СИСТЕМЫ СЧИСЛЕНИЯ

1. Кодирование и единицы измерения информации. Представление числовой информации в ПЭВМ.
2. Системы счисления, применяемые в ПЭВМ.
3. Способы перевода чисел из одной позиционной системы счисления в другую.
4. Формы представления чисел в ПЭВМ.
5. Способы кодирования двоичных чисел в ПЭВМ.

1. КОДИРОВАНИЕ И ЕДИНИЦЫ ИНФОРМАЦИИ. ПРЕДСТАВЛЕНИЕ ЧИСЛОВОЙ ИНФОРМАЦИИ В ПЭВМ.

Окружающую нас информацию можно разделить на:

- цифровую (различного вида числа);
- текстовую (буквы, знаки препинания, числа, математические знаки и специальные общеупотребительные знаки);
- звуковую;
- графическую (схемы, чертежи, фотографии, рисунки);
- видеоинформация.

Для передачи, хранения и обработки информации, а так же определения ее количества все виды информации нужно преобразовать в единый стандартный вид. Такое преобразование информации называется *кодированием*. ***Кодирование — это процесс представления символов одного алфавита символами другого.***

КОДИРОВАНИЕ ДАННЫХ ДВОИЧНЫМ КОДОМ

Естественные человеческие языки – это не что иное, как системы кодирования понятий для выражения мыслей посредством речи. К языкам близко примыкают ***азбуки*** (системы кодирования компонентов языка с помощью графических символов). Системы кодирования успешно применяются в отдельных отраслях техники, науки и культуры. В качестве примеров можно привести систему записи математических выражений, телеграфную азбуку, морскую флажковую азбуку, систему Брайля для слепых и т.п.

Возможность представления информации двоичными цифрами впервые была предложена Вильгельмом Лейбницем. К этой идее его привело исследование концепции единства и борьбы противоположностей (добра и зла, «черного» и «белого», мужского и женского). Попытка применить к изучению мироздания методы «чистой» математики подтолкнули Лейбница к изучению свойств двоичного представления данных с помощью нулей и единиц.

В электрических и электронных устройствах речь идет о регистрации состояний элементов устройства: включен – выключен, открыт – закрыт, заряжен – разряжен и т.п.

Своя система кодирования существует и в вычислительной технике – она называется ***двоичным кодированием*** и основана на представлении данных последовательностью всего двух знаков: 0 и 1. Эти знаки называются двоичными цифрами, по-английски – *binary digit* или сокращенно *bit* (бит).

Одним битом могут быть выражены два понятия: 0 или 1 (да или нет, истина или ложь и т.п.). Если увеличить количество битов до двух, то уже можно выразить четыре различных понятия:

00, 01, 10, 11. Соответственно тремя битами можно закодировать восемь различных состояний: 000, 001, 010, 011, 100, 101, 110, 111. Увеличивая на единицу количество разрядов в системе двоичного кодирования, в два раза увеличивается количество значений, которое может быть выражено в данной системе, то есть общая формула имеет вид:

$N = 2^m$,

где N – количество независимых кодируемых значений;

m – разрядность двоичного кодирования, принятая в данной системе.

Наименьшая единица информации, принимающая значение 0 или 1, называется ***битом*** (от англ. binary digit — двоичная цифра). Один бит информации содержит, например, один элемент (разряд) ВЗУ ЭВМ. Число событий, которое может быть закодировано определенным количеством разрядов, показано в таблице 1.1.

Таблица 1.1. Связь между количеством разрядов и числом возможных значений при двоичном кодировании.

Кол-во разрядов (бит)	1	2	3	4	…	8	…	16	…	32
Число возможных значений	$2^1=2$	$2^2=4$	$2^3=8$	$2^4=16$	…	$2^8=256$	…	$2^{16}=65536$	…	$2^{32}=$ 4294967296

Пользуясь таблицей, можно определить необходимое число разрядов для кодирования определенного числа возможных событий. Так как значащими цифрами в *десятичной системе счисления* являются: 0,1,2,3,4,5,6,7,8,9, то для кодирования указанных цифр достаточно четырех разрядов (четырехразрядной последовательности 0 и 1), которые обеспечивают 16 различных значений.

Для удобства работы биты объединяются в группы из восьми битов, которые называются ***байтами***. Понятие о байте, как группе взаимосвязанных битов, появилась вместе с первыми образцами электронной вычислительной техники. Долгое время оно было машинно-зависимым, то есть для разных вычислительных машин длина байта была разной. Только в конце 60-х годов понятие байта стало универсальным и машинно-независимым.

Более крупная единица измерения – килобайт (Кбайт). Для вычислительной техники, работающей с двоичными числами, более удобно представление чисел в виде степени двойки, и потому на самом деле 1 Кбайт равен 2^{10} байт (1024) байт.

Более крупные единицы измерения данных образуются добавлением префиксов мега-, гига-, тера-; в более крупных единицах пока нет практической надобности.

Единицы измерения информации:

1 бит=1 двоичный разряд= 0 или 1.

1 байт=8 бит(byte), количество битов, используемое для кодирования одного символа.

1 Кбайт = 1024 байт = 2^{10} байт.

1 Мбайт = 1024 Кбайт = 2^{20} байт

1 Гбайт = 1024 Мбайт = 2^{30} байт

1 Тбайт = 1024 Гбайт = 2^{40} байт

Единицы хранения данных

При хранении данных решаются две проблемы: как сохранить данные в наиболее компактном виде и как обеспечить к ним удобный и быстрый доступ.

В качестве единицы хранения данных принят объект переменной длины, называемый файлом. ***Файл*** – это последовательность произвольного числа байтов, обладающая уникальным собственным именем. Обычно в отдельном файле хранят данные, относящиеся к одному типу. В этом случае тип данных определяет тип файла. Проще всего представить себе файл в виде безразмерного канцелярского досье, в которое можно по желанию добавлять содержимое или извлекать его оттуда.

При определении файла особое внимание уделяется имени файла, так как оно несет в себе адресные данные, без которых невозможно осуществить доступ к хранящимся в файле данным.

Кодирование текстовых данных

Кодирование текстовых данных производится путем сопоставления символов алфавита определенным целым числам. В этом случае восьми двоичных разрядов достаточно для кодирования 256 различных символов. Этого хватит, чтобы выразить различными комбинациями восьми битов все символы английского и русского языков, как строчные, так и прописные, а также знаки препинания, символы основных арифметических действий и некоторые общепринятые специальные символы, например символ «$».

Институт стандартизации США (*ANSI – American National Standard Institute*) ввел в действие систему кодирования ASCII (*American Standard Code for Information Interchange – стандартный код информационного обмена США*). В системе ASCII закреплены две таблицы кодирования – *базовая* и *расширенная*. Базовая таблица закрепляет значения кодов от 0 до 127, а расширенная относится к символам с номерами от 128 до 255.

Первые 32 кода базовой таблицы являются *управляющими кодами*, которым не соответствуют никакие символы языков, и, соответственно, эти коды не выводятся ни на экран, ни на устройства печати, но ими можно управлять тем, как производится вывод данных.

Начиная с кода 32 по код 127 размещены коды символов английского алфавита, знаков препинания, цифр, арифметических действий и некоторых вспомогательных символов.

Таблица 1.1. Базовая таблица кодировки ASCII

32	пробел	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	

Аналогичные системы кодирования текстовых данных были разработаны и в других странах. Так, например, в СССР в этой области действовала система кодирования КОИ-7 (*код обмена информацией, семизначный*). Однако поддержка производителей оборудования и программ вывела американскую систему кодирования ASCII на уровень международного стандарта, а

национальным системам кодирования пришлось “отступить” во вторую, расширенную часть системы кодирования, определяющую значения кодов со 128 по 255. При этом отсутствие единого стандарта в этой области привело к множественности одновременно действующих кодировок. Например, в России можно указать три действующих стандарта кодировки и еще два устаревших:

- Windows-1251 – кодировка символов русского языка, введенная компанией Microsoft, для компьютеров, работающих на платформе операционной системы Windows;

Таблица 1.2. Кодировка Windows 1251

128 Ъ	144 ђ	160	176 °	192 А	208 Р	224 а	240 р
129 Ѓ	145 ‘	161 Ў	177 ±	193 Б	209 С	225 б	241 с
130 ‚	146 ’	162 ў	178 І	194 В	210 Т	226 в	242 т
131 ѓ	147 “	163 Ј	179 і	195 Г	211 У	227 г	243 у
132 „	148 ”	164 ¤	180 ґ	196 Д	212 Ф	228 д	244 ф
133 …	149 •	165 Ґ	181 µ	197 Е	213 Х	229 е	245 х
134 †	150 –	166 ¦	182 ¶	198 Ж	214 Ц	230 ж	246 ц
135 ‡	151 —	167 §	183 ·	199 З	215 Ч	231 з	247 ч
136 €	152	168 Ё	184 ё	200 И	216 Ш	232 и	248 ш
137 ‰	153 ™	169 ©	185 №	201 Й	217 Щ	233 й	249 щ
138 Љ	154 љ	170 Є	186 є	202 К	218 Ъ	234 к	250 ъ
139 ‹	155 ›	171 «	187 »	203 Л	219 Ы	235 л	251 ы
140 Њ	156 њ	172 ¬	188 ј	204 М	220 Ь	236 м	252 ь
141 Ќ	157 ќ	173 -	189 S	205 Н	221 Э	237 н	253 э
142 Ћ	158 ћ	174 ®	190 s	206 О	222 Ю	238 о	254 ю
143 Џ	159 џ	175 Ї	191 ї	207 П	223 Я	239 п	255 я

- КОИ-8 (*код обмена информацией, восьмизначный*) - кодировка символов русского языка, имеющая широкое распространение в российском секторе Интернета;

Таблица 1.3. Кодировка КОИ-8

128	144 ▒	160 ─	176 ├	192 ю	208 п	224 Ю	240 П
129 |	145 ▓	161 Ё	177 ├	193 а	209 я	225 А	241 Я
130 ┌	146 ▒	162 ┌	178 ┤	194 б	210 р	226 Б	242 Р
131 ┐	147 ┌	163 ё	179 Ё	195 ц	211 с	227 Ц	243 С
132 └	148 ▪	164 ┌	180 ┤	196 д	212 т	228 Д	244 Т
133 ┘	149 ·	165 ┌	181 ┤	197 е	213 у	229 Е	245 У
134 ├	150 √	166 ┐	182 ┬	198 ф	214 ж	230 Ф	246 Ж
135 ┤	151 ≈	167 ┐	183 ┬	199 г	215 в	231 Г	247 В
136 ┬	152 ≤	168 ┐	184 ┬	200 х	216 ь	232 Х	248 Ь
137 ┴	153 ≥	169 └	185 ┴	201 и	217 ы	233 И	249 Ы
138 ┼	154	170 └	186 ┴	202 й	218 з	234 Й	250 З
139 ▀	155 ⌠	171 └	187 ┴	203 к	219 ш	235 К	251 Ш
140 ▄	156 °	172 ┘	188 ┼	204 л	220 э	236 Л	252 Э
141 ▌	157 ²	173 ┘	189 ┼	205 м	221 щ	237 М	253 Щ
142 ▐	158 ·	174 ┘	190 ┼	206 н	222 ч	238 Н	254 Ч
143 ▀	159 ÷	175 ├	191 ё	207 о	223 ъ	239 О	255 Ъ

- ISO - кодировка символов русского языка по международному стандарту. На практике редко используется;
- ГОСТ и ГОСТ-альтернативная - кодировка символов русского языка для компьютеров, работающих на платформе операционной системы MS-DOS.

В связи с изобилием систем кодирования текстовых данных возникает задача межсистемного преобразования данных – это одна из распространенных задач информатики.

- Unicode (Юникод, или Уникод) — стандарт кодирования символов, позволяющий представить знаки практически всех письменных языков. Стандарт предложен в 1991 году некоммерческой организацией «Консорциум Юникода» (англ. Unicode Consortium), объединяющей крупнейшие IT-корпорации. Применение этого стандарта позволяет закодировать очень большое число символов из разных письменностей: в документах Unicode могут соседствовать китайские иероглифы, математические символы, буквы греческого

алфавита и кириллицы, при этом становятся ненужными кодовые страницы. Коды в стандарте Unicode разделены на несколько областей. Область с кодами от U+0000 до U+007F содержит символы набора ASCII с соответствующими кодами. Далее расположены области знаков различных письменностей, знаки пунктуации и технические символы. Часть кодов зарезервирована для использования в будущем. Под символы кириллицы выделены коды от U+0400 до U+052F. Большинство современных операционных систем в той или иной степени обеспечивают поддержку Юникода. В операционных системах семейства Windows NT для внутреннего представления имён файлов и других системных строк используется двухбайтовая кодировка UTF-16LE. Системные вызовы, принимающие строковые параметры, существуют в однобайтном и двухбайтном вариантах. UNIX-образные операционные системы, в том числе, Linux, BSD, Mac OS X, используют для представления Юникода кодировку UTF-8. Большинство программ могут работать с UTF-8 как с традиционными однобайтными кодировками, не обращая внимания на то, что символ представляется как несколько последовательных байт. Для работы с отдельными символами строки обычно перекодируются в UCS-4, так что каждому символу соответствует машинное слово. Одной из первых успешных коммерческих реализаций Юникода стала среда программирования Java. В ней принципиально отказались от 8-битного представления символов в пользу 16-битного. Сейчас большинство языков программирования поддерживают строки Юникода, хотя их представление может различаться в зависимости от реализации.

