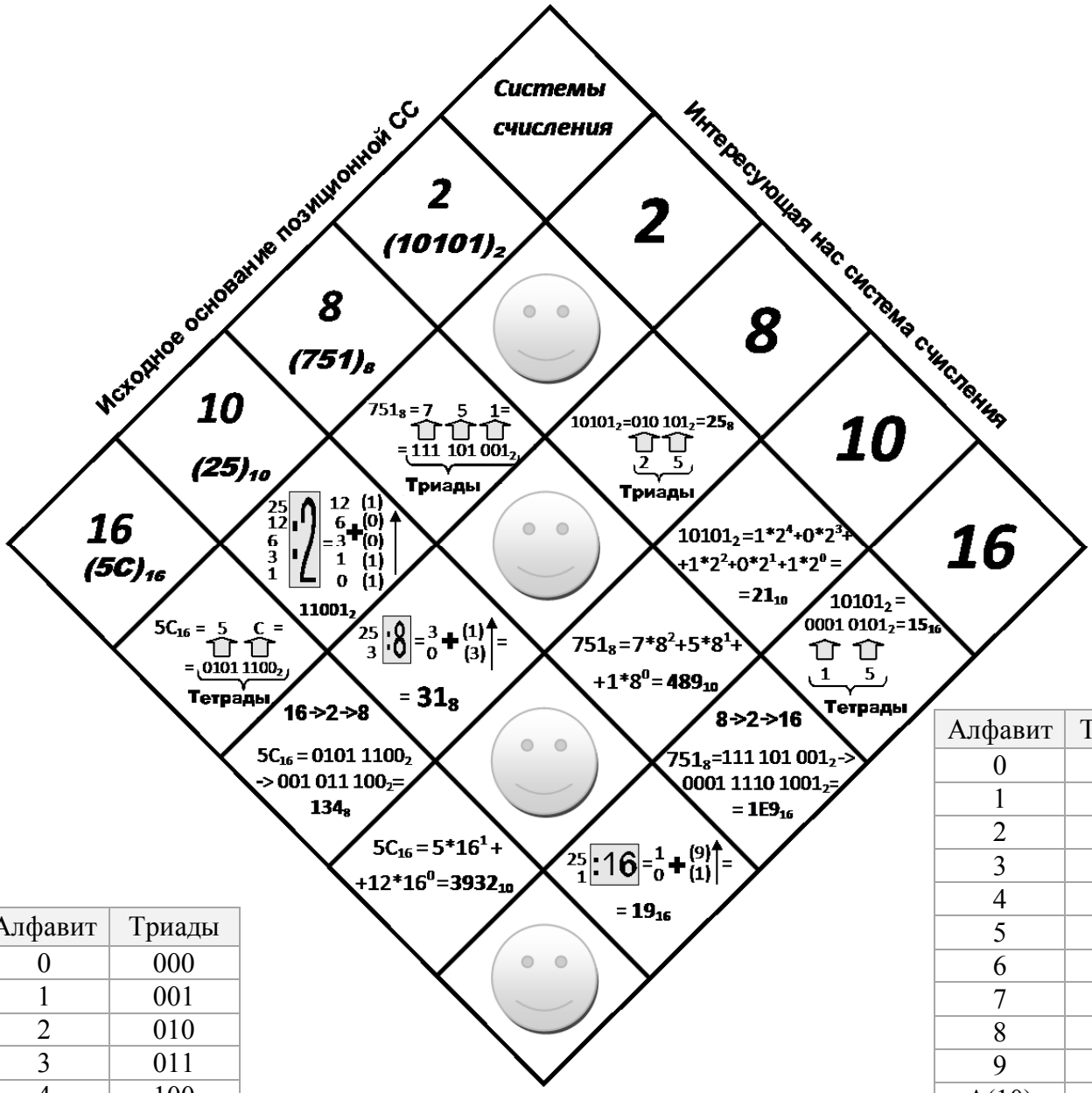
Приложения

1. Правила перевода целых чисел из одной системы счисления в другую



Алфавит	Триады
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

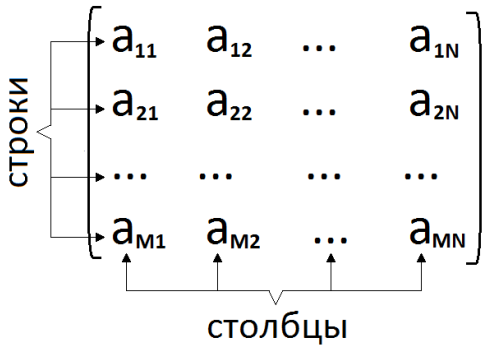
Алфавит	Тетрады
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A(10)	1010
B(11)	1011
C(12)	1100
D(13)	1101
E(14)	1110
F(15)	1111

Пояснение:

В первой строке слева написаны исходные числа для перевода ($(5C)_{16}$, $(25)_{10}$,$(751)_8$,$(10101)_2$). Справа в первой строке написана система счисления, в которую необходимо перевести (2,8,10,16). В ячейках на пересечении диагональных строк, в виде примеров, расположены правила перевода. При совпадении оснований систем счисления, на пересечении изображен **знак**  , показывающий, что в этой ячейке правило отсутствует.

2. Матрица

Матрица - это прямоугольная таблица из чисел, содержащая **M** строк и **N** столбцов. Числа **M** и **N** называются ***порядками*** матрицы. Число элементов содержащихся в матрице называется её ***размером***.



Матрица - это группа пронумерованных ячеек памяти собранная под общим именем (для нашего случая **a**). Элемент матрицы – это ячейка памяти, предназначенная для хранения некоторого значения (целочисленного, вещественного, символьного типа в зависимости от потребностей задачи).

Для обращения к ячейки памяти (элементу) матрицы в алгоритме указывается имя матрицы (для нашего случая **a**), номера строки и столбца, в которых в матрице она располагается.

ИМЯ_МАТРИЦЫ [номер_строки][номер_столбца]

M - количество строк,
N - количество столбцов

К примеру:

```
a[2][3] = 7;   // в ячейку памяти располагающуюся во 2-й строке и 3-м столбце матрицы a
               // записываем 7
```

Затем через некоторое время, тем значением, которое мы записали в элемент матрицы, можем воспользоваться (оно будет сохранено и равно последней нами записанной величины) к примеру, его запишем в переменную **x**:

x = a[2][3].

После этого переменная **x** будет равна **7**, а значение в **a[2][3]** останется без изменения и будет продолжать равняться **7**.

Матрицы, как и вектора (вектор – это матрица, состоящая из одной строки) удобны для групповой обработки данных. Когда допустим, мы хотим выполнить одну и ту же операцию сразу над несколькими значениями. Используя цикл (реализующий в алгоритмах повторение) – эту операцию мы можем совершить над несколькими ячейками матрицы, содержащие необходимые нам значения. При этом доступ к ячейке матрицы целесообразно выполнять косвенно через некоторые переменные (называемые ***индексами***) хранящие номера строки или столбца элемента, к которому происходит обращение.

К примеру:

```
i = 2;
j = 3;
a[i][j] = 8;
```

В примере в **a[2][3]** запишется **8**, так как на момент записи **i=2**, **j=3**. Переменные **i** и **j** в этом примере являются ***индексами*** т.к. используются для хранения номеров строки и столбца элемента матрицы.

Переменные, используемые для обращения к элементу матрицы в процессе выполнения алгоритма возможно изменять:

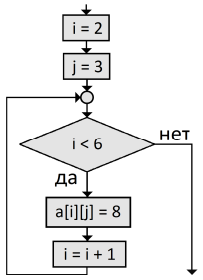
К примеру:

```
i = i +1;
a[i][j] = 8;
i = i + 1;
a[i][j] = 8;
i = i + 1;
a[i][j] = 8;
```

При выполнении получится
i = i + 1 = 2 + 1 = 3
a[**3**][3] = 8;
i = i + 1 = 3 + 1 = 4
a[**4**][3] = 8;
i = i + 1 = 4 + 1 = 5
a[**5**][3] = 8;

С помощью одной и той же записи **a[i][j] = 8;** используя косвенную адресацию изменяя **i** и **j** в любое количество ячеек матрицы возможно записать значение **8** не изменяя при этом само выражение записи.

Индексные переменные удобно изменять в цикле. В цикле значения этих переменных изменяются от начального до конечного значения с некоторым шагом – что упрощает доступ к нужных нам элементам матрицы.

фрагмент блок-схемы алгоритма	фрагмент программы	Пояснения
	**i = 2;** **j = 3;** **while (i < 6){** **a[i][j] = 8;** **i = i + 1;** **}**	Результатом этих алгоритмов будет: **a[2][3]=8;** **a[3][3]=8;** **a[4][3]=8;** **a[5][3]=8;**

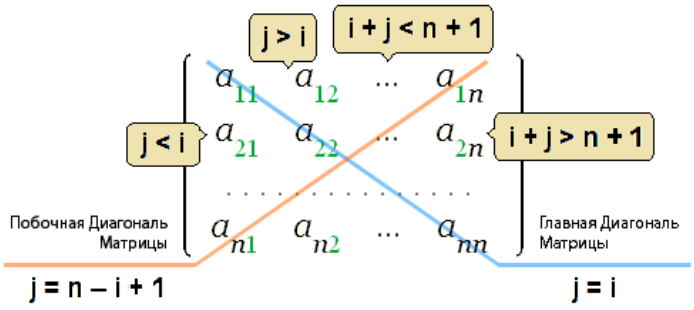
Матрица, у которой число строк равно числу столбцов называется **квадратичной**.

Главной диагональю квадратичной матрицы называется диагональ $a_{11}\ a_{22}\ \dots\ a_{nn}$ идущая из левого верхнего угла в правый нижний угол.

у элементов, лежащих на главной диагонали, номер строки и номер столбца совпадают, т.е. **j = i**.

Побочной диагональю квадратичной матрицы называется диагональ $a_{n1}\ a_{(n-1)2}\ \dots\ a_{1n}$ идущая из левого нижнего угла в правый верхний угол.

у элементов, лежащих на побочной диагонали номер строки и номер столбца связаны следующим соотношением, если i – номер строки, то номер столбца **j = n – i + 1.**



Две матрицы ***равны***, если эти матрицы имеют одинаковые размеры, и все их соответствующие элементы совпадают.

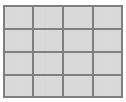
Матрицы возможно транспонировать, складывать, умножать на число, перемножать.

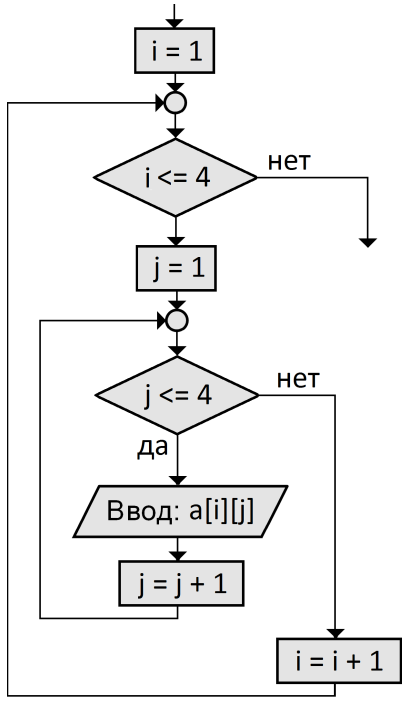
При транспонировании матрица ложится на бок, меняются местами её строки со столбцами.