0400 **Cyrillic** **04FF**

	040	041	042	043	044	045	046	047	048	049	04A	04B	04C	04D	04E	04F
0	Ѐ 0400	А 0410	Р 0420	а 0430	р 0440	ѐ 0450	Ѡ 0460	Ѱ 0470	ҁ 0480	Ґ 0490	Ҡ 04A0	Ұ 04B0	І 04C0	Ӑ 04D0	З 04E0	Ӱ 04F0
1	Ё 0401	Б 0411	С 0421	б 0431	с 0441	ё 0451	ѡ 0461	ѱ 0471	ҁ 0481	ґ 0491	ҡ 04A1	ұ 04B1	Ӂ 04C1	ӑ 04D1	з 04E1	ӱ 04F1
2	Ђ 0402	В 0412	Т 0422	в 0432	т 0442	ђ 0452	Ѣ 0462	Ѳ 0472	≠ 0482	Ғ 0492	Ң 04A2	Ҳ 04B2	ж 04C2	Ӓ 04D2	Й 04E2	Ӳ 04F2
3	Ѓ 0403	Г 0413	У 0423	г 0433	у 0443	ѓ 0453	ѣ 0463	ѳ 0473	◌҃ 0483	ғ 0493	ң 04A3	ҳ 04B3	Җ 04C3	ӓ 04D3	й 04E3	ӳ 04F3
4	Є 0404	Д 0414	Ф 0424	д 0434	ф 0444	є 0454	Ѥ 0464	Ѵ 0474	◌҄ 0484	Ҕ 0494	Н 04A4	Ҵ 04B4	ҕ 04C4	Æ 04D4	Ӣ 04E4	Ӵ 04F4
5	Ѕ 0405	Е 0415	Х 0425	е 0435	х 0445	ѕ 0455	ѥ 0465	ѵ 0475	◌҅ 0485	ҕ 0495	н 04A5	ҵ 04B5	Ӆ 04C5	æ 04D5	ӣ 04E5	ӵ 04F5
6	І 0406	Ж 0416	Ц 0426	ж 0436	ц 0446	і 0456	Ѧ 0466	Ѷ 0476	◌҆ 0486	Ж 0496	Ҧ 04A6	Ҷ 04B6	ӆ 04C6	Ӗ 04D6	Ӧ 04E6	
7	Ї 0407	З 0417	Ч 0427	з 0437	ч 0447	ї 0457	ѧ 0467	ѷ 0477		ж 0497	ҧ 04A7	ҷ 04B7	Ӈ 04C7	ӗ 04D7	ӧ 04E7	
8	Ј 0408	И 0418	Ш 0428	и 0438	ш 0448	ј 0458	Ѩ 0468	Ѹ 0478	◌҈ 0488	Ҙ 0498	Ҩ 04A8	Ҹ 04B8	ң 04C8	Ә 04D8	Ө 04E8	Ӹ 04F8
9	Љ 0409	Й 0419	Щ 0429	й 0439	щ 0449	љ 0459	ѩ 0469	ѹ 0479	◌҉ 0489	ҙ 0499	ҩ 04A9	ҹ 04B9	Ӊ 04C9	ә 04D9	ө 04E9	ӹ 04F9
A	Њ 040A	К 041A	Ъ 042A	к 043A	ъ 044A	њ 045A	Ѫ 046A	Ѻ 047A	Ҋ 048A	Қ 049A	Ҫ 04AA	Һ 04BA	ӊ 04CA	Ӛ 04DA	Ө 04EA	
B	Ћ 040B	Л 041B	Ы 042B	л 043B	ы 044B	ћ 045B	ѫ 046B	ѻ 047B	ҋ 048B	қ 049B	ҫ 04AB	һ 04BB	Ҍ 04CB	ӛ 04DB	ө 04EB	
C	Ќ 040C	М 041C	Ь 042C	м 043C	ь 044C	ќ 045C	Ѭ 046C	Ѽ 047C	Ҍ 048C	Ҝ 049C	Ҭ 04AC	Ҽ 04BC	ҍ 04CC	Ӝ 04DC	Ӭ 04EC	
D	Ѝ 040D	Н 041D	Э 042D	н 043D	э 044D	ѝ 045D	ѭ 046D	ѽ 047D	ҍ 048D	ҝ 049D	ҭ 04AD	ҽ 04BD	Ӎ 04CD	ӝ 04DD	ӭ 04ED	
E	Ў 040E	О 041E	Ю 042E	о 043E	ю 044E	ў 045E	Ѯ 046E	Ѿ 047E	Ҏ 048E	Ҟ 049E	Ү 04AE	Ҿ 04BE	ӎ 04CE	Ӟ 04DE	Ӯ 04EE	
F	Џ 040F	П 041F	Я 042F	п 043F	я 044F	џ 045F	ѯ 046F	ѿ 047F	ҏ 048F	ҟ 049F	ү 04AF	ҿ 04BF		ӟ 04DF	ӯ 04EF	

Кодирование графических данных

Для кодирования графической информации применяется точечный способ. Кодируемое изображение разбивается на отдельные элементы, называемые *пикселями* (picture element). Чем больше точек, тем точнее будет информация о кодируемом изображении. Информация о каждой точке должна содержать номер(координаты) точки и код цвета(любой цвет может быть представлен суммой RGB)

Поэтому растровое кодирование позволяет использовать двоичный код для представления графических данных. Общепринятым на сегодняшний день считается представление черно-белых изображений в виде комбинаций точек с 256 градациями серого цвета, поэтому для кодирования яркости любой точки достаточно восьмиразрядного двоичного числа.

Для кодирования цветных графических изображений применяется принцип декомпозиции произвольного цвета на основные составляющие. В качестве таких составляющих используют три основные цвета: красный, зеленый и синий. На практике считается, что любой цвет, видимый человеческим глазом, можно получить путем механического смешения этих трех основных цветов. Такая система кодирования называется системой RGB по первым буквам названий основных цветов.

Если для кодирования яркости каждой из основных составляющих использовать по 256 значений (восемь двоичных разрядов), то на кодирование цвета одной точки надо затратить 24 разряда. При этом система кодирования обеспечивает однозначное определение 16,5 млн различных цветов, что на самом деле близко к чувствительности человеческого глаза. Режим представления цветной графики с использованием 24 двоичных разрядов называется полноцветным (True Color).

Кодирование звуковой информации

Приемы и методы работы со звуковой информацией пришли в вычислительную технику наиболее поздно. В итоге методы кодирования звуковой информации двоичным кодом далеки от стандартизации. Множество компаний разработали свои стандарты, но если говорить обобщенно, то можно выделить два основных направления.

Метод FM (Frequency Modulation) основан на том, что теоретически любой сигнал можно разложить на последовательность простых гармонических сигналов разных частот, каждый из которых представляет собой правильную синусоиду, а, следовательно, может быть описан числовыми параметрами, то есть кодом. Разложение аналоговых звуковых сигналов в гармонические ряды и представление в виде дискретных цифровых сигналов выполняют специальные устройства – аналого-цифровые преобразователи (АЦП). Обратное преобразование для воспроизведения звука, закодированного числовым кодом, выполняют цифро-аналоговые преобразователи (ЦАП). При таких преобразованиях неизбежны потери информации, связанные с методом кодирования, поэтому качество звукозаписи получается не вполне удовлетворительным и соответствует качеству звучания простейших электромузыкальных инструментов. В то же время данный метод кодирования обеспечивает компактный код, поэтому он нашел применение в те годы, когда ресурсы средств вычислительной техники были явно недостаточны.

Метод таблично-волнового (Wave-Table) ***синтеза*** лучше соответствует современному уровню развития техники. В заранее подготовленных таблицах хранятся образцы звуков для множества различных музыкальных инструментов. Такие образцы называются ***сэмплами***. Числовые коды выражают тип инструмента, номер его модели, высоту тона, продолжительность и интенсивность звука, динами его изменения, некоторые параметры среды, в которой происходит звучание, а также прочие параметры, характеризующие особенности звука. Поскольку в качестве образцов используются реальные звуки, то качество звука, полученного в

результате синтеза, получается очень высоким и приближается к качеству звучания реальных музыкальных инструментов.

Кодирование целых и действительных чисел

Целые числа кодируются двоичным кодом достаточно просто – необходимо целое число делить пополам до тех пор, пока в остатке не образуется ноль или единица. Совокупность остатков от каждого деления, записанная справа налево вместе с последним остатком, и образует двоичный аналог десятичного числа.

Таким образом, $23_{10} = 10111_2$.

Для кодирования целых чисел от 0 до 255 требуется иметь 8 разрядов двоичного кода (8 бит). Шестнадцать бит позволяют закодировать целые числа от 0 до 65535, 24 бита – более 16,5 миллионов разных значений.

Для кодирования действительных чисел используют 80-разрядное кодирование. При этом число предварительно преобразуется в нормализованную форму:

$3{,}1416=0{,}31416*10^1$

$300000=0{,}3*10^6$

Первая часть числа называется *мантиссой*, а вторая – *характеристикой*. Большую часть из 80 бит отводят для хранения мантиссы (вместе со знаком) и некоторое фиксированное количество разрядов отводят для хранения характеристики (тоже со знаком).

2. СИСТЕМЫ СЧИСЛЕНИЯ, ПРИМЕНЯЕМЫЕ В ПЭВМ

Системой счисления называется совокупность приемов наименования и обозначения (записи) чисел для их однозначного толкования. Условные знаки, применяемые при обозначении чисел, называют *цифрами*. Наиболее часто в повседневной жизни мы используем для изображения чисел арбские цифры (символы 0,1,2,3,4,5,6,7,8,9) и римские цифры (символы I,V,X,L,C и др.).

В зависимости от способа изображения чисел различают *непозиционные* и *позиционные* системы счисления**.**

В *непозиционных системах счисления* значения цифр не зависят от их местоположения (позиции) в числе. Примером является римская система счисления: XX = X + X = 20, XVII = 10 + 5 + 1 + 1 = 17, то есть символ X имеет величину десять независимо от того, какую позицию этот символ занимает в числе.

Кириллическая система счисления — система счисления Древней Руси, основанная на алфавитной записи чисел с использованием кириллицы или глаголицы. В основных чертах схожа с греческой системой счисления. Использовалась в России до начала XVIII века, когда была заменена на систему счисления, основанную на арабских цифрах. В настоящее время используется в книгах на церковнославянском языке.

Башенные часы с кириллическими числами в Суздале

1	2	3	4	5	6	7	8	9
А	В	Г	Д	Е	Ѕ	З	И	Ѳ
10	20	30	40	50	60	70	80	90
І	К	Л	М	Н	Ѯ	О	П	Ч
100	200	300	400	500	600	700	800	900
Р	С	Т	У	Ф	Х	Ѱ	Ѡ	Ц
11	12	13	14	15	16	17	18	19
аі	ві	гі	ді	еі	ѕі	зі	иі	ѳі
222		319		431		988		
СКВ		ТѲІ		УЛА		ЦПИ		
222		319		431		988		
1000		2000		20000		43000		
А		В		К		МГ		
10000		300000		4000000		80000000		
А		Г		Д		И		

Примеры записи чисел кириллицей

1 α	10 ι	100 ρ
2 β	20 κ	200 σ
3 γ	30 λ	300 τ
4 δ	40 μ	400 υ
5 ε	50 ν	500 φ
6 ς	60 ξ	600 χ
7 ζ	70 ο	700 ψ
8 η	80 π	800 ω
9 θ	90 ϟ	900 ϡ

Запись чисел в греческой системе счисления

Такие системы счисления не нашли применение в ЭВМ вследствие громоздкости записи больших и малых (дробных) чисел и трудностей при выполнении арифметических операций. Поэтому в дальнейшем будем вести речь только о позиционных системах счисления.

В *позиционных системах счисления* значения цифр зависят от их места (позиции) в числе. Примером является десятичная система : $A_{10} = 131 = 100 + 30 + 1$; здесь первый символ 1 имеет величину сотни, а третий точно такой же символ 1 имеет величину единицы.

В современных ЭВМ используются только позиционные системы счисления.

Кроме того, позиционные системы, в свою очередь, разделяют на однородные и смешанные. Во всех разрядах числа, представленного в однородной системе, используются цифры из одного и того же множества. Например, в десятичной системе во всех разрядах любого числа используются цифры из множества {0, 1, …, 9}, в двоичной системе – цифры из множества {0, 1} и т.п. В смешанных системах множества цифр различны для разных разрядов. Примерами смешанных систем являются система для измерения углов и дуг (в разрядах минут и секунд могут быть использованы 60 различных цифр, в разряде градусов – 360 различных цифр), система измерения времени, например, в секундах, минутах, часах, сутках, неделях, система английских денежных единиц и др.

Основными характеристиками позиционных систем счисления являются:

- *основание* системы счисления q;
- *значения цифр* (символов) a_k ;
- *вес разряда* (позиции) в числе R_j, где j – номер разряда.

Основанием системы счисления (q) называется количество различных цифр, которые используются для изображения чисел. Эта величина всегда является целым положительным числом, большим единицы. Например, для десятичной системы счисления ее основание равно десяти, так как для изображения чисел используется десять символов-цифр – от 0 до 9.

Значения цифр (a_k) системы счисления в общем случае могут быть любыми, но их обычно выбирают так, чтобы они составляли отрезок натурального ряда чисел, включая нуль.

В позиционных системах счисления каждому разряду (позиции) числа присваивается определенный *весовой коэффициент* R_j (вес), где j – номер разряда. Нулевой номер (j=0) принадлежит такому разряду числа, который имеет самый меньший целый вес (для десятичной системы счисления по порядку справа налево (j=1, 2, …)). Разряды дробной части числа нумеруются отрицательными числами (j=-1, -2, …) слева направо.

Веса разрядов определяют в соответствии с выражением $\mathbf{R_j = q^j}$. Например, для десятичной системы счисления

$R_0 = 10^0 = 1$; $R_1 = 10^1 = 10$; $R_2 = 10^2 = 100$; $R_{-1} = 10^{-1} = 0{,}1$;

Это правило обеспечивает простоту арифметических действий над числами, простоту округления и т. п.

Для определения величины числа в любой позиционной системе счисления необходимо цифру a_k каждого разряда умножить на соответствующий данному разряду вес ($R_j = q^j$) и просуммировать полученные произведения.

Например: $1961{,}56 = 1*10^3 + 9*10^2 + 6*10^1 + 1*10^0 + 5*10^{-1} + 6*10^{-2}$.

В общем случае величина числа в позиционной системе счисления представляется в виде:

$$A_q = a_{n-1}q^{n-1} + \ldots + a_0 q^0 + a_{-1}q^{-1} + \ldots + a_m q^{-m} = \sum_{k=-m}^{n-1} a_k \cdot q^k, \quad (3.1)$$

где

k – номер разряда числа;

m – количество разрядов дробной части числа;

n – количество разрядов в целой части числа;

a_k – значение цифры в k-м разряде.

При выполнении вычислений основание системы счисления строго определено, оно одно и то же для всех чисел. Поэтому для представления чисел нет необходимости указывать величину q (она подразумевается) и числа изображаются в кратком (символическом) виде:

$$A_k : a_{n-1}a_{n-2}\cdots a_0 a_{-1}\cdots a_{-m} = \sum_{k=-m}^{n-1} a_k.$$

В настоящее время в ПЭВМ наибольшее распространение нашли следующие позиционные системы счисления:

1. *Двоичная система счисления* ($q = 2_{10} = 10_2$). В этой системе счисления используются цифры 0 и 1, а величина k-разрядного числа определяется суммой степени двойки:

$$A_2 = \sum_{k=-m}^{n-1} a_k \cdot 10_2^k, где\ a_k \in \{0,1\}.$$

При этом при записи числа, обычно используют только цифры a_k:

$$A_2 = a_{n-1}\cdots a_0, a_{-1}\cdots a_{-m}.$$

Например:

$$A_2 = 110{,}01 = 1\cdot 10_2^{10} + 1\cdot 10_2^1 + 0\cdot 10_2^0 + 0\cdot 10_2^{-1} + 1\cdot 10_2^{-10} =$$

$$= 1\cdot 2_{10}^2 + 1\cdot 2_{10}^1 + 0\cdot 2_{10}^0 + 0\cdot 2_{10}^{-1} + 1\cdot 2_{10}^{-2} = 6{,}25_{10}.$$

2.*Шестнадцатеричная система счисления* ($q = 16_{10} = 10_{16}$). Для изображения чисел используется 16 символов от 0 до q-1. Поскольку известно только 10 арабских цифр, то цифры от десяти до пятнадцати не имеют соответствующих им арабских аналогов. Эти цифры изображаются следующим образом: $10_{10} = A_{16}$, $11_{10} = B_{16}$, $12_{10} = C_{16}$, $13_{10} = D_{16}$, $14_{10} = E_{16}$, $15_{10} = F_{16}$.

Например, запись $E6_{16}$ соответствует шестнадцатеричному числу, величина которого, представленная в десятичной системе счисления, равна:

$$14\cdot 16_{10}^1 + 6\cdot 16_{10}^0 = 230_{10}.$$

При этом величина числа определяется суммой степеней 16:

$$A_{16} = \sum_{k=-m}^{n-1} a_k \cdot 10_{16}^k,$$

$$a_k \in \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}.$$

Троичная система счисления — позиционная система счисления с основанием 3. Существует в двух вариантах: несимметричная (цифры 0, 1, 2) и симметричная (цифры −1, 0, 1). Симметричная система позволяет изображать отрицательные числа, не используя отдельный знак минуса.

«Сéтунь» — малая ЭВМ на основе троичной логики, разработанная в вычислительном центре Московского государственного университета в 1959 году.

Десятичная система	3	2	1										
Троичная несимметричная	10	2	1				0	1	2	0	1	2	00
Троичная симметричная	0	1				1	0	1	11	10	11	01	00

В общем случае система счисления, применяемая в ЭВМ, должна обеспечивать:

- простоту технической реализации q-позиционного запоминающего элемента. С этой точки зрения наиболее удобна система с наименьшей величиной q, то есть **двоичная**;
- наибольшую помехоустойчивость кодирования цифр информации на носителях информации. Очевидно, что чем меньше состояний имеет цифра, тем лучше эти состояния различать. Наименьшее число состояний у **двоичной** системы счисления (два состояния);
- минимум затрат оборудования при построении узлов и блоков ЭВМ. Расчеты показывают, что наименьшую сложность обеспечивает **троичная** система счисления, затем идет двоичная система счисления. Далее увеличение основания системы счисления приводит к увеличению сложности устройств;
- простоту арифметических действий, а следовательно, и простоту управления этими действиями. Чем меньше цифр в системе счисления, тем проще арифметические действия. Например, таблица сложения двоичных чисел имеет вид:

  0+0=0; 0+1=1; 1+0=1; 1+1=10.

  Таким образом, предпочтение отдается **двоичной** системе счисления.
- наибольшее быстродействие при выполнении операций обеспечивается с увеличением основания q системы счисления, так как один и тот же диапазон чисел представляется меньшим числом разрядов и уменьшается время распространения переносов;
- удобство работы человека с ЭВМ.

Анализ различных систем счисления с рассмотренных позиций показывает, что в наибольшей степени перечисленным требованиям удовлетворяет двоичная система счисления. Она нашла применение во всех ПЭВМ.

2. СПОСОБЫ ПЕРЕВОДА ЧИСЕЛ ИЗ ОДНОЙ ПОЗИЦИОННОЙ СИСТЕМЫ СЧИСЛЕНИЯ В ДРУГУЮ

При работе с ПЭВМ необходимо уметь переводить числа из одной системы счисления в другую. Существуют два основных метода перевода чисел из одной системы в другую: табличный и расчетный.

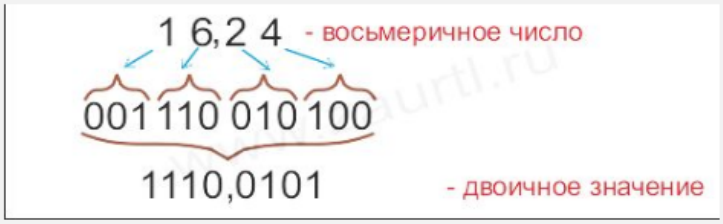
Табличный метод основан на составлении специальных таблиц соответствия чисел в различных системах счисления. Табличный метод удобен на начальном этапе работы с ПЭВМ. Так, при изучении двоичной системы счисления полезно запомнить двоичные эквиваленты десятичных цифр.

10-я	2-я	8-я	16-я
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9

10-я	2-я	8-я	16-я
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13

Для перевода числа из восьмеричной системы счисления в двоичную необходимо каждую цифру этого числа записать трехразрядным двоичным числом (триадой).

Пример: записать число 16,24(8) в двоичной системе счисления.



Ответ: 16,24(8)= 1110,0101(2) .

Примечание: незначащие нули слева для целых чисел и справа для дробей не записываются.

Для обратного перевода двоичного числа в восьмеричную систему счисления, необходимо исходное число разбить на триады влево и вправо от запятой и представить каждую группу цифрой в восьмеричной системе счисления. Крайние неполные триады дополняют нулями.

Пример: записать число 1110,0101(2) в восьмеричной системе счисления.

Ответ: 1110,0101(2)= 16,24(8) .

Для перевода числа из шестнадцатеричной системы счисления в двоичную необходимо каждую цифру этого числа записать четырехразрядным двоичным числом (тетрадой).

Пример: записать число 7A,7E(16) в двоичной системе счисления.

Ответ: 7A,7E(16)= 1111010,0111111(2) .

Для обратного перевода двоичного числа в шестнадцатеричную систему счисления, необходимо исходное число разбить на тетрады влево и вправо от запятой и представить каждую группу цифрой в шестнадцатеричной системе счисления. Крайние неполные триады дополняют нулями.

Пример: записать число 1111010,0111111(2) в шестнадцатеричной системе счисления.

Ответ: 1111010,0111111(2)= 7A,7E(16) .

Расчетный метод более универсален. При использовании расчетного метода могут встретиться три случая: перевод целых чисел, перевод правильных дробей, перевод неправильных дробей.

Перевод целых чисел из одной позиционной системы счисления в другую осуществляется по следующим правилам. Исходное целое число делится на основание новой системы счисления до тех пор, пока не получится частное, у которого целая часть равна нулю. Деление необходимо

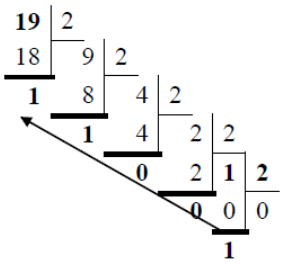
производить в той системе счисления, в которой задано исходное число. Число в новой системе счисления записывается из остатков от последовательного деления, причем последний остаток будет старшей цифрой нового числа.

Пример.

Переведите число 19_{10} в двоичную систему счисления.

Решение

Последовательно делим исходное десятичное число и получаемые частные на основание системы (в данном задании – 2) нацело до тех пор, пока не получится частное, равное нулю. Полученные остатки от целочисленного деления записываем в обратной последовательности.



Ответ: (19)10 = (10011)2

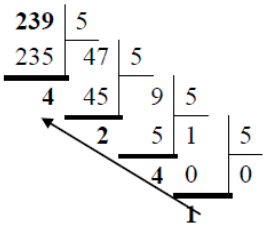
Стрелкой показан порядок записи числа в новой системе счисления.

Пример.

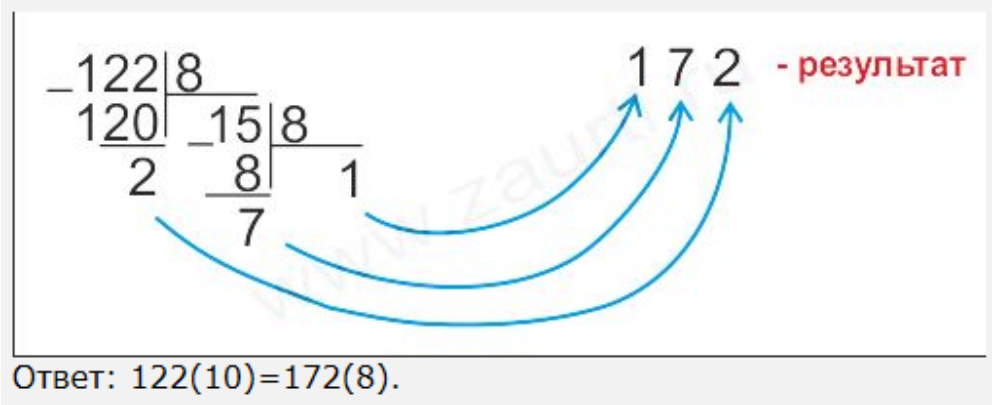
Переведите число 239_{10} в пятиричную систему счисления.

Решение

Последовательно делим исходное десятичное число и получаемые частные на основание системы (в данном задании – 5) нацело до тех пор, пока не получится частное, равное нулю. Полученные остатки от целочисленного деления записываем в обратной последовательности.



Ответ: (239)10 = (1424)5



Перевод правильных дробей осуществляется путем последовательного умножения исходного числа на основание новой системы счисления. Умножение на основание новой системы счисления производится до тех пор, пока в новой дроби не будет нужного количества цифр, которое определяется требуемой точностью представления дроби. Правильная дробь в

новой системе счисления записывается из целых частей произведений, получающихся при последовательном умножении, причем первая целая часть будет старшей цифрой новой дроби. Умножение выполняется в той системе, в которой представлено исходное число.

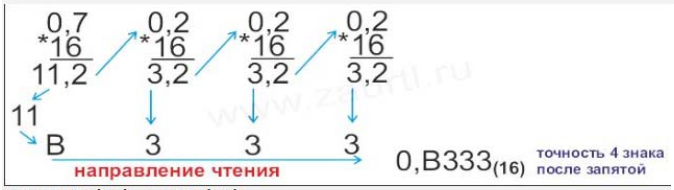
Например, перевести число $A = 0{,}675_{10}$ в двоичную систему счисления:

0	675 × 2
1	350 × 2
0	700 × 2
1	400

Ответ: 0,625(10)=0,101(2).

Стрелка показывает порядок записи дроби в новой системе.

Ответ: 0,6(10)=0,463(8).



Ответ: 0,7(10)=0,B333(16).

При *переводе неправильных дробей* отдельно преобразуют целую и дробную части по соответствующим правилам, приведенным выше, а затем записывают их через запятую в новой системе счисления.

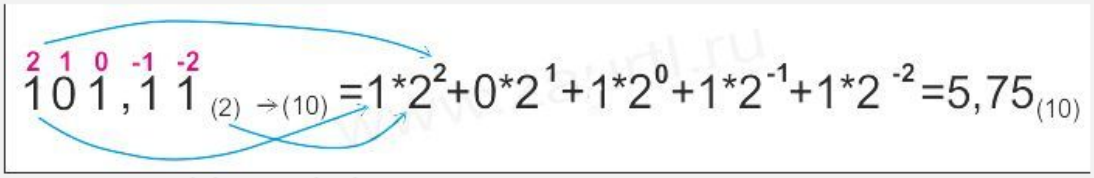
Например, $A = 19{,}675_{10} = 10011{,}101_2$.

Рассмотренные методы удобны в том случае, если исходной системой счисления является десятичная. Если же перевод осуществляется из недесятичной системы, то вычисления затруднительны. В этом случае для преобразования чисел можно воспользоваться формулой (3.1). Расчеты ведутся в новой системе счисления.

Например, перевести число $A = 101110_2$ в десятичную систему счисления:

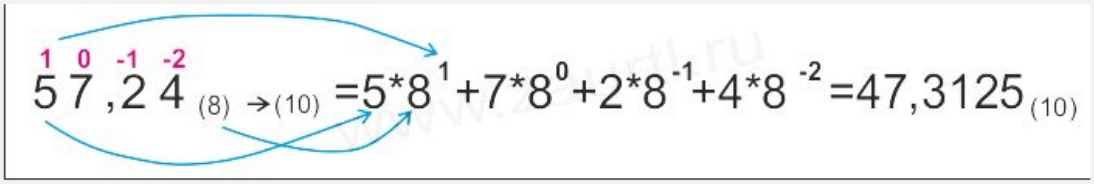
$101110_2 = 1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 0*2^0 = 46_{10}$.

Пример 1. Перевести число 101,11(2) в десятичную систему счисления.



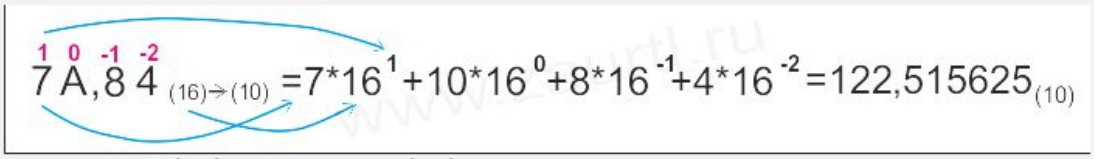
Ответ: 101,11(2)= 5,75(10) .

Пример 2. Перевести число 57,24(8) в десятичную систему счисления.



Ответ: 57,24(8) = 47,3125(10) .

Пример 3. Перевести число 7A,84(16) в десятичную систему счисления.



3. ФОРМЫ ПРЕДСТАВЛЕНИЯ ЧИСЕЛ В ПЭВМ

В ЭВМ для представления каждого числа используется стандартная разрядная сетка, состоящая из заданного количества двоичных разрядов.

Под *разрядной сеткой* понимают совокупность двоичных разрядов, используемых в ЭВМ для представления одного числа. Знак числа является двузначной величиной, и поэтому его можно обозначить нулем или единицей, имея для этого соответствующий разряд. Обычно знак «плюс» изображается нулем в специальном знаковом разряде, а знак «минус» - единицей в этом же разряде.

В зависимости от способа использования разрядной сетки различают две формы представления чисел: естественную форму (с фиксированным положением точки) и нормальную форму (с плавающей точкой).

В *естественной форме* числа представляются последовательностью цифр, разделенных запятой на целую и дробную части. Весовые коэффициенты разрядов здесь фиксированы, поэтому и положение точки фиксировано.

ЭВМ, в которых используется естественная форма представления чисел, называется ЭВМ с фиксированной точкой или фиксированной запятой. Наиболее часто ЭВМ с фиксированной точкой оперируют с числами, меньшими единицы, у которых точка фиксируется перед старшим разрядом. Для знака числа выделяется дополнительный разряд. Знак числа веса не имеет. Так, двоичное число +0,10101 в разрядной сетке запишется в виде:

0	1	0	1	0	1

а число -0,10101 – в виде:

1	1	0	1	0	1

В общем случае m-разрядное число в разрядной сетке ЭВМ записывается в виде:

знак	a_{-1}	a_{-2}	a_{-3}	…	a_{-m}

При таком способе фиксации точки минимальное по модулю число, отличное от нуля, представимое в ЭВМ, равно:

$|A|_{\min} = 2^{-m} = 0{,}00...1_2$,

а максимальное

$|A|_{\max} = 1 - 2^{-m} = 0{,}11...1_2$.

Следовательно, диапазон чисел одного знака, представимых в такой ЭВМ, составит:

$2^{-m} \le |A| \le 1 - 2^{-m}$.

Для представления в таких ЭВМ чисел по абсолютной величине больших 1 вводятся соответствующие масштабные коэффициенты. Обычно естественная форма представления чисел применяется в специализированных ЭВМ.