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \qquad A^{T}=\begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{bmatrix}$$

Чтобы выполнить проход по всем элементам матрицы в алгоритме используют два вложенных цикла: первый (наружный цикл) изменяет значение номера строки **i**, второй (внутренний цикл) изменяет значение номера столбца - **j**. При этом индекс строки **i** меняется медленнее индекса столбца **j** благодаря чему *происходит постепенный перебор* всех элементов каждой строки, т.е. *столбцов **i**-й строки.*

Пример ввода элементов матрицы **a**, содержащей 4 строки по 4 столбца в каждой. **a**



фрагмент блок-схемы алгоритма	фрагмент программы	пояснения
	**i = 1;** **while (i <= 4) {** **j = 1;** **while (j <= 4) {** **scanf("%d", &a[i][j]);** **j = j + 1;** **}** **i = i + 1;** **}**	Результатом этих алгоритмов будет ввод пользователем с клавиатуры значений элементов матрицы: **a[1][1], a[1][2], a[1][3], a[1][4], a[2][1], a[2][2], a[2][3], a[2][4], a[3][1], a[3][2], a[3][3], a[3][4], a[4][1], a[4][2], a[4][3], a[4][4]** После этого элементы матрицы будут хранить значения, введённые пользователем.

3. Таблица кодирования символов ASCII

Основная таблица ASCII

	00	10	20	30	40	50	60	70
0		►		0	@	P	`	p
1	☺	◄	!	1	A	Q	a	q
2	☻	↕	"	2	B	R	b	r
3	♥	‼	#	3	C	S	c	s
4	♦	¶	$	4	D	T	d	t
5	♣	§	%	5	E	U	e	u
6	♠	▬	&	6	F	V	f	v
7	•	↨	'	7	G	W	g	w
8	◘	↑	(	8	H	X	h	x
9	○	↓	)	9	I	Y	i	y
A	◙	→	*	:	J	Z	j	z
B	♂	←	+	;	K	[	k	{
C	♀	∟	,	<	L	\	l	|
D	♪	↔	-	=	M	]	m	}
E	♫	▲	.	>	N	^	n	~
F	☼	▼	/	?	O	_	o	⌂

Расширенная таблица ASCII

	80	90	A0	B0	C0	D0	E0	F0
0	А	Р	а	░	└	╨	р	≡
1	Б	С	б	▙	┴	╤	с	±
2	В	Т	в	▛	┬	╥	т	≥
3	Г	У	г	|	├	╙	у	≤
4	Д	Ф	д	┤	─	╘	ф	⌠
5	Е	Х	е	╡	┼	╒	х	⌡
6	Ж	Ц	ж	╢	╞	╓	ц	÷
7	З	Ч	з	╖	╟	╫	ч	≈
8	И	Ш	и	╕	╚	╪	ш	°
9	Й	Щ	й	╣	╔	┘	щ	∙
A	К	Ъ	к	║	╩	┌	ъ	·
B	Л	Ы	л	╗	╦	█	ы	√
C	М	Ь	м	╝	╠	▄	ь	№
D	Н	Э	н	╜	═	▌	э	²
E	О	Ю	о	╛	╬	▐	ю	■
F	П	Я	п	┐	╧	▀	я	

Примечание: каждый символ в этой таблице имеет свой уникальный код в шестнадцатеричной системе счисления. Код символа определяется его местоположением в этой таблице и равен:

Код символа = № столбца + № строки.

К примеру: ‘J’ = “код символа J по табл. ASCII” = 0x40 + 0xA = 0x4A;
‘S’ = “код символа S по табл. ASCII” = 0x50 + 0x3 = 0x53.

4. Базовые типы данных

тип данных	размер в байтах	диапазон значений	название
void	0	0	пусто
char	1	-128... 127	целый длиной в 1 байт
unsigned char	1	0... 255	беззнаковый целый длиной в 1 байт
int	4	-2147483648... 2147483647	целый
unsigned int	4	0 ... 4294967295	беззнаковый целый
short	2	-32768 ... 32767	короткий целый
unsigned short	2	0 ... 65535	беззнаковый короткий целый
long	4	-2147483648... 2147483647	длинный целый
unsigned long	4	0 ... 4294967295	беззнаковый длинный целый
float	4	$3.4\cdot10^{-38}$ ... $3.4\cdot10^{+38}$	вещественный одинарной точности
double	8	$1.7\cdot10^{-308}$ ... $1.7\cdot10^{+308}$	вещественный двойной точности
long double	10	$3.4\cdot10^{-4932}$... $1.1\cdot10^{+4932}$	вещественное максимальной точности
enum	2	-32768... 32767	перечисление (целого типа)

5. Приоритеты операций

ранг	операция	вид операции	порядок выполнения в выражении
1	**( ) [ ] -> .**	выражение	→
2	**! ~ ++ -- + - * & (тип) sizeof**	унарный	←
3	*** / %**	мультипликативный	→
4	**+ -**	аддитивный	→
5	**<< >>**	сдвиг	→
6	**< <= > >=**	отношение	→
7	**== !=**	отношение	→
8	**&**	поразрядное «И»	→
9	^	поразрядное исключающее «ИЛИ»	→
10	|	поразрядное «ИЛИ»	→
11	**&&**	логическое «И»	→
12	||	логическое «ИЛИ»	→
13	**?:**	условная	←
14	**= *= /= %= += -= &= |= ^= <<= >>=**	простое и составное присваивание	←
15	**, (операция запятая)**	последовательное вычисление	→

6. Справочник библиотечных функций

(заголовочный файл <math.h>)

acos

double acos(double x);

Возвращает арккосинус аргумента. Функция возвращает значение арккосинуса аргумента **x** в радианах.

asin

double asin(double x);

Возвращает арксинус аргумента. Функция возвращает значение арксинуса аргумента **x** в радианах.

atan

double atan(double x);

Возвращает арктангенс аргумента. Функция возвращает значение арктангенса аргумента **x** в радианах.

atan2

double atan2(double y, double x);

Возвращает арктангенс отношения аргументов. Функция возвращает значение арктангенса отношения параметров **y**/**x** в радианах.

ceil

double ceil(double x);

Округляет вверх. Функция округляет вещественное значение **x** до ближайшего большего целого и возвращает его как вещественное.

cos

double cos(double x);

Вычисляет косинус. Функция возвращает значение косинуса угла, равного **x** радиан.

cosh	**double cosh(double x);** Вычисляет гиперболический косинус. Функция возвращает значение гиперболического косинуса угла, равного **x** радиан. Если значение функции окажется вне представимого диапазона, то функция возвращает значение HUGEVAL, а глобальная переменная **errno** получает значение ERANGE.
fabs	**double fabs(double x);** Возвращает модуль числа. Функция возвращает абсолютное значение числа **num**.
floor	**double floor(double x);** Округляет вниз. Функция округляет вещественное значение **x** до ближайшего меньшего целого и возвращает его как вещественное.
fmod	**double fmod(double x, double y);** Возвращает остаток от деления **x** на **y**. Функция возвращает остаток от деления **x** на **y**. Аналогична операции *%,* но работает с вещественными числами.
frexp	**double frexp(double x, int *expptr);** Выделяет из числа мантиссу и экспоненциальную часть. Функция выделяет мантиссу и показатель степени числа **x**. Возвращает значение мантиссы и копирует экспоненциальную часть по адресу **expptr**.
ldexp	**double ldexp(double x, int exp);** Преобразует мантиссу и показатель степени в число. Функция получает мантиссу **x** и показатель степени **exp** и возвращает число, равное произведению мантиссы на 2 в степени показатель степени. Противоположна функции **frexp**.
log	**double log(double x);** Вычисляет натуральный логарифм. Функция возвращает значение натурального логарифма **x**.
log10	**double log10(double x);** Вычисляет логарифм по основанию 10. Функция возвращает значение логарифма **x** по основанию 10.
modf	**double modf(double x, double *intptr);** Разбивает число на целую и дробную части. Функция разбивает **x** на целую и дробную части, причем дробную часть числа возвращает, а целую часть числа помещает по адресу, определяемому указателем **intptr**.
pow	**double pow(double x, double y);** Возводит число в степень. Функция вычисляет значение числа **x** в степени **y**.
sin	**double sin(double x);** Вычисляет синус. Функция возвращает значение синуса угла, равного **x** радиан.
sinh	**double sinh(double x);** Вычисляет гиперболический синус. Функция возвращает значение гиперболического синуса угла, равного **x** радиан. Если значение функции окажется вне представимого диапазона, то функция возвращает значение HUGEVAL, а глобальная переменная **errno** получает значение ERANGE.

sqrt	**double sqrt(double x);** Вычисляет квадратный корень. Функция возвращает квадратный корень из числа **x**.

tan	**double tan(double x);** Возвращает тангенс аргумента. Функция возвращает значение тангенса аргумента **x**.

tanh	**double tanh(double x);** Возвращает гиперболический тангенс аргумента. Функция возвращает значение гиперболического тангенса аргумента **x**.

(заголовочный файл <stdlib.h>)

abs	**int abs(int num);** Функция возвращает абсолютное значение числа **num**.

labs	**long int labs(long int num);** Функция возвращает абсолютное значение числа **num**.

rand	**int rand(void);** Функция **rand** возвращает псевдослучайное число в диапазоне от 0 до RAND_MAX (RAND_MAX > 32767).

srand	**void srand(unsigned int seed);** Функция **srand** использует параметр **seed** как инициализирующее значение (затравку) для новой последовательности псевдослучайных чисел. Вначале параметр **seed** равен **1**.

7. Пример трассировки алгоритма

Необходимо выполнить трассировку алгоритма определения наибольшего общего делителя двух натуральных чисел (алгоритма Евклида).

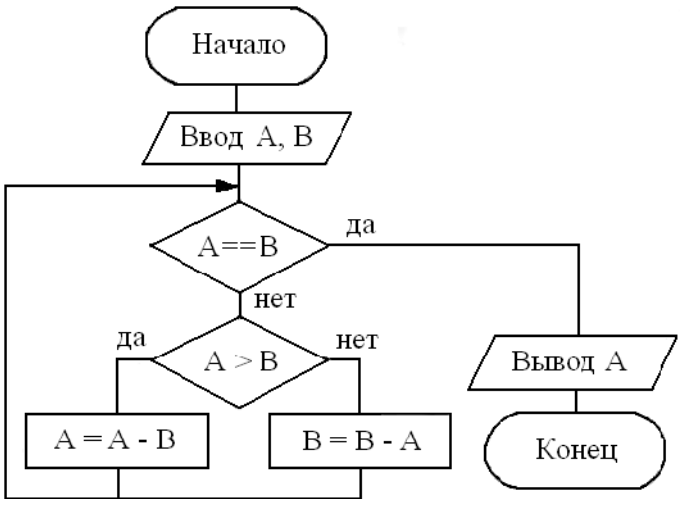


Рис. Блок - схема алгоритма Евклида.

Таблица трассировки алгоритма Евклида

Команда	Значения переменных и условий			
	A	**B**	**A==B**	**A>B**
Ввод A, B	32	24		
Условие: A==B			нет	
Условие: A>B				да
A = A - B	8			
Условие: A==B			нет	
Условие: A>B				нет
B = B - A		16		
Условие: A==B			нет	
Условие: A>B				нет
B = B - A		8		
Условие: A==B			да	
Вывод A	8			