В *нормальной форме* числа представляются с помощью мантиссы и порядка в виде:

$A_q = M_A q^L$,

где M_A – мантисса числа A; q – основание системы счисления; L – порядок числа A.

Например, десятичное число $A = 175_{10}$ может быть представлено в виде:

$175_{10} = 0{,}175*10^3 = 0{,}0175*10^4 = 1750*10^{-1}$.

Аналогично в двоичной системе счисления:

$10{,}101_2 = 0{,}10101*10^{10} = 101{,}01*10^{-01} = 10101*10^{-11}$.

Таким образом, при изображении числа в нормальной форме положение точки не фиксировано. Место точки указывает порядок числа. В разрядной сетке ЭВМ помещаются две группы цифр: мантисса со знаком и порядок со своим знаком. Основание системы счисления в разрядной сетке не указывается.

Обычно мантиссы чисел, представляемых в ЭВМ, выбирают меньше 1, и при этом всегда в старший разряд заносится 1, то есть мантисса записывается в *нормализованном виде*.

Таким образом, величина мантиссы лежит в пределах:

$q^{-1} \le |M| \le 1$.

Числа в нормализованной форме представляются в разрядной сетке ЭВМ в виде:

±	1	2	…	k-1	k	±	1	2	…	p-1	p

k+1 разрядов мантиссы p+1 разрядов порядка

Диапазон представимых при этом чисел лежит в пределах:

$2^{-2^p} \le A \le 2^{(2^p-1)}$.

Из этой формулы видно, что практически диапазон чисел зависит только от количества разрядов порядка.

Для типового микропроцессора под мантиссу отводится 24 разряда (1 знаковый), а под порядок 8 разрядов (1 знаковый).

Обычно ЭВМ оперируют с нормализованными числами. При этом нормализация выполняется автоматически.

4. СПОСОБЫ КОДИРОВАНИЯ ДВОИЧНЫХ ЧИСЕЛ В ПЭВМ

Необходимость кодирования двоичных чисел в ЭВМ связана с двумя основными причинами:

- ✓ необходимо кодировать знак числа;
- ✓ для упрощения выполнения операции сложения отрицательных чисел.

Числа, представляемые в естественной и нормальной формах, кодируются в ЭВМ с помощью специальных машинных кодов:

- прямого;
- дополнительного;
- обратного.

Прямой код является простейшим машинным кодом и получается при кодировании в числе только знаковой информации. Прямой код положительного числа совпадает с его изображением в естественной форме. Прямой код отрицательного числа совпадает с его изображением в естественной форме, за исключением знакового разряда, в который ставится единица.

Общая формула при этом имеет вид:

$$A_{пр} = \begin{cases} A, \text{ если } A \geq 0, \\ 1\text{-}A, \text{ если } A < 0. \end{cases}$$

$A = -0{,}101 \rightarrow Апр = 1 – (-0{,}101) = 1.101$

Например:

$A = 0{,}10110 \rightarrow A_n = 0.10110$

$A = -\ 0{,}01101 \rightarrow A_n = 1.01101$

Знак числа веса не имеет. Код получается добавлением знака числа к двоичному коду. Знак отделяется от основного числа точкой.

Следует отметить, что “ноль” в прямом коде имеет два значения: положительное и отрицательное, т.е.:

$+ 0{,}00 \ldots 0 \rightarrow 0.00 \ldots 0$,

$- 0{,}00 \ldots 0 \rightarrow 1.00 \ldots 0$.

Прямой код обычно применяется при сложении и умножении положительных чисел, для записи и хранения чисел в запоминающем устройстве, при вводе информации и выводе результатов.

Недостатком кодирования чисел в прямом коде является то, что правила счета оказываются разными для положительных и отрицательных чисел. Поэтому в ЭВМ операцию вычитания (сложения чисел с разными знаками) заменяют операцией сложения чисел в специальных кодах, пропорциональных исходным числам. Такими кодами являются *дополнительный* и *обратный*.

Обратный код является инверсией числа и образуется путем замены его цифр взаимно обратными.

Обратный код числа образуется в соответствии с формулой:

$$A_{обр} = \begin{cases} A, \text{ если } A \geq 0 \\ 10_2 - 10^{-n} + A, \text{ если } A < 0 \end{cases}$$

где n – количество разрядов дробной части числа; 10^{-n} – единица младшего разряда числа A.

Например:

$A = +0{,}1011 \rightarrow A_{обр} = 0.1011$

$A = -0{,}1011 \rightarrow A_{обр} = 10 – 0{,}0001 + (– 0{,}1011) = 1.0100$

Из примера видно, что изображение положительного числа в обратном коде совпадает с его изображением в прямом коде. Обратный код отрицательного числа образуется заменой значащих цифр исходного числа на обратные и постановкой в знаковый разряд единицы.

Из формулы видно, что нуль в обратном коде имеет два изображения:

$A = +0{,}00\ldots0 \rightarrow A_{обр} = 0.00\ldots0$

$A = -0{,}00\ldots0 \rightarrow A_{обр} = 1.11\ldots1$

Это следует иметь в виду при сравнении чисел в ЭВМ.

Переход от обратного кода к прямому осуществляется по тому же правилу, что и от прямого к обратному.

Например: $A_{пр} = 1.0011 \rightarrow A_{обр} = 1.1100$

Дополнительный код находится как дополнение модуля отрицательного числа до некоторого граничного числа, представимого в данной ЭВМ.

Дополнительный код образуется в соответствии с формулой:

$$A_{доп} = \begin{cases} A, & \text{если } A \geq 0, \\ 10_2 + A, & \text{если } A < 0. \end{cases}$$

Например: $A = +0{,}1101 \rightarrow A_{доп} = 0.1101$

$A = -0{,}1101 \rightarrow A_{доп} = 10 + (-0{,}1101) = 1.0011$

Из формул и примера видно, что изображение положительных чисел в дополнительном коде совпадает с изображением его в прямом коде. Для представления отрицательного числа в дополнительном коде необходимо в знаковом разряде поставить единицу и произвести обращение всех разрядов числа (единицы заменить нулями, а нули – единицами) и к полученному коду прибавить единицу младшего разряда. Если дополнительный код отрицательного числа образуется из прямого, то знаковый разряд остается неизменным, а указанные преобразования осуществляются только с цифровыми разрядами.

Обратное преобразование, т.е. получение числа или прямого кода из дополнительного, осуществляется по тем же правилам, что и получение дополнительного кода. Например:

$A_{доп} = 1.1011 \rightarrow A_{пр} = 1.0101$

Обратный и дополнительный коды используются при выполнении операций сложения отрицательных чисел

Операнды	Дополнительный код	Обратный код
$A_2 = 0,10101$ $B_2 = 0,00111$	$[A]_д = 0.10101$ $+$ $[B]_д = 0.00111$ $[C]_п = [C]_д = 0.11100$	$[A]_о = 0.10101$ $+$ $[B]_о = 0.00111$ $[C]_п = [C]_о = 0.11100$
$A_2 = 0,10101$ $B_2 = -0,00111$	$[A]_д = 0.10101$ $+$ $[B]_д = 1.11001$ 10.01110 Отбрасывается $[C]_п = [C]_д = 0.01110$	$[A]_о = 0.10101$ $+$ $[B]_о = 1.11000$ 10.01101 $\rightarrow 1$ $[C]_п = [C]_о = 0.01110$
$A_2 = -0,10101$ $B_2 = 0,00111$	$[A]_д = 1.01011$ $+$ $[B]_д = 0.00111$ $[C]_д = 1.10010$ Преобразование кода $[C]_п = 1.01110$	$[A]_о = 1.01010$ $+$ $[B]_о = 0.00111$ $[C]_о = 1.10001$ Преобразование кода $[C]_п = 1.01110$
$A_2 = -0,10101$ $B_2 = -0,00111$	$[A]_д = 1.01011$ $+$ $[B]_д = 1.11001$ 11.00100 Отбрасывается $[C]_д = 1.00100$ Преобразование кода $[C]_п = 1.11100$	$[A]_о = 1.01010$ $+$ $[B]_о = 1.11000$ 11.00010 $\rightarrow 1$ $[C]_о = 1.00011$ Преобразование кода $[C]_п = 1.11100$

Методы определения переполнения разрядной сетки при сложении

Переполнение разрядной сетки при сложении – это искажение результата (суммы) вследствие невозможности его представления в заданном формате данных.

Рассмотрим два примера.

$[A]_п = [A]_д = +\mathbf{0}.10001$
$[B]_п = [B]_д = \mathbf{0}.11000$
$[A+B]_д = \mathbf{1}.01001$

$[A]_п = 1.10101$
$[B]_п = 1.01111$

$[A]_д = +\mathbf{1}.01011$
$[B]_д = \mathbf{1}.10001$
$[A+B]_д = \mathbf{0}.11100$

Рис. 1

Переполнение при сложении чисел с фиксированной точкой возникает только в случае равенства знаков операндов и выходе значения суммы модулей за пределы диапазона представимых чисел.

Для примера на рис. 2 диапазон представимых чисел (при интерпретации целыми числами) составляет -32...31, а значение суммы для первого случая должно быть равным (41), а для второго – (-36).

Для обнаружения переполнения разрядной сетки используются следующие методы.

1. Сравнение знаковых разрядов операндов со знаком суммы при равенстве знаковых разрядов слагаемых.

2. Сравнение переносов в знаковый разряд и из знакового разряда.

При наличии или отсутствии обоих переносов результат правильный, в противном случае – переполнение разрядной сетки.

3. Использование модифицированных кодов, в которых для представления знака числа используется два двоичных разряда.

Знак ‘+’ кодируется двумя нулями (“00”), знак ‘–’ двумя единицами. Наличие в знаковых разрядах результата комбинаций “01” или “10” свидетельствует о переполнении разрядной сетки (рис. 3).

$$[A]^{м}_{п} = [A]^{м}_{д} = \mathbf{00.}10001 \qquad [A]^{м}_{п} = 11.10101 \qquad [A]^{м}_{д} = \mathbf{11.}01011$$
$$[B]^{м}_{п} = [B]^{м}_{д} = \mathbf{00.}11000 \qquad [B]^{м}_{п} = 11.01111 \qquad [B]^{м}_{д} = \mathbf{11.}10001$$
$$[A+B]^{м}_{д} = \mathbf{01.}01001 \qquad [A+B]^{м}_{д} = \mathbf{10.}11100$$

Рис.2

Перевести число 100 из 13-ричной системы счисления в двоичную:

Для этого переведем его сначала в десятичную вот так :

$100_{13} = 1 \cdot 13^2 + 0 \cdot 13^1 + 0 \cdot 13^0 = 169 + 0 + 0 = 169_{10}$

Получилось: 169_{10}

Переведем 169_{10} в двоичную систему вот так:

Целая часть числа находится делением на основание новой

169	2							
-168	84	2						
1	-84	42	2					
	0	-42	21	2				
		0	-20	10	2			
			1	-10	5	2		
				0	-4	2	2	
					1	-2	1	
						0		

Направление взгляда

Получилось: $169_{10} = 10101001_2$

Результат перевода:

$100_{13} = 10101001_2$

Перевести число 10010 из двоичной системы счисления в 13-ричную:

Для этого переведем его сначала в десятичную вот так :

$10010_2 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 16 + 0 + 0 + 2 + 0 = 18_{10}$

Получилось: 18_{10}

Переведем 18_{10} в 13-ричную систему вот так:

Целая часть числа находится делением на основание новой

18	13
-13	1
5	

Направление взгляда

Получилось: $18_{10} = 15_{13}$

Результат перевода:

$10010_2 = 15_{13}$

ОДНОМЕРНЫЕ МАССИВЫ

1. Объявление массива
2. Инициализация элементов массива
3. Доступ к отдельным элементам массива
4. Границы массива
5. Заполнение массива
6. Объем памяти, занятой массивом
7. Сравнение двух массивов
8. Реализация метода Горнера
9. Действия с n-мерными векторами

В лекции рассматриваются особенности использования массивов при составлении программ на языке С++.

Массивы позволяют удобным образом организовать размещение и обработку больших объемов информации.

Массив *представляет собой набор однотипных объектов, имеющих общее имя и различающихся местоположением в этом наборе (или индексом, присвоенным каждому элементу массива). Элементы массива занимают один непрерывный участок памяти компьютера и располагаются последовательно друг за другом.*

Одномерный массив – это список связанных переменных. Такие списки довольно популярны в программировании. Например, одномерный массив можно использовать для хранения учетных номеров активных пользователей сети или результатов игры любимой футбольной команды. Если потом нужно усреднить какие-то данные (содержащиеся в массиве), то это очень удобно сделать посредством обработки значений массива.

1. Объявление массива

формат 1:
тип имя[ ];
int a[ ];

формат 2:
тип имя[размер];
int a[10];

Тип задает тип элементов объявляемого массива.

Имя – имя элементов массива. Квадратные скобки указывают на то, что объявляется массив.

Размер в квадратных скобках задает количество элементов в массиве.

```
int b[10]; //объявляется массив целых чисел,
           //состоящий из 10 элементов с именем b
```

Компилятор **гарантирует** размещение элементов массива в смежных ячейках памяти!

Размер массива при объявлении массива может быть опущен в следующих случаях:
– при объявлении массив инициализируется;
– массив объявлен как формальный параметр;
– массив объявлен как ссылка на массив, явно определенный в другом файле.

2. Инициализация элементов массива

При объявлении массива может быть выполнена инициализация его элементов, то есть присвоение начальных значений его элементам при объявлении. Список инициализирующих значений состоит из констант, разделенных запятыми. Весь список заключен в фигурные скобки. Количество значений в инициализаторе не должно быть больше размера массива. Если значений в инициализаторе меньше, остальным элементам массива присваивается 0. Каждая константа должна иметь тип, совместимый с типом массива.

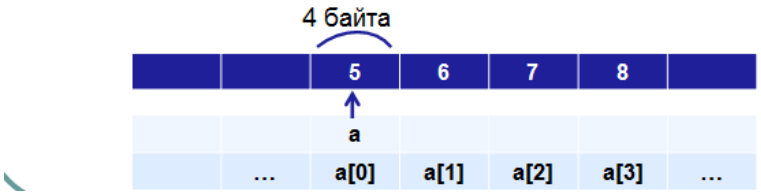
Существуют *две формы инициализации элементов массива*:

1) явное указание числа элементов массива и список начальных значений, возможно, с меньшим числом элементов. Например:

```
float x[10]={5.1,1.3,4.7,-17.4};
```

Здесь описывается массив из 10 элементов. Первые четыре элемента массива инициализируются значениями 5.1, 1.3, 4.7, -17.4. Значение остальных шести элементов не определено, они инициализируются значением 0. Если список начальных значений содержит больше элементов, чем число в квадратных скобках, компилятор генерирует сообщение об ошибке;

```
const int n=4;
int   a[n] = {5, 6,7,8};
```

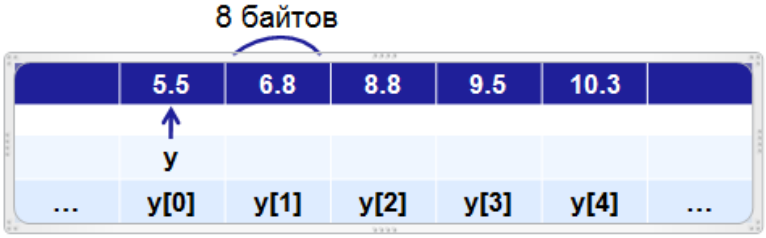


2) только со списком начальных значений. Компилятор определяет число элементов массива по списку инициализации. Например:

```
int x[]={1,22,138};
```

В результате создается массив из трех элементов целого типа и эти элементы получают начальные значения из списка инициализации. Массивы, объявленные без указания границы изменения индекса, называют безразмерными.

```
double y [] = {5.5, 6.8, 8.8, 9.5, 10.3};
```



Если при описании массива отсутствуют значение в квадратных скобках и список начальных значений, компилятор обычно регистрирует ошибку.