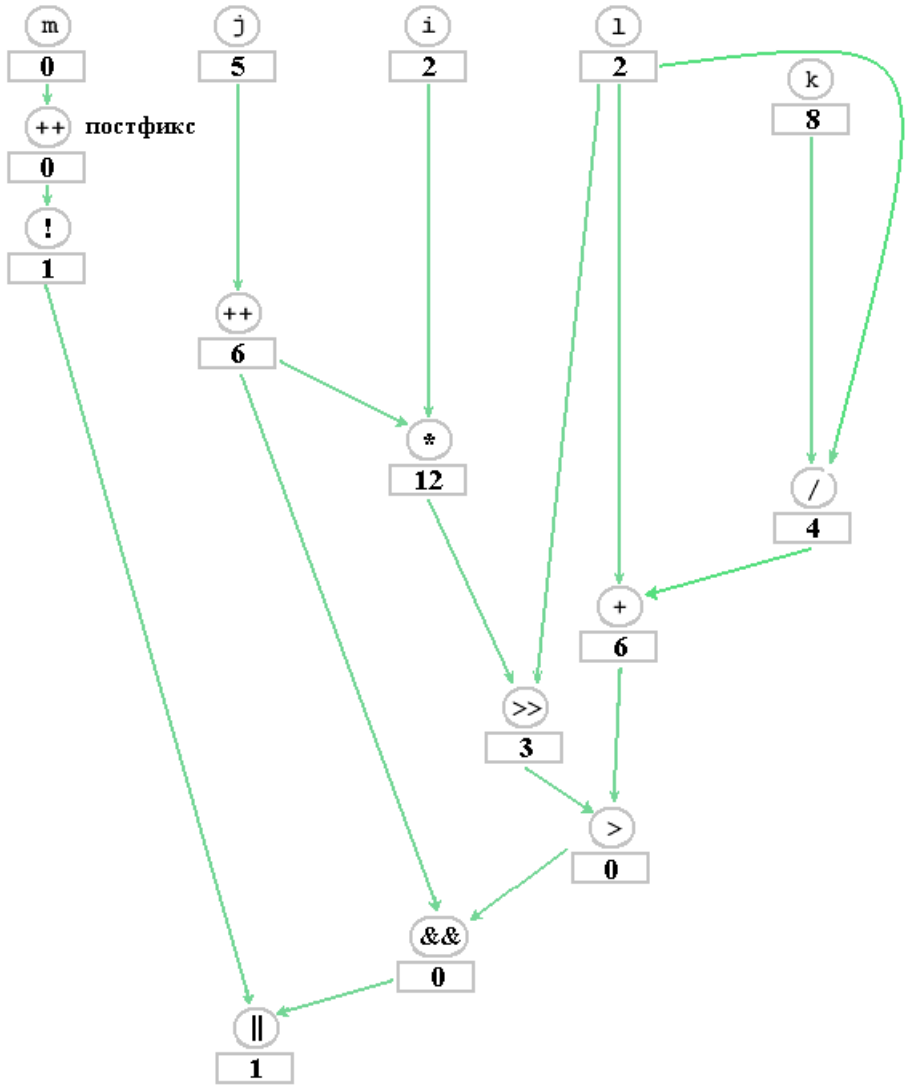
8. Пример разбора выражения i*j>>l>k/l+l&&++j||!m++

Исходные значения переменных: i = 2 j = 5 k = 8 l = 2 m = 0

	Номера приоритетов операций																
		3		5		6		3		4		11	2		12	2		2
Выражение 4:		i	*	j	>>	l	>	k	/	l	+	l	&&	++	j	||	!	m	++
Порядковые номера вычислений			4		7		8		5		6		9	3		10	2		1

Замечание: порядковые номера вычисления операций пишутся строго под знаками операций.

Дерево разбора выражения:



Замечание: в основании дерева разбора разместите переменные (в примере это **i**, **j**, **k**, **l**, **m**) в порядке, чтобы количество пересечений было минимальным.

9. Правила оформления программного кода

Программный код предназначен микропроцессору компьютера для исполнения, которое происходит после его компиляции, компоновки и трансляции в машинный код. Но при этом этот код, набор инструкций (выделения памяти, манипулирования данных хранящихся в этой памяти, взаимодействия с периферийными устройствами) пишет человек.

Одну и ту же последовательность инструкций возможно написать по разному: “правильно” когда код представлен в читабельном виде и с первого взгляда понятен программисту, и “не правильно” когда он плохо читается и необходимо поломать немало времени программисту, чтобы понять что он делает. Второй стиль “неумелого” программирования кроме усложнения самого процесса программирования является ещё и благодатной почвой для разнообразных ошибок.

В этом приложение приведены правила оформления программного кода, которым необходимо придерживаться при выполнении лабораторных работ и создания собственных программ.

9.1 Имена. При программировании объектам программы (переменным, функциям, меткам, пользовательским типам) программист даёт имена (уникальные в этой программе слова, состоящие из латинских букв, арабских цифр и знака подчёркивания и начинающихся либо с букв, либо со знака подчёркивания), чтобы ими можно было бы в этой программе пользоваться (к ним обращаться). Как и в реальной жизни, в программе к её объекту обращение идёт по имени.

Имя объекту должно быть выбрано программистом, осмысленно и понятно, в соответствии тому свойству или процессу, который он обозначает. Оно должно быть построено на основе английских слов. Не должно состоять целиком из больших букв. Имена констант следует задавать только большими буквами. При определении имён переменных используйте Венгерскую нотацию.

9.2 Начало и конец управляющей конструкции. В управляющих конструкциях начало (заголовок оператора и “**{**“) и конец “**}**”) должны располагаться от левого края в тексте программы на одном и том же расстоянии (равному количеству пробелов). Для первого уровня их вообще не должно быть.

К примеру:

```
if (condition>0){
        // программный код 1
}
else {
        // программный код 2
}
```

```
for (i=0; i<10; ++i){
        // программный код 3
}
```

```
int Func (int nParam1, long lParam2)
{
    // программный код функции Func
}
```

```
while (i!=0) {
    // программный код 4
}
```

```
switch (i){
    // программный код 5
}
```

9.3 Вложенность управляющих конструкций. На одной строке программного кода должно находится не более одного оператора C/C++. При программировании достаточно часто одна конструкция может находиться внутри другой конструкции, то есть вкладываться. К примеру, необходимо повторить некоторое действие, которое содержит условие. В этом случае условный блок if будет вложен (будет находиться внутри) блока цикла (for, while либо do-while). В программе блок

цикла будет многократно повторять блок if, поэтому блок if будет находиться внутри блока и будет подчиненном. Начало и конец подчиненного блока относительно управляющего должны быть сдвинуты вправо на несколько пробелов.

Пример:

```
// управляющий блок
for (i=0; i<10; ++i){
    // подчиненный блок, находится внутри управляющего блока
    if (condition>0){
        // программный код 1
    }
    else {
        // программный код 2
    }
}
```

В один управляющий блок могут быть вложено несколько подчиненных блоков. Если у них один и тот же уровень вложенности, то оно должны быть сдвинуты вправо на одно и то же количество пробелов.

Пример:

```
// управляющий блок
for (i=0; i<10; ++i){

    // подчиненный блок 1, находится внутри управляющего блока
    if (i>5){
        // программный код 1
    }

    // подчиненный блок 2, находится внутри управляющего блока
    while (i!=0) {
        // программный код 2
    }
}
```

Если уровень вложенности у подчинённых блоков разный, то они должны быть сдвинуты вправо на разное количество пробелов.

Пример:

```
// управляющий блок
for (i=0; i<10; ++i){
    // подчиненный блок 1, находится внутри управляющего блока
    if (i>5){
        // подчиненный блок 2, находится внутри подчиненного блока 1
        while (i!=0) {
            // программный код 2
        }
    }
}
```

9.4 Комментарии. Комментарии - это примечания, помогающие понять смысл программы. Они предназначены для программистов и поэтому игнорируются компилятором. В программе на языке C/C++ могут существовать двухсторонние комментарии: /* комментарий1 */ , либо односторонние:

// комментарий 2

При комментировании программы необходимо использовать только односторонние комментарии.

Строка комментария должна иметь такой же отступ, как и операция, которую она комментирует. In-line комментарии должны размещаться справа от комментируемого кода. Оператор комментария “//” должен отделятся от текста комментария по крайней мере одним пробелом.

Пример:

```
POINT ptCurrentMenuArea;   // in-line comment

// check if designated point is in the menu area
if (!IsMenuArea(ptCurrentMenuArea)){
        // out of the menu area
        return NO_MENU_ID;
}
```

10. Схема иерархии типов

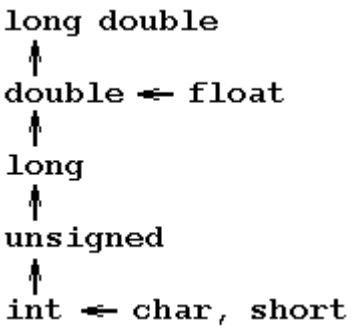
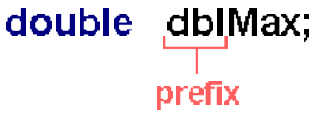


Схема иерархии типов показывает упорядоченность арифметических типов. Тип результата каждой арифметической операции есть тип того ее операнда, который имеет в соответствии с приведенной схемой более высокий тип. Аналогично, если сравниваются два значения, то значение, имеющее низкий тип, преобразовывается к высшему согласно приведенной схеме. Упорядоченность типов на схеме показывают вертикальные стрелки: самый высший тип - **`long double`**, самый низший - **`int`**. Горизонтальные стрелки задают порядок автоматического приведения типов в выражении. А именно: операнды типа **`float`** всегда преобразуются перед использованием в значения типа **`double`**, а операнды типа **`char`** или **`short`** всегда преобразуются в значения типа **`int`**.

11. Венгерская нотация

Венгерская нотация — соглашение программистов об именовании идентификаторов в коде программ.



Тип →	Prefix →	Пример
char	ch, c	*ch*Grade
char[]	sz, str	*sz*Buffer
short	s	*s*Index
int	n, i	*n*Length
unsigned	u	*u*Size
long	l	*l*Sum
unsigned long	ul	*ul*FreeSpace
float	flt	*flt*Result
double	dbl	*dbl*Result
long double	ldbl	*ldbl*Result

12. Библиотека дополнительных функций, для выполнения лабораторных работ по курсу "Информатика" (student.h)

```
// Этот файл должен быть помещен в папку          C:/Program Files/Borland/CBuilder6/Include
// и далее перед использованием библиотеку необходимо подключить      #include <student.h>
const float pi = 3.14159265358979323846264643;   // Пи
const float e = 2.71828182845904523536028747;    // число Эйлера

// функция для вычисления факториала (n!)
long double factorial(long double n) {
  if (n<=0) return 1;
  else return n*factorial(n-1);
}
```

```
// функция для вычисления двойного факториала (n!!), с шагом 2
long double dbl_factorial(long double n) {
  if (n<=0) return 1;
  else return n*dbl_factorial(n-2);
}

// Функция перекодировки кириллицы
char* Rus(char * S) {
  static char *cp_str = NULL;
  static long len_str = 0;
  int i;

  // Определяем длину выводимой строки
  for (i=0;S[i]!=0;++i);

  // Выделяем динамическую память
  if (i+1 > len_str){
       if (cp_str != NULL) delete [] cp_str;
       len_str = i+1;
       cp_str = new char [len_str];
  }

  for (i=0;S[i]!=0;++i) {
    if (S[i]>='A' && S[i]<='п') cp_str[i]=S[i]-64;
    else {
       if (S[i]>='р' && S[i]<='я') cp_str[i]=S[i]-16;
       else {
            if (S[i]=='Ё') cp_str[i]=240;
            else {
                if (S[i]=='ё') cp_str[i]=241;
                else {
                     if (S[i]==(char)0xB9) cp_str[i]=0xFC;
                     else cp_str[i]=S[i];
                }
            }
       }
    }
  }

  cp_str[i]=0; // конец строки
  return cp_str;
}
```

13. Генератор случайных чисел

(заголовочные файлы <stdlib.h>, <time.h>)

Генератор псевдослучайных чисел предназначен для выработки друг на друга непохожих чисел. Обычно он реализовывается на основе операции деления. Для получения случайного числа в программе из библиотеки **<stdlib.h>** используют функцию **rand** (англ. *random – случайный*).

int rand(void); Функция **rand** возвращает псевдослучайное число в диапазоне от **0** до **RAND_MAX**; значение **RAND_MAX** — не меньше **32767**.

```
int X, Y;
X = rand();  // псевдослучайное число
Y = rand();  // это уже другое число!
```

Чтобы генератор случайных чисел перед генерацией последовательности случайных чисел всегда начинал работать “по разному”, его “затравливают” случайным значением через функцию **srand**, в качестве которого обычно используют значение системного времени **time(NULL)** (из библиотеки **time.h**).

void srand(unsigned int seed); Функция **srand** использует параметр **seed** как инициализирующее значение (затравка) для новой последовательности псевдослучайных чисел. Вначале параметр **seed** равен **1**.

```
srand (time(NULL));        // обновляет базу случайных чисел
```