Однако в тех случаях, где *не требуется резервирование места в памяти*, этого не происходит:

1) в определении функции, если указатель на первый элемент массива является аргументом функции. Например:

```
int func(int x[], int y)
   {
   /* тело функции */
   }
```

2) в прототипах функций, использующих в качестве аргумента указатель на первый элемент массива. Например:

```
int func(int x[], int y);
```

3. Доступ к отдельным элементам массива

Доступ к отдельному элементу массива осуществляется с помощью индекса.

Индекс описывает позицию элемента внутри массива. *Индекс* – целочисленное выражение, значение которого изменяется между 0 и (*Размер*-1). Для переменной – индекса лучше выбирать тип целый со знаком, так как контроль выхода значений индекса за допустимый диапазон не ведется.

Таким образом, первый элемент массива имеет нулевой индекс.

```
const  int n=4;
int   a[n] = {5, 6,7,8};
int i = 3;
a[1] = a[2]+a[i]-6;
```

Поскольку массив *a* содержит 4 элементов, его индексы изменяются от 0 до 3. Чтобы получить доступ к элементу массива по индексу, достаточно указать нужный номер элемента в квадратных скобках. Так, первым элементом массива *a* является *a*[0], а последним – *a*[*3*].

Пример 1. Составить программу, в которой поместить в массив *a* числа от 0 до 9. Вывести содержимое массива на экран.

Решение:

```
#include <conio.h>
#include <iostream.h>

int _tmain()
{
int a[10]; // Эта инструкция резервирует область
                  // памяти для 10 элементов типа int
int t;
// Помещаем в массив значения
for(t = 0;t<10;++t)
a[t] = t ; // Обратите внимание
                  // на то, как индексируется массив a
```

```
// Отображаем содержимое массива.
for(t=0;t<10;++t)
cout<<" a["<<t<<"] = "<<a[t]<<"\n";
getch();
}
```

Результаты выполнения этой программы выглядят так:

a[0] = 0
a[1] = 1
a[2] = 2
a[3] = 3
a[4] = 4
a[5] = 5
a[6] = 6
a[7] = 7
a[8] = 8
a[9] = 9

В C++ все массивы занимают смежные ячейки памяти. (Другими словами, элементы массива в памяти расположены последовательно друг за другом.) Ячейка с наименьшим адресом относится к первому элементу массива, а с наибольшим – к последнему. Например, после выполнения этого фрагмента кода

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

массив nums будет выглядеть следующим образом.

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]
0	1	2	3	4

Массивы часто используются в программировании, поскольку позволяют легко обрабатывать большое количество связанных переменных.

Пример 2. Составить программу, в которой создать массив из десяти элементов, каждому элементу присвоить некоторое число, а затем вычислить и отобразить среднее значение, а также минимальное и максимальное.

Решение:

```
int _tmain()
{
int i, av, mn, mx;
int nums[10];
nums[0]=12; //первый элемент массива
                    //инициализируется значением 12
nums[1]=10;
nums[2]=0;
nums[3]=-1;
nums[4]=5;
nums[5]=75;
nums[6]=3;
```

```
nums[7]=52;
nums[8]=-6;
nums[9]=80;
// вычисление среднего значения
av = 0 ;
for(i=0;i<10;i++)
av += nums[i]; // суммируем значения массива nums
av /= 10;       // вычисляем среднее значение
cout<<"Srednee znatshenie ravno"<< av << "\n";
// Определение минимального и максимального значений.
mn=mx=nums[0];
for(i=1; i<10; i++)
{
if(nums[i] < mn) mn=nums[i]; // минимум
if(nums[i] > mx) mx=nums[i]; // максимум
}
cout<< "Minimalnoe znathenie: " << mn << "\n";
cout<< "Maksimalnoe znathenie: "<< mx << "\n";
getch();
}
```

Результат выполнения программы:
Srednee znathenie ravno 23
Minimalnoe znathenie: -6
Maksimalnoe znathenie: 80

Ввод с клавиатуры элементов массива `nums[10]` можно выполнить так:

```
for(i=0;i<10;i++)
cin>>nums[i];
```

Обратите внимание на то, как в программе опрашиваются элементы массива ***nums***. Тот факт, что обрабатываемые здесь значения сохранены в массиве, значительно упрощает процесс определения среднего, минимального и максимального значений. Управляющая переменная цикла ***for*** используется в качестве индекса массива при доступе к очередному его элементу.

В C++ нельзя присвоить один массив другому, указав лишь их имена, также нельзя сравнивать массивы, складывать и т. д. Все операции разрешено выполнять только с отдельными элементами массива.

В следующем фрагменте кода, например присваивание $a = b$; недопустимо.

```
int a[10], b[10];
a = b; // Ошибка!!!
```

Чтобы поместить содержимое одного массива в другой, необходимо отдельно выполнить присваивание каждого значения. Например:

```
for(i=0; i<10; i++) a[i]=b[i];
```

4. Граница массива

В C++ не выполняется никакой проверки «нарушения границ» массивов, т. е. ничего не может помешать программисту обратиться к массиву за его пределами. Другими словами, обращение к массиву (размером N элементов) за границей N-го элемента может привести к разрушению программы при отсутствии каких-либо замечаний со стороны компилятора и без выдачи сообщений об ошибках во время работы программы. Например, C++-компилятор «молча» скомпилирует и

позволит запустить следующий код на выполнение, несмотря на то, что в нем происходит выход за границы массива crash.

```
int crash[10], i;
for(i=0; i<100; i++)
crash[i]=i;
```

В данном случае цикл for выполнит 100 итераций, несмотря на то, что массив crash предназначен для хранения лишь 10 элементов. При выполнении этой программы возможна перезапись важной информации, что может привести к аварийному останову программы.

Если «нарушение границы» массива происходит при выполнении инструкции присваивания, могут быть изменены значения в ячейках памяти, выделенных некоторым другим переменным. Если границы массива нарушаются при чтении данных, то неверно считанные данные могут попросту разрушить вашу программу.

В любом случае вся ответственность за соблюдение границ массивов лежит только на программистах, которые должны гарантировать корректную работу с массивами. Другими словами, программист обязан использовать массивы достаточно большого размера, чтобы в них можно было без осложнений помещать данные, но лучше всего в программе предусмотреть проверку пересечения границ массивов.

Пример 3. Несоблюдение границ массива. В программе сделана попытка выполнить обращение к массиву, указав в качестве индекса значение, превышающее его границу. Программа успешно прошла компиляцию, и на допущенную в программе ошибку не было указано.

Элементу присвоено значение соседней с массивом области памяти.

```
// Несоблюдение границ
const int N=10;
int v[N];
cout<<"\n v: ";
for (int i=0;i<N;i++) {

v[i] = i+1;
cout<<' '<<v[i];
}
// индекс за границей массива:
cout<<"\n v[20]: "<<v[N+10]<<"\n";
```

5. Заполнение массива

Пример 4. Создать два массива `a` и `b`. Массив `b` состоит из тех же данных, что и `a`, но записанных в обратном порядке.

```
const int N=10;
int a[N];
int b[N];
for (int i=0;i<N;i++)
a[i] = i+1;
for (int i=0;i<N;i++)
b[i] = a[N-i-1];
cout<<"\n a: ";
for (int i=0;i<N;i++)
cout<<' '<<a[i];
```

```
cout<<"\n b: ";
for (int i=0;i<N;i++)
cout<<' '<<b[i];
```

Выражение `N`, задающее количество элементов в объявлении массива, должно быть константой (как в примере), или выражением, значение которого известно во время компиляции.

Пример 5. Заполнение массива случайными числами.

```
const int N=10;
int d[N];
randomize();
for (int i=0; i<N;i++)
d[i] = random(10);
//можно d[i] = 1+rand()%10;
cout<<"\n d: ";
for (int i=0;i<N;i++)
cout<<' '<<d[i];
```

6.Объем памяти, занятой массивом

Объем памяти, занятой элементами массива, можно определить с помощью функции `sizeof()` по формуле

количество_байтов = sizeof(тип) × размер_массива

Пример 6. Вычисление объема памяти, занятой массивом.

```
const int N=10;
float x[N];
int size_x = sizeof(float) * N;
cout<<"\n Memory size for x ="<<size_x<<" bytes\n";
```

7.Сравнение двух массивов

Пример 7. Сравнение двух массивов на равенство элементов.

```
// Сравнение массивов
const int N=10;
int a[N]={1,2,3,4,5,6,7,8,9,10},
    b[N]={1,2,3,42,52,6,7,8,9,10};
bool p=0;
int k=0;
for (int i=0;i<N;i++)
{
  p = (b[i] != a[i]);
  k=k+p;
  }
if (k>0) cout<<"\n a!=b \n";
else cout<<"\n a=b \n";
```

Пример 8. Можно улучшить код и избежать лишних проверок. Если условие продолжения цикла заменить условием (i<N) && p, выход из цикла будет выполнен при первом же несовпадении элементов массивов.

```
// Сравнение массивов
const int N=10;
int a[N], b[N];
for (int i=0;i<N;i++) {
a[i] = i*2; b[i] = i*3;
}
// проверяем, равны ли массивы a и b
bool p=1;
for (int i=0;(i<N) && p;i++) p = (b[i] == a[i]);
cout<<"\n a: ";
for (int i=0;i<N;i++) cout<<' '<<a[i];
cout<<"\n b: ";
for (int i=0;i<N;i++) cout<<' '<<b[i];
cout<<"\n";
if (p) cout<<"\n a=b \n";
else cout<<"\n a!=b \n";
```

Сложнее проверить содержат ли массивы одни и те же данные, т.е. массивы совпадут после перестановки элементов в одном из них.

Пример 9. Совпадение данных.

```
const int N=10;
int a[N], b[N];
for (int i=0;i<N;i++) {
a[i] = i+1; b[i] = N-i;
}
// проверяем, все ли элементы массива a есть в b
bool q=1; // проверочный флаг
for (int i=0;(i<N) && q;i++) {
q = 0;
for (int j=0;j<N;j++)
if (a[i]==b[j]) { // найдено совпадение
q=1;
break; // к следующему индексу i
}
}
if (q) cout<<"\n |a| = |b| \n";
else cout<<"\n |a| != |b| \n";
cout<<"\n a: ";
for (int i=0;i<N;i++) cout<<' '<<a[i];
cout<<"\n b: ";
for (int i=0;i<N;i++) cout<<' '<<b[i];
```

Пример 10. Тот же пример, но в качестве данных берутся наборы случайных чисел из одного диапазона. В коде изменен только блок инициализации массивов.

```
...
// Совпадение случайных данных
```

```
const int N=100;
int a[N], b[N];

for (int i=0;i<N;i++) a[i] = 1+rand()%100;
for (int i=0;i<N;i++) b[i] = 1+rand()%100;
. . .
```

Распространенной операцией с массивами является слияние двух и более массивов в один результирующий массив.

Пример 11. Слияние двух массивов.

```
// Слияние двух массивов
const int N=10;
const int M=12;
int a[N], b[M], c[N+M];
randomize();
// инициализация массивов
for (int i=0;i<N;i++) a[i] = random(100);
for (int i=0;i<M;i++) b[i] = random(100);

// образуем массив присоединением массива b к a
int k=0;
for (int i=0;i<N;i++) c[k++] = a[i];
for (int i=0;i<M;i++) c[k++] = b[i];
// вывод результатов
cout<<"\n a: ";
for (int i=0;i<N;i++) cout<<' '<<a[i];
cout<<"\n b: ";
for (int i=0;i<M;i++) cout<<' '<<b[i];
cout<<"\n c: ";
for (int i=0;i<N+M;i++) cout<<' '<<c[i];
cout<<"\n";
```

Пример 12. Слияние двух упорядоченных массивов
a[0]≤a[1]≤…≤a[N] и b[0]≤b[1]≤…≤b[N].

```
// Слияние двух упорядоченных массивов
const int N=9;
const int M=11;
int a[N], b[M], c[N+M];
// инициализация массивов
for (int i=0;i<N;i++) a[i] = 2*i+1;
for (int i=0;i<M;i++) b[i] = 2*(i+1);
int i,j,k;
i=j=k=0;
while ((i<N) && (j<M))
if (a[i]<b[j]) c[k++] = a[i++];
else c[k++] = b[j++];
// остаток массива a:
while (i<N) c[k++] = a[i++];
// остаток массива b:
```

```
while (j<M) c[k++] = b[j++];
```

8.Реализация метода Горнера

Инициализацию в объявлении массива удобно использовать при реализации численных методов.

Пример 13. Вычисление значения многочлена
$u(x) = x^4 - 3x^3 + 6x^2 - 10x + 16$ при $x = 4$ методом Горнера.

Многочлен от одной переменной x имеет вид: $P_n(x)=a_n \cdot x^n+a_{n-1} \cdot x^{n-1}+...+a_1 \cdot x+a_0$.
Схема Горнера:
$y=(...((a_n \cdot x+a_{n-1}) \cdot x+a_{n-2}) \cdot x+...+a_1) \cdot x+a_0$

Например: $P(x)=3 \cdot x^4+5 \cdot x^3-2 \cdot x+6 = (((3 \cdot x+5) \cdot x+0) \cdot x-2) \cdot x+6$

```
const int n=4;
double a[n+1]={16.0,-10.0,6.0,-3.0,1.0};
double u,x;
int k;
// Вычисление значения многочлена
for (x = 4,u = a[n],k=n-1;k>=0;k--)
u = u*x + a[k];
cout<<"\n u("<<x<<")= "<<u<<"\n";
```

Пример 14. Вычисление суммы элементов безразмерного числового массива. Количество элементов массива найдено с помощью функции `sizeof()`.

```
// безразмерные числовые массивы
double x[]={1.0,2,3.0,4.,5e1,6.6,0.7,80};
int k_x = sizeof(x)/sizeof(double);
/* количество элементов массива x */
cout<<"\n Size x = "<<k_x;
double s_x=0;
for (int i=0;i<k_x;i++)
s_x += x[i];
cout<<"\nSumma x = "<<s_x<<"\n";
```

9.Действия с n-мерными векторами

Пример 15. Составить программу, выполняющую ввод, сложение, вычитание, скалярное умножение двух n-мерных векторов и вывод результатов

Структура данных для представления n-мерного вектора: одномерный массив размера n типа double

$\vec{a} = (a_1, a_2, ..., a_n); \quad \vec{b} = (b_1, b_2, ..., b_n).$

$\vec{a} + \vec{b} = (a_1 + b_1, a_2 + b_2, ..., a_n + b_n);$

$\vec{a} - \vec{b} = (a_1 - b_1, a_2 - b_2, ..., a_n - b_n);$

$\vec{a}\vec{b} = a_1 b_1 + a_2 b_2 + ... + a_n b_n.$

Если список инициализации содержит меньше значений, чем число элементов массива, оставшиеся элементы будут инициализированы 0 .

```
const int n=4;
double a[n]={0}, b[n]={0}, res[n]={0};
cout << "Enter..." << endl;
for (int i=1; i<=n; i++)
  {
  cout << endl << "a" << i << ": ";
  cin >> a[i-1];
  cout << "b" << i << ": ";
  cin >> b[i-1];
  }
//Sum
for (int i=1; i<=n; i++)
  res[i-1]=a[i-1]+b[i-1];
  cout << endl << "a + b = ( ";
for (int i=1; i<=n; i++)
  cout << res[i-1] << " ";
  cout << ")";

//Difference
for (int i=1; i<=n; i++)
  res[i-1]=a[i-1]-b[i-1];
  cout << endl << "a - b = ( ";
for (int i=1; i<=n; i++)
  cout << res[i-1] << " ";
    cout << ")";

//Product
double r=0;
for (int i=1; i<=n; i++)
  r+=a[i-1]*b[i-1];
  cout << endl << "a * b = " << r;
```

Пример 16. Нахождение количества повторений каждой цифры числа

```
int a[10];
int n,i;
cout << "Input ";
cin >> n;
for (i=0; i<=9; i++)
   a[i]=0;
```

```
while (n!=0)
   {
   i = n % 10;
   a[i] = a[i]+1;
   n = n / 10;
   }
for (i=0; i<10; i++)
   if (a[i]!=0)
   cout<<i<<"-"<<a[i]<<endl;
```

Пример 17. **(Решето Эратосфена).**

Найти простые числа, меньшие наперед заданного n

Идея алгоритма Эратосфена: из списка всех чисел от 2 до n последовательно вычеркивать числа, кратные уже известным простым числам.

Основная структура данных программы: одномерный массив размера 2..n типа boolean. Индексы массива соответствуют анализируемым числам. Элемент массива имеет значение true, если индекс этого элемента - простое число.

Алгоритм на псевдокоде

Данные.

Максимальное анализируемое число n

Массив, хранящий информацию, простым ли является значение индекса каждого элемента: bool erato [n]

Алгоритм.

1. Инициализировать константу n
2. Объявить массив erato и инициализировать его значениями true
3. Для всех i от 2 до n

3.1. Если erato [i] == true ,

3.1.1. Заменить на false значения элементов массива erato с индексами от i до n, кратными i

4. Вывести значения индексов элементов массива erato , принимающих значение true

Оптимизируем и уточняем алгоритм на псевдокоде

Данные.

Максимальное анализируемое число n

Массив, хранящий информацию, простым ли является значение индекса каждого элемента: bool erato [n]

Алгоритм.

1. Инициализировать константу n
2. Объявить массив erato и инициализировать его значениями true
3. Для всех i от 2 до целой части sqrt (n)

3.1. Если erato [i] == true ,

3.1.1. Заменить на false значения элементов массива erato с индексами от i до n, равными i * j, где j изменяется от i до n

4. Вывести значения индексов элементов массива erato , принимающих значение true

```
const unsigned int n = 1000;
const unsigned int size = n+1;
bool erato [size]={0};
```

```
for (int i = 2; i <= n; i++)
erato [i] = 1;

for (int i = 2; i <= (int) (sqrt((float) (n))); i++)
   {
   if (erato[i])
   for (int j = i; j <= n; j++)
   {
   if (i*j > n)break;
   erato [i*j]=false;
   }
   }
   for(int i=2; i <= n; i++)
   if (erato[i]) cout << i << endl;
```

МНОГОМЕРНЫЕ МАССИВЫ

1. Объявление массива
2. Доступ к отдельным элементам массива
3. Инициализация элементов массива
4. Объем памяти, занятой массивом
5. Операции с матрицами

В лекции рассматриваются особенности использования массивов при составлении программ на языке C++.

Массивы позволяют удобным образом организовать размещение и обработку больших объемов информации.

В C++ можно использовать *многомерные* массивы.

1. Объявление массива

При объявлении многомерных массивов задается тип элементов массива, имя массива и затем, в отдельных квадратных скобках – количества элементов по каждой размерности:

тип имя_массива [***Размер1***][***Размер2***]. . . [***РазмерN***];

Простейший многомерный массив – двумерный. Двумерный массив, по сути, представляет собой список одномерных массивов. Чтобы объявить двумерный массив целочисленных значений размером 10×20 с именем **D**, достаточно записать следующее:

```
int D[10][20];
```

В двумерном массиве позиция любого элемента определяется двумя индексами. Если представить двумерный массив в виде таблицы данных, то один индекс означает строку, а второй – столбец. Из этого следует, что, если доступ к элементам массива предоставить в порядке, в котором они реально хранятся в памяти, то правый индекс будет изменяться быстрее, чем левый.

Пример

```
const int K=4;
const int N=3;
const int M=5;
int one[N];
int two[N][M];
int three[K][N][M];

for (int i=0;i<K;i++)
  one[i]=i*10;

for (int p=0;p<N;p++)
for (int q=0;q<M;q++)
  two[p][q]=p*q;

for (int i=0;i<K;i++)
for (int p=0;p<N;p++)
```

```
for (int q=0;q<M;q++)
  three[i][p][q]=one[i]+ two[p][q];
```

2. Доступ к отдельным элементам массива

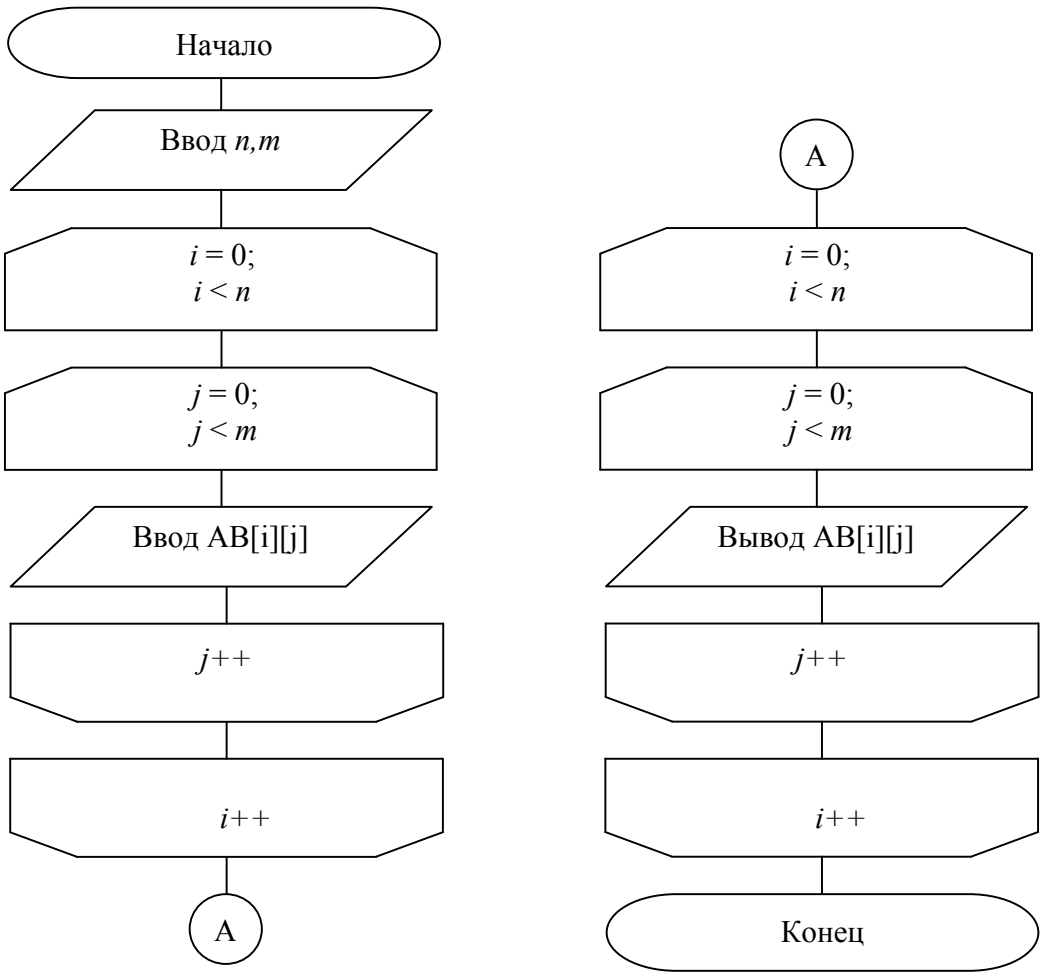
В C++ каждая размерность заключается в собственную пару квадратных скобок. Так, чтобы получить доступ к элементу массива **D** с координатами 3,5(четвертая строка, шестой столбец), необходимо использовать запись D [3][5].

Ввод-вывод элементов двумерного массива

```
const int nm=20; //макс размер матрицы
  int AB [nm][nm],m;
cin >> n >> m;
cout << "Элементы матрицы?\n";
for (int i=0;i<n;i++)
      for (int j=0;j<m;j++)
         cin >> AB[i][j];

    //вывод матрицы
cout << "Введенная матрица" <<endl;
for (int i=0;i<n;i++)
{
      for (int j=0;j<m;j++)
         cout << AB[i][j] << " ";
         cout << endl;
}
```

блок-схема



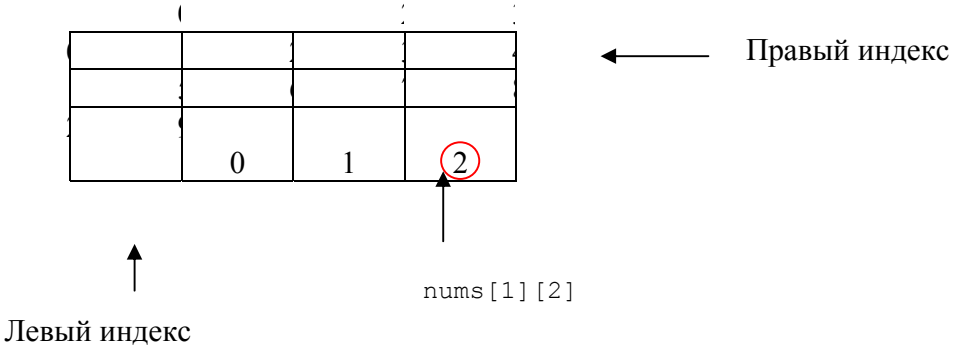
Пример 2. Составить программу, в которой в двумерный массив *nums*[3][4] поместить последовательные числа от 1 до 12. Вывести содержимое массива на экран.

Решение:

```
int _tmain()
{
int t, i, nums[3][4];
for(t=0; t < 3; ++t)
{
for(i=0; i < 4; ++i)
{
nums[t][i] = (t*4)+i+1;          // Для индексации массива
                                 // nums требуется два индекса.
cout<<nums[t][i]<<"\t";
}
cout<<"\n";
}
```

```
getch();
}
```

В этом примере элемент *nums*[0][0] получит значение 1, элемент *nums*[0][1] – значение 2, элемент *nums*[0][2] – значение 3 и т. д. Значение элемента *nums*[2][3] будет равно числу 12. Схематически этот массив можно представить следующим образом:



3. Инициализация элементов массива

Многомерные массивы инициализируются по аналогии с одномерными. Например, в следующем фрагменте программы массив *sq* инициализируется числами от 1 до 10 и квадратами этих чисел.

```
int sq[10][2] = { 1, 1,
                  2, 4,
                  3, 9,
                  4, 16,
                  5, 25,
                  6, 36,
                  7, 49,
                  8, 64,
                  9, 81,
                  10,100};
```

Пример. Инициализация двумерного массива (матрицы). Значения можно записать подряд или же разбить на группы, выделив каждую строку матрицы, фигурными скобками (этот способ называют ***subaggregate grouping*** , т.е. ***группирование подагрегатов***).

Многомерные массивы могут быть безразмерными – можно не указывать размер самого левого измерения, если при объявлении используется инициализатор.

```
// Магический квадрат
const int N=4;
int mag1[N][N]={16,3,2,13,5,10,11,8,
                9,6,7,12,4,15,14,1};
int mag2[N][N] ={
```

```
                    {16,3,2,13},
                    {5,10,11,8},
                    {9,6,7,12},
                    {4,15,14,1}
                };

for (int i=0;i<N;i++)
{
for (int j=0;j<N;j++)
cout<<mag1[i][j]<<"\t";
cout<<"\n"; // новая строка матрицы
}

// How to initialize two-dimensional arrays
const int m=4, n =6;           // maximum array dimensions
   int a[m][n] = {{0,1,2,3,4,5},
                  {10,11,12,13,14,15},
                  {20,21,22,23,24,25},
                  {30,31,32,33,34,35}};
// third & fourth rows are set to zero
int b[m][n] = {0,1,2,3,4,5,10,11,12,13,14,15};
int d[m][n] = {0};   // the whole array is set to zero
// first dimension's size will be calculated by compiler
int c[][n]={
                {0 , 1, 2, 3, 4, 5},
                {10,11,12,13,14,15},
                {20,21,22,23,24,25},
                {30,31,32,33,34,35}
           };
```

4. Объем памяти, занятой массивом

Необходимо помнить, что место хранения для всех элементов массива определяется во время компиляции. Кроме того, память, выделенная для хранения массива, используется в течение всего времени существования массива. Для определения количества байтов памяти, занимаемой двумерным массивом, используйте следующую формулу:

число_байтов = число_строк×число_столбцов×размер_типа_в_байтах

Следовательно, двумерный целочисленный массив размерностью 10×5 занимает в памяти 10×5×4, т. е. 200 байт (если целочисленный тип имеет размер 4 байт).

В C++, помимо двумерных, можно определять массивы трех и более измерений. Вот как объявляется ***многомерный*** массив.

тип имя[размер1] [размер2]... [размеры] ;

Например, с помощью следующего объявления создается трехмерный целочисленный массив размером 4×10×3.

```
int  multidim[4][10][3] ;
```

Массивы с числом измерений, превышающим три, используются нечасто, хотя бы потому, что для их хранения требуется большой объем памяти. Ведь, как упоминалось выше, память, выделенная для хранения всех элементов массива, используется в течение всего времени его существования. Например, хранение элементов четырехмерного символьного массива размером 10×6×9×4 займет 2 160 байт. А если каждую размерность увеличить в 10 раз, то занимаемая массивом память возрастет до 21 600 000 байт. Как видно, большие многомерные массивы способны «съесть» большой объем памяти, а программа, которая их использует, может очень быстро столкнуться с проблемой нехватки памяти.