Функция **rand** возвращает псевдослучайное число в диапазоне от **0** до **RAND_MAX** (на отрезке **[0, RAND_MAX]**). Если же мы хотим получить случайные числа на **отрезке [a, b]** то для этого необходимо в программе написать следующее:

```
int X;
X = a + rand() % (b - a + 1);   // случайные числа из диапазона [a, b]
```

Чтобы не писать громозких выражений в программе лучше всего для генерации псевдослучайных чисел в диапазоне **[a, b]** использовать следующую функцию:

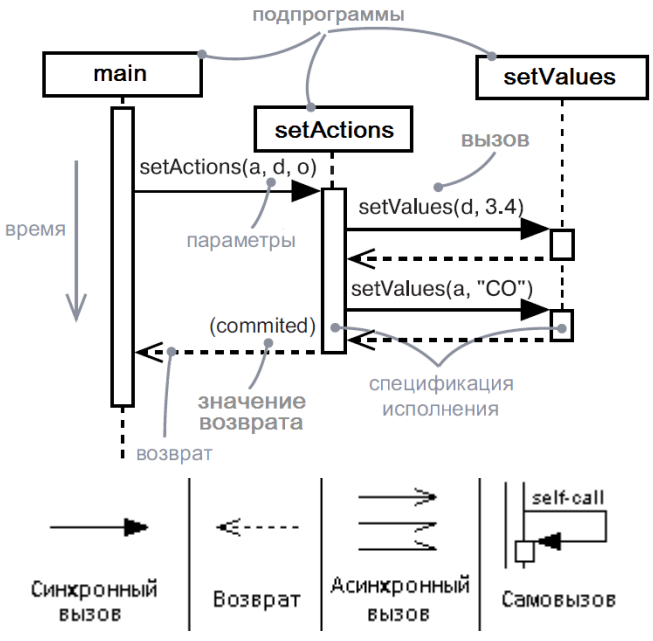
```
int irand ( int a, int b )
{
  return a + rand()% (b - a + 1);
}

int X;
X = irand(10, 99);   // X будет равно случайному числу из диапазона [10, 99]
```

14. Диаграмма UML последовательности

Диаграмма последовательности (sequence diagram) — это способ описания поведения программы "на примерах". Это запись протокола конкретного сеанса работы программы. В процедурном программировании самым существенным во время выполнения является вызов подпрограмм.

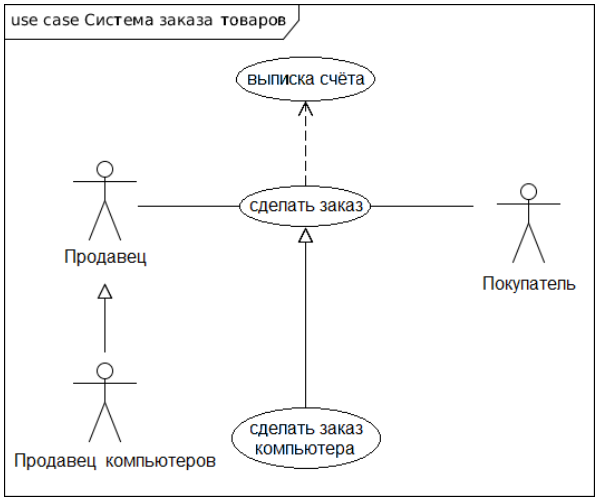
Формирование диаграммы последовательности начинается с размещения подпрограмм, участвующих во взаимодействии, в верхней части диаграммы, по горизонтальной оси. Далее по вертикальной оси в хронологическом порядке расставляются их вызовы.



Возможные виды взаимодействий:

15. Диаграмма UML вариантов использования

Диаграмма вариантов использования (use case diagram) — описывает последовательности действий (транзакций), выполняемых системой в ответ на событие, инициируемое некоторым внешним объектом (действующим лицом).

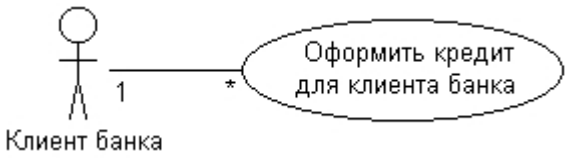


Каждый вариант использования определяет последовательность действий, которые должны быть выполнены проектируемой системой при взаимодействии ее с соответствующим действующем лицом.

Вариант использования описывает типичное взаимодействие между пользователем и системой.

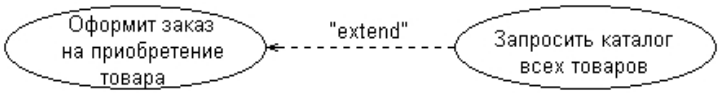
На диаграмме использования применяются два типа основных сущностей: варианты использования и действующие лица, между которыми устанавливаются следующие основные типы отношений.

Отношение ассоциации - это отношение устанавливает, какую конкретную роль играет актер при взаимодействии с экземпляром варианта использования.



Отношение ассоциации обозначается сплошной линией между актером и вариантом использования. Эта линия может иметь дополнительные условные обозначения, такие, например, как имя и кратность.

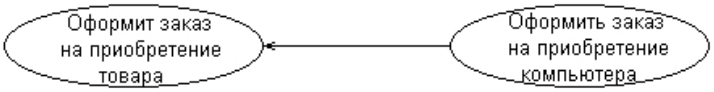
Отношение расширения - определяет взаимосвязь экземпляров отдельного варианта использования с более общим вариантом, свойства которого определяются на основе способа совместного объединения данных экземпляров. Так, если имеет место отношение



Отношение расширения между вариантами использования обозначается пунктирной линией со стрелкой (вариант отношения зависимости), направленной от того варианта использования, который является расширением для исходного варианта использования.

расширения от варианта использования **A** к варианту использования **B**, то это означает, что свойства экземпляра варианта использования **B** могут быть дополнены благодаря наличию свойств у расширенного варианта использования **A**.

Отношение обобщения служит для указания того факта, что некоторый вариант использования A может быть обобщен до варианта использования B. В этом случае вариант A будет являться специализацией варианта B. При этом B

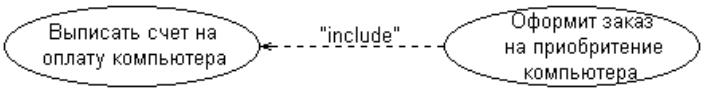


Графически данное отношение обозначается сплошной линией со стрелкой в форме не закрашенного треугольника, которая указывает на родительский вариант использования.