Пример. Безразмерный двумерный массив.


```
// Магический квадрат Дюрера
const int M=4; /* граница 2-го измерения массива*/
int mag[][M] ={
{16,3,2,13},
{5,10,11,8},
{9,6,7,12},
{4,15,14,1}
};
// Вычисляем границу первого измерения:
int n =          /* граница 1-го измерения массива*/
sizeof(mag)      /* память под весь массив*/
/(sizeof(int)*M) /* память, занятая строкой массива*/;
for (int i=0;i<n;i++){
for (int j=0;j<M;j++) cout<<mag[i][j]<<"\t";
cout<<"\n"; // новая строка матрицы
```

```
}
```

5. Операции с матрицами

1. Сложение матриц - поэлементная операция

$$A_{mn} + B_{mn} = C_{mn} = \begin{vmatrix} a_{11}+b_{11} & a_{12}+b_{12} \dots a_{1n}+b_{1n} \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} \dots a_{mn}+b_{mn} \end{vmatrix}$$

```
for (int i=0;i<n;i++)
{
for (int j=0;j<m;j++)
{
c[i][j]= a[i][j]+ b[i][j];
cout<<c[i][j]<<"\t";
}
cout<< "\n";
}
```

2. Вычитание матриц - поэлементная операция

$$A_{mn} - B_{mn} = \begin{vmatrix} a_{11}-b_{11} & a_{12}-b_{12} \dots a_{1n}-b_{1n} \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ a_{m1}-b_{m1} & a_{m2}-b_{m2} \dots a_{mn}-b_{mn} \end{vmatrix}$$

```
...
c[i][j]= a[i][j]- b[i][j];

...
```

3. Произведение матрицы на число - поэлементная операция

$$\lambda A = \begin{pmatrix} \lambda a_{11} & \lambda a_{12} & \dots & \lambda a_{1n} \\ \lambda a_{21} & \lambda a_{22} & \dots & \lambda a_{2n} \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ \lambda a_{m1} & \lambda a_{m2} & \dots & \lambda a_{mn} \end{pmatrix}$$

```
...
c[i][j]= n*a[i][j];
```

. . .

4. Умножение **A*B** матриц по правилу **строка на столбец** (число столбцов матрицы A должно быть равно числу строк матрицы B)

$\mathbf{A_{mk}*B_{kn}=C_{mn}}$ причем каждый элемент $\mathbf{c_{ij}}$ матрицы $\mathbf{C_{mn}}$ равен сумме произведений элементов i-ой строки матрицы A на соответствующие элементы j-го столбца матрицы B , т.е.

$$C_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj} = \sum_{l=1}^{n} a_{il}b_{lj}$$

Например,

$$C_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + \ldots + a_{1n}b_{n1}$$

$$C_{12} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + \ldots + a_{1n}b_{n2}$$

```
int sum;
// перемножение матриц
for (i = 0; i < n; ++i)
for (j = 0; j < n; ++j)
{
  sum = 0;
  for (k = 0; k < n; ++k)
  sum += a[i][k] * b[k][j];
  c[i][j] = sum;
}
```

5. Транспонирование матрицы A. Транспонированную матрицу обозначают A^T или A'

$$Если\ A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{22} & a_{22} & \ldots & a_{2n} \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix},\ то$$

$$A^T = A' = \begin{pmatrix} a_{11} & a_{21} & \ldots & a_{m1} \\ a_{12} & a_{22} & \ldots & a_{m2} \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ a_{1n} & a_{2n} & \ldots & a_{mn} \end{pmatrix}$$

Строки и столбцы поменялись местами

Пример

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix},$$

$$A^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

```
...
c[i][j]= a[j][i];
...
```

Пример. Элементы матрицы вводятся с клавиатуры. Вычисляется след матрицы (сумма диагональных элементов).

```
// След матрицы
const int N=3;
double a[N][N]; int i,j;
// Блок ввода значений
for (i=0;i<N;i++)
for (j=0;j<N;j++) {
cout<<"a["<<i<<"]["<<j<<"]="; cin>>a[i][j];
}
double tr_a=0.0;
for (i=0;i<N;i++) tr_a+=a[i][i];
cout<<"\n Tr (a) = "<<tr_a<<"\n";
```

Пример. Вычисление произведения сумм элементов строк матрицы:

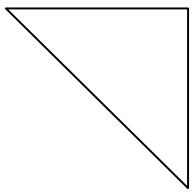
$$P = \prod_{i=1}^{N} \sum_{j=1}^{M} a_{ij}$$

Матрица заполняется случайными числами.

```
// P= \Pi_{i=1}^n \Sigma_{j=1}^m a_{ij}
```

```
double a[5][6]; int i,j;
randomize();
for (i=0;i<5;i++)
for (j=0;j<6;j++)
a[i][j] = (rand()%100) * 0.1;
double p,s;
p=1.0;// произведение
for (i=0;i<5;i++)
{
s = 0; // s - сумма элементов строки
for (j=0; j<6;j++)
s += a[i][j];
p *= s;
}
// Вывод результатов
for (i=0;i<5;i++){
for (j=0;j<6;j++)
cout<<a[i][j]<<"\t";
cout<<"\n"; // новая строка матрицы
}
cout<<"\n p = "<<p<<"\n";
```

Пример.



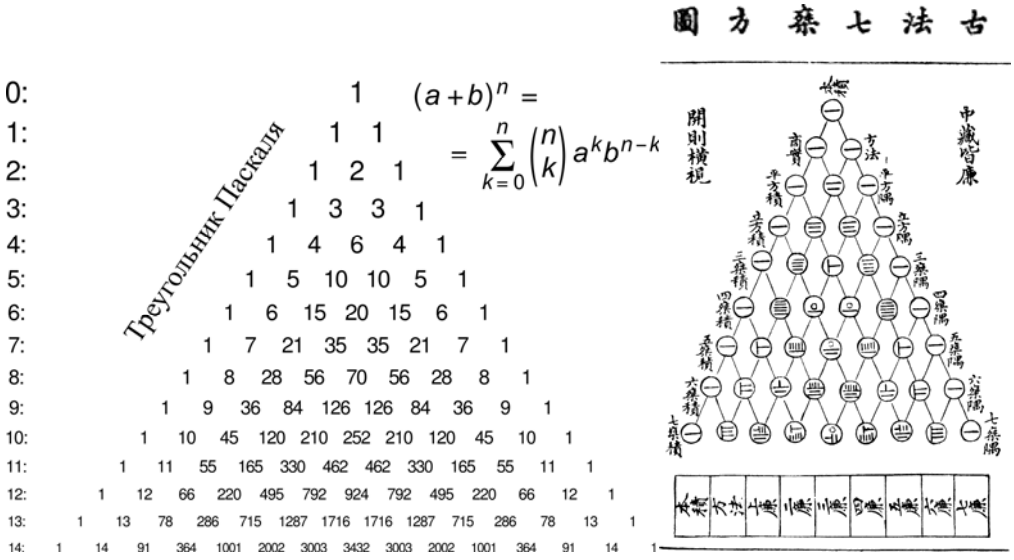
```
/*сумма элементов верхнего правого треугольника матрицы*/
/*максимальный размеры матрицы*/
const nmax = 10;
float a[nmax][nmax];
int n,i,j;
```

```
 cout << "введите размерность<10\n";
 cin >> n;
 cout << "введите матрицу по строкам\n";
for (int i=0; i<n; i++)
for (int j=0; j<n; j++)
cin >> a[i][j];

float sum=0;
for (int i=0; i<n; i++)
for (int j=i; j<n; j++)
sum = sum + a[i][j];
cout << "sum= " << sum;
```

Треугольник Паскаля — бесконечная таблица биномиальных коэффициентов, имеющая треугольную форму. В этом треугольнике на вершине и по бокам стоят единицы. Каждое число равно сумме двух расположенных над ним чисел. Строки треугольника симметричны относительно вертикальной оси. Назван в честь Блеза Паскаля. Числа, составляющие треугольник Паскаля, возникают естественным образом в алгебре, комбинаторике, теории вероятностей, математическом анализе, теории чисел.

Первое упоминание треугольной последовательности биномиальных коэффициентов под названием *meru-prastaara* встречается в комментарии индийского математика X века Халаюдхи[hi] к трудам другого математика, Пингалы. Треугольник исследуется также Омаром Хайямом около 1100 года, поэтому в Иране эту схему называют треугольником Хайяма. В 1303 году была выпущена книга «Яшмовое зеркало четырёх элементов» китайского математика Чжу Шицзе, в которой был изображен треугольник Паскаля на одной из иллюстраций; считается, что изобрёл его другой китайский математик, Ян Хуэй (поэтому китайцы называют его треугольником Яна Хуэя). На титульном листе учебника арифметики, написанном в 1529 году Петром Апианом, астрономом из Ингольштадтского университета, также изображён треугольник Паскаля. А в 1653 году (в других источниках в 1655 году или в 1665 году) вышла книга Блеза Паскаля «Трактат об арифметическом треугольнике».



Биномиальные коэффициенты часто обозначаются $\binom{n}{k}$ или C_n^k и читаются как «число сочетаний из n элементов по k».

- Числа треугольника симметричны (равны) относительно вертикальной оси.
- В строке с номером n:
  - первое и последнее числа равны 1.
  - второе и предпоследнее числа равны n.
  - третье число равно треугольному числу $T_{n-1} = \frac{n(n-1)}{2}$, что также равно сумме номеров предшествующих строк.
  - четвёртое число является тетраэдрическим.
  - m-е число (при нумерации с 0) равно биномиальному коэффициенту $C_n^m = \binom{n}{m} = \frac{n!}{m!(n-m)!}$.
- Сумма чисел восходящей диагонали, начинающейся с первого элемента (n-1)-й строки, есть n-е число Фибоначчи:

$$\binom{n-1}{0} + \binom{n-2}{1} + \binom{n-3}{2} + \ldots = F_n.$$

- Если вычесть из центрального числа в строке с чётным номером соседнее число из той же строки, то получится число Каталана (числовая последовательность, встречающаяся во многих задачах комбинаторики).
- Сумма чисел n-й строки треугольника Паскаля равна 2^n.
- Все числа в n-й строке, кроме единиц, делятся на число n, тогда и только тогда, когда n является простым числом[4] (следствие теоремы Люка).
- Если в строке с нечётным номером сложить все числа с порядковыми номерами вида 3n, 3n+1, 3n+2, то первые две суммы будут равны, а третья на 1 меньше.
- Каждое число в треугольнике равно количеству способов добраться до него из вершины, перемещаясь либо вправо-вниз, либо влево-вниз.

треугольник Паскаля

Всем известны простые формулы

- $(a+b)^2 = a^2 + 2ab + b^2$
- $(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$

А как находить коэффициенты в разложении $(a+b)^n$?

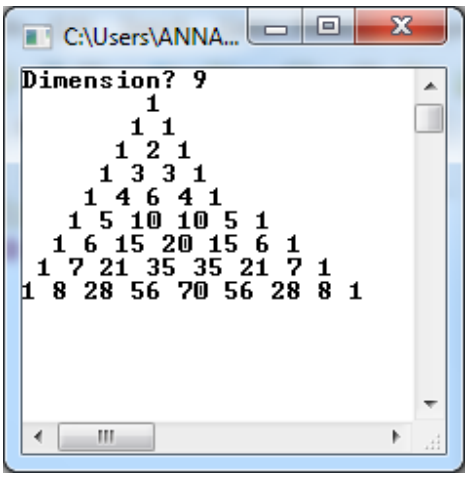
$(a+b)^0$	=	1	0								1											
$(a+b)^1$	=	$a+b$	1							1			1									
$(a+b)^2$	=	$a^2+2ab+b^2$	2						1		2			1								
$(a+b)^3$	=	$a^3+3a^2b+3ab^2+b^3$	3					1		3			3			1						
$(a+b)^4$	=	$a^4+4a^3b+6a^2b^2+4ab^3+b^4$	4				1		4			6			4			1				
$(a+b)^5$	=	$\ldots$	5			1		5			10			10			5			1		
$(a+b)^6$	=	$\ldots$	6	1			6			15			20			15			6			1

Построение треугольника Паскаля (вид 1)

```
1
1     1
1     2    1
1     3    3     1
1     4    6     4     1
1     5   10    10     5    1
1     6   15    20    15    6    1
1     7   21    35    35   21    7    1
1     8   28    56    70   56   28    8    1
```

```
#include <iomanip.h>
// Установить ширину поля, где n — количество позиций, символов
// cout<<setw(n);
. . .
//треугольник Паскаля
 const int nmax=10;
 int n,i,j;
 int Ma[nmax][nmax];
 cout<<"Dimension? ";
cin>>n;
//расчёт
Ma[0][0]=1;
for (i=1; i<n; i++)
      {
       Ma[i][0]=1;
       Ma[i][i]=1;
for (j=1; j<i; j++)
       Ma[i][j]=Ma[i-1][j-1]+Ma[i-1][j];
       }
//вывод на экран
 cout<<"\t"<<setw(n);
```

```
 for (i=0; i<n; i++)
      {
      cout<<setw(n-j);
 for (j=0; j<i+1; j++)
      cout<<Ma[i][j]<<" ";
      cout<<"\n";
      }
```



Представить целочисленную квадратную матрицу 4x4 в виде массива. Присвоить элементам на главной диагонали значение 1, выше главной диагонали - 2, ниже – 0.

```
const int n=4;
int a[n][n] = {0};
for (int i=0; i<=n-1; i++)
for (int j=0; j<=n-1; j++)
    {
      if (i == j) a[i][j]=1;
      else if (i < j) a[i][j]=2;
    }
for (int i=0; i<=n-1; i++)
    {
      cout << endl;
for (int j=0; j<=n-1; j++)
     cout << a[i][j];
    }
```

УКАЗАТЕЛИ, ДИНАМИЧЕСКОЕ РАСПРЕДЕЛЕНИЕ ПАМЯТИ

1. Указатели
2. Операторы, используемые с указателями
3. Свободная память и указатели
4. Арифметические операции с указателями
5. Присваивание указателей
6. Указатели и массивы
7. Динамическое распределение памяти

В лекции рассматриваются особенности использования массивов и указателей при составлении программ на языке С++.

Массивы позволяют удобным образом организовать размещение и обработку больших объемов информации.

Указатели − один из самых важных и сложных аспектов С++. Вместе с тем они зачастую являются источниками потенциальных проблем. В значительной степени мощь многих средств С++ определяется использованием указателей. Например, благодаря им обеспечивается поддержка связных списков и механизма динамического выделения памяти, и именно они позволяют функциям изменять содержимое своих аргументов.