называется предком или родителем по отношению A, а вариант A - потомком по отношению к варианту использования B.

Отношение обобщения между вариантами использования применяется в том случае, когда необходимо отметить, что дочерние варианты использования обладают всеми атрибутами и особенностями поведения родительских вариантов.

Отношение включения между двумя вариантами использования указывает, что некоторое заданное поведение для одного варианта использования включается в качестве составного компонента в последовательность поведения другого варианта использования.

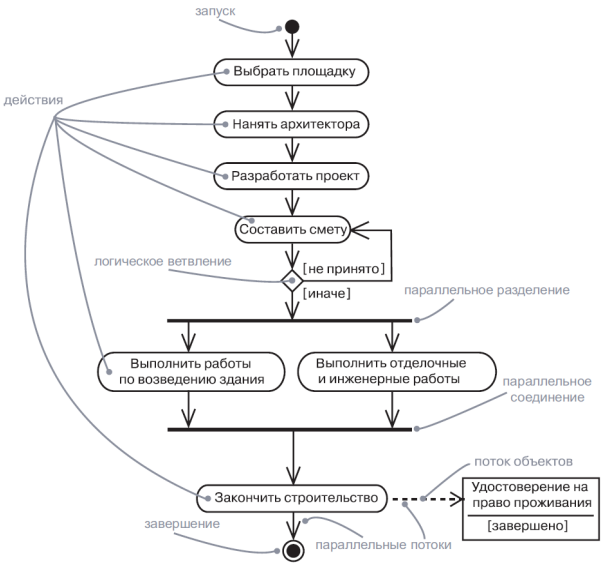


Графически данное отношение обозначается пунктирной линией со стрелкой (вариант отношения зависимости), направленной от базового варианта использования к включаемому.

Отношение включения, направленное от варианта использования **A** к варианту использования **B**, указывает, что каждый экземпляр варианта **A** включает в себя функциональные свойства, заданные для варианта **B**.

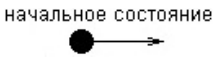
16. Диаграмма UML деятельности

Диаграмма деятельности (activity diagram) — предназначена для описания поведения, который визуально напоминает блок-схему алгоритма. Используется для моделирования процесса выполнения операций. В ней детально отображаются особенности алгоритмической и логической реализации выполняемых системой операций.

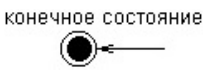


На диаграмме деятельности применяют один основной тип сущностей — действие, и один тип отношений — переходы (передачи управления). Также используются такие конструкции как развилки, слияния, соединения, ветвления. Рекомендуется в качестве имени простого действия использовать глагол с пояснительными словами.

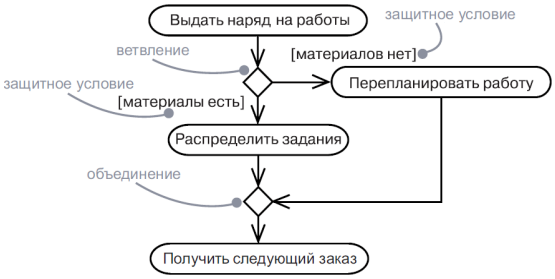
Начальное состояние служит для указания на диаграмме состояний графической области, от которой начинается процесс изменения состояний объекта.



Конечное (финальное) состояние, - это состояние, в котором объект оказывается после завершения исполнения диаграммы деятельности (работы автомата).

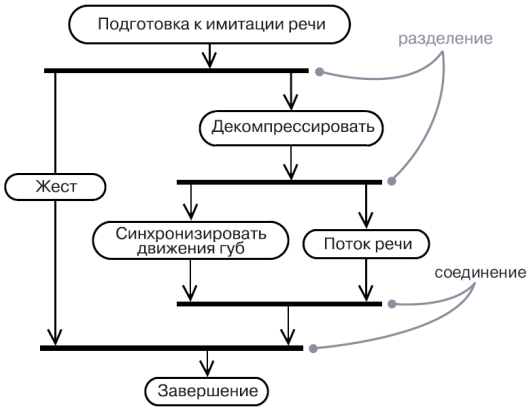


Ветвление (branching) имеет один вход и несколько выходов. На каждом выходе помещается логическое выражение условия, при выполнении которого управление передаётся по нему. Условия не должны пересекаться, чтобы не возникала неоднозначность. Для удобства в качестве логического выражения условия можно использовать ключевое слово **else**.



Линейка синхронизации (synchronization bar) – жирная горизонтальная или вертикальная линия, отображающая в UML разделение и соединение параллельно выполняющихся действий (потоков). Число потоков, исходящих из точки разделения, должно быть равно числу потоков, приходящих в соответствующую точку соединения. Параллельные потоки, могут обмениваться информацией между собой, посылая друг другу сигналы.

В качестве примера приведена система, имитирующая человеческую речь и жестикуляцию, выполняющихся параллельно.



Список литературы

1. Демидович Е.М. Основы алгоритмизации и программирования. Язык СИ : учеб. пособие / Е.М. Демидович. — СПб.: БХВ-Петербург, 2006. — 440 с.

2. Березин Б. И., Березин С. Б. Начальный курс С и С++. - М.: ДИАЛОГ-МИФИ, 2005. - 288 с.

3. Подбельский В. В., Фомин С. С. Язык СИ++: Учеб. пособие. - 2-е доп. изд. - М.: Финансы и статистика, 2008. — 600 с: ил. [Шифр книги в каталоге МАИ: 004.4(075) П-44]

4. Павловская Т. А. С/С++. Программирование на языке высокого уровня — СПб.: Питер, 2006. — 461 с: ил. [Шифр книги в каталоге МАИ: 004.4(075) П124]

5. Г. С. Иванова, Т.Н. Ничушкина, О.В. Платонова. Создание консольных приложений в среде Turbo Delphi 2006. Методические указания по выполнению лабораторных работ и практикума. Москва 2007. МГТУ им. Н. Э. Баумана.