1. Указатели

Каждый байт в памяти ЭВМ имеет адрес, по которому можно обратиться к определенному элементу данных. Адресом (*address*) называют число, идентифицирующее ячейку в памяти.

Указатель – это объект, который содержит некоторый адрес памяти. Чаще всего этот адрес обозначает местоположение в памяти другого объекта, такого как переменная. Например, если *x* содержит адрес переменной *y*, то о переменной *x* говорят, что она «указывает» на *y*.

Значение указателя является целым числом, но сам указатель целым числом не является. Тип указателя позволяет выполнять операции над адресами, в то время как тип **int** позволяет выполнять (арифметические и логические) операции над целыми числами.

Значения переменных указателей зависят от того, как компилятор размещает переменные, на которые они указывают, в памяти, т.е. от его реализации. Обозначение, используемое для значения указателя (адрес), также может изменяться в зависимости от того, какие соглашения приняты в системе; для обозначения значений указателей часто используются шестнадцатеричные числа. При выводе значения адреса на экран он автоматически отображается в формате адресации, применяемом для текущего процессора и среды выполнения.

Переменные-указатели (или *переменные типа указатель)* должны быть соответственно объявлены. Формат объявления переменной-указателя таков:

Базовый_тип *имя_указателя;

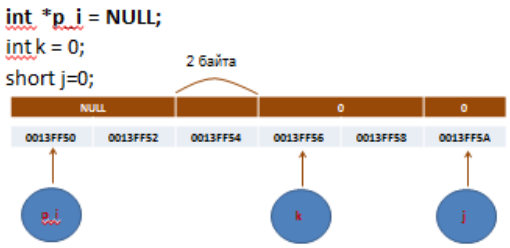
Базовый тип определяет, на какой тип данных будет ссылаться объявляемый указатель, причем он должен быть допустимым С++-типом. Элемент *имя_ указателя* представляет собой имя переменной-указателя. Например, чтобы объявить переменную ***p_i*** указателем на ***int***-значение, используйте следующую инструкцию:

```
int *p_i;
```

Для объявления указателя на float-значение используйте такую инструкцию.

```
float *fp;
```

В общем случае использование символа «звездочка» (*) перед именем переменной в инструкции объявления превращает эту переменную в указатель.



Указатель содержит **адрес,** начиная с которого размещается переменная, на которую ссылается указатель.

По описанию указателя компилятор получает информацию о том, какова длина области памяти, на которую ссылается указатель (которую занимает переменная, на которую он ссылается) и о том, как интерпретировать данные в этой области памяти.

```
int *ip, *iq;   // указатели на целые объекты
float *fp;          // указатель на символьный объект
char *cp;     // указатель на символьный объект
char *const cp;          // константный указатель на char
char const* pc;          //указатель на константу типа char
int x;
```

```
char*p;
char s[] = "Ci++";
const char* pc = s;    //указатель на константу

pc[3] = 'g';
//  ошибка: pc указывает на константу

pc = p;                  // правильно
```

Особенности синтаксиса объявления указателей

- Первый способ записи:
  Базовый_тип *Имя_Указателя;
  Нет проблем с объявлением нескольких указателей:
  int *pa, *pb, *pc;

- Второй способ записи:
  Базовый_тип* Имя_Указателя;
  Позволяет идентифицировать тип «указатель на базовый тип» как *Базовый_тип** . Однако , эта форма записи вызывает проблемы при множественном объявлении переменных:
  int* pa, ib, ic;
  Во избежание путаницы, если Вы придерживаетесь второго способа, никогда не декларируйте в одном инструкции более одной (указательной) переменной.
- В качестве базового типа можно указать void: void *p;
- Указатель на void не может быть разыменован

2. Операторы, используемые с указателями

С указателями используются два оператора: "*" и "**&**".

Оператор "**&**" (получение адреса) – унарный. Он возвращает адрес памяти, по которому расположен его операнд. (Вспомните: унарному оператору требуется только один операнд.) Например, при выполнении следующего фрагмента кода

```
p_i = &k;
```

в переменную **p_i** помещается адрес переменной ***k***. Этот адрес соответствует области во внутренней памяти компьютера, которая принадлежит переменной ***k***. Выполнение этой инструкции никак не повлияло на значение переменной ***k***. Назначение оператора "**&**" можно пояснить как «адрес переменной», перед которой он стоит. Следовательно, приведенную выше инструкцию присваивания можно выразить так: «переменная **p_i** получает адрес переменной ***k***». Чтобы лучше понять суть этого присваивания, предположим, что переменная ***k*** расположена в области памяти с адресом **0013FF56**. Следовательно, после выполнения этой инструкции переменная **p_i** получит значение **0013FF56**.

Переменные-указатели должны всегда указывать на соответствующий тип данных, т.е. тип указателя должен быть как у переменной, на которую он указывает. Например, при объявлении указателя типа **int** компилятор "предполагает", что все значения, на которые ссылается этот указатель, имеют тип **int**. C++-компилятор попросту не позволит выполнить операцию присваивания с участием указателей (с обеих сторон от оператора присваивания), если типы этих указателей несовместимы (по сути, не одинаковы). Например, следующий фрагмент кода некорректен.

int *p_i = NULL;
int k = 0;
short j=0;
нельзя p_i = &j;
(p_i – указатель на int, а j объявлена как short)

Операция приведения типов решит проблему формально, но может привести к сомнительным результатам, поскольку именно базовый тип указателя определяет, как компилятор будет обращаться с данными, на которые он ссылается.

Операция взятия адреса: как размещаются в памяти переменные и элементы массивов

```
short c1 = 1, c2 = 2, c3 = 3;
cout << &c1 << endl;
cout << &c2 << endl;
cout << &c3 << endl;
cout << endl;
```

2 байта между адресами переменных, неупорядочены

```
12ff52
12ff50
12ff4e
```

```
short c4[3] = {1,2,3};
cout << &c4[0] << endl;
cout << &c4[1] << endl;
cout << &c4[2] << endl;
cout << endl;
```

2 байта между адресами элементов массива, строго упорядочены

```
12ff28
12ff2a
12ff2c
```

```
int b=5, b1=1;
cout << &b << endl;
cout << &b1 << endl;
```

4 байта между адресами переменных, неупорядочены

```
12ff50
12ff4c
```

Второй оператор работы с указателями * (разадресация) служит дополнением к первому (**&**). Это унарный оператор, который обращается к значению переменной, расположенной по адресу, заданному его операндом. Другими словами, он возвращает значение переменной, хранящееся по заданному адресу, то есть выполняет действие, обратное операции &. Если (продолжая работу с предыдущей инструкцией присваивания) переменная **p_i** содержит адрес переменной ***k*** , то при выполнении инструкции

```
*p_i=123;
```

переменной ***k*** будет присвоено значение 123. Назначение оператора "*" можно выразить словосочетанием «по адресу».

В данном случае предыдущую инструкцию можно прочитать так: «переменная, расположенная по адресу ***p_i*** получает значение 123».

Запись

j = *p_i; // j = k

означает, что переменная j получает значение, хранящееся по адресу **p_i .**

Операторы "*" и "*&*" имеют более высокий приоритет, чем любой из арифметических операторов, за исключением унарного минуса, приоритет **которого** такой же, как у операторов, применяемых для работы с указателями.

Операции, выполняемые с помощью указателей, часто называют *операции непрямого доступа,* поскольку они позволяют получить доступ к одной переменной посредством некоторой другой переменной.

Пример. Объявление указателей и обращение к памяти. Значение переменной `px` – адрес памяти, а `*px` – данные, записанные по адресу `px`.

```
int x=100;
     cout<<"\n x="<<x;
     int *px;
// указателю присвоили адрес переменной x
     px=&x;
```

```
// Изменения значения, записанного по адресу px
     *px=200;
     cout<<"\n px="<<px;
cout<<"\n *px="<<*px<<"\n x="<<x<<"\n";
(*px)++;
cout<<"\n *px="<<*px<<"\n x="<<x<<"\n";
```

```
x=100
px=12ff50
*px=200
x=200

*px=201
x=201
```

Пример. Составить программу, в которой продемонстрировать использование операций адресации и разадресации.

```
int total;
int *ptr;
int val;
total = 3200; // Переменной total присваиваем 3200.
ptr = &total; // Получаем адрес переменной total.
val = *ptr;   // Получаем значение, хранимое
              // по этому адресу,
cout << "Rezultat: " << val << "\n";
```

Тип данных, адресуемый указателем, определяется базовым типом указателя. В данном случае, поскольку **ptr** представляет собой указатель на целочисленный тип, C++-компилятор скопирует в переменную **val** из области памяти, адресуемой указателем **ptr** , 4 байт информации (что справедливо для 32-разрядной среды), но если бы мы имели дело с **double**-указателем, то в аналогичной ситуации скопировалось бы 8 байт.

Инициализация указателей

- Указатель можно инициализировать адресом переменной, которая уже определена:
  ```
  double  dvar = 0.0;
  double *pvar = &dvar;
  ```
- Инициализация значением NULL гарантирует, что указатель не содержит адреса, который воспринимается как корректный, а значение можно проверить в инструкции if :
  ```
  int  *pinteger = NULL;
  if (pinteger = = NULL) cout << "pinteger  is null";
  ```
- равнозначная альтернатива инициализировать указатель 0:
  ```
  int  *pinteger = 0;
  if (!pinteger)  cout << "pinteger is null";
  ```

3. Свободная память и указатели

Когда начинается выполнение программы, написанной на языке C++, компилятор резервирует память для ее кода (иногда эту память называют **кодовой** (code storage), или текстовой (text storage)) и глобальных переменных (эту память называют **статической** (static storage)).

Кроме того, он выделяет память, которая будет использоваться при вызове функций для хранения их аргументов и локальных переменных (эта память называется **стековой** (stack storage), или **автоматической** (automatic storage)). Остальная память компьютера может использоваться для других целей; она называется **свободной** (free) или кучей(**heap**). Это распределение памяти можно проиллюстрировать следующим образом.

Рис. Схема памяти

Оператор **new** выполняет выделение (**allocation**) свободной памяти (**free store**). При этом:

• Оператор **new** возвращает указатель на выделенную память.

• Значением указателя является адрес первого байта выделенной памяти.

• Указатель в левой части оператора присваивания должен быть указателем на тот же тип данных.

• Указатель не знает, на какое количество элементов он ссылается.

Выделение памяти оператором `new`:

Имя_Указателя = **new** *Тип*;

Освобождение памяти:
delete *Имя_Указателя*;

Оператор **new** может выделять память как для отдельных элементов, так и для последовательности элементов с помощью оператора индексирования [ ].

double* p = new double [4]; // размещаем 4 числа double в свободной памяти

Пример. Оператором `new` выделена память для данных типа `int`, с помощью оператора `delete` выполнено освобождение памяти.

```
int *p;
p = new int; /* Выделение памяти для целого. Далее
выполняется проверка, удачно ли произошло выделение
памяти, если нет – выход из программы с кодом
завершения 1: */
if(!p) {
cout << "\nНедостаточно памяти\n";
return 1;
}
*p = 1000;
cout << "По адресу p="<<p<<" записано: "<<*p<<"\n";
```

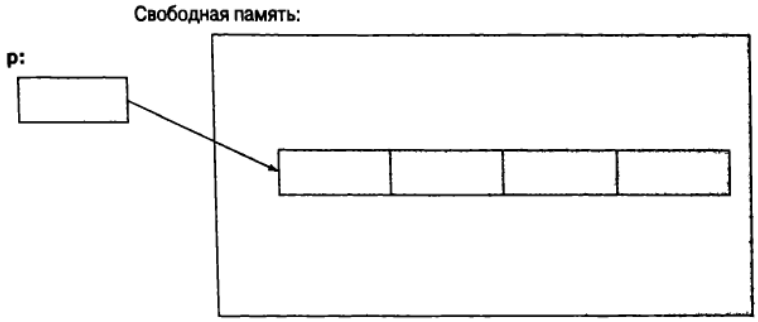
```
delete p; // освобождение памяти
```

Пример. Использование операторов `new` и `delete`

```
double *px = new double;
*px = 10.102;
int *pn = new int;
*pn = 100000;

char *pc = new char;
*pc = 'A';
cout << *px << '\t' << *pn << '\t' << *pc;
cout << '\n';
delete px;
delete pn;
delete pc;
```

Указанная выше инструкция просит систему выполнения программы разместить четыре числа типа double в свободной памяти и вернуть указатель на первое из них. Этот указатель используется для инициализации переменной p. Схематически это выглядит следующим образом.



Пример. Пример инициализации динамической переменной.

Имя_Указателя = **new** *Тип* (*инициализирующее_значение*)**;**

Значение, используемое для инициализации, указано в круглых скобках оператора `new`.

```
int *p;
p = new int(5); // начальное значение равно 5
if(!p) {
cout << "\nНедостаточно памяти\n ";
return 1;
}
cout<<"\n По адресу p="<<p<<" записано "<<*p<<"\n";
delete p; // освобождение памяти
```

4. Арифметические операции с указателями

В C/C++ разрешены несколько арифметических операций с участием переменных-указателей. К указателям можно прибавлять и вычитать целые числа, выполнять операции инкремента и декремента, а также вычитать два указателя. Использовать арифметику указателей, как правило, имеет смысл применительно к элементам массивов

Увеличение и уменьшение указателя на целое число

Увеличение указателя на `1`, означает, что указатель будет ссылаться на следующий блок памяти, занятый переменной того же типа. Увеличение же указателя на целое число `n`, передвигает указатель на `n` блоков в сторону увеличения адресов (значение указателя увеличивается на n*L, где L – длина базового типа на который ссылается указатель).

Уменьшение указателя на целое число `n`, передвигает указатель на `n` блоков (длин элементов базового типа) в сторону уменьшения адресов (значение указателя уменьшается на N*L).

Вычитание указателей

Можно вычитать два указателя одного и того же типа. Результат этой операции – количество данных, т.е. объектов базового типа, уместившихся между адресами, являющихся значениями этих указателей. Нельзя производить вычитание разнотипных указателей. Нельзя складывать указатели.

Пример. Выделяется блок для размещения 50 целых чисел.

Указателю `ps` присвоено значение `p+10`, т.е. этот указатель ссылается на 10

блок данных, а значения адресов отличаются на число 40 (28 в шестнадцатеричной системе счисления). Указатель `pf` ссылается на 40 блок данных. Из одного указателя вычитается другой.

```
int *p, *ps, *pf;
p = new int [50];
for (int i=0;i<50;i++) *(p+i) = i * 2;
ps=p+10; pf=p+40;
cout<<"\n *ps="<<*ps<<" *pf="<<*pf;
cout<<"\n p="<<p<<" ps="<<ps<<" pf="<<pf;
cout<<"\n ps-p= "<<ps-p<<" pf-ps= "<<pf-ps;
delete [] p;
```

```
*ps=20 *pf=80
p=11f1840 ps=11f1868 pf=11f18e0
ps-p= 10 pf-ps= 30_
```

Сравнение указателей

Указатели можно сравнивать между собой.

Как правило, указатели сравнивают, когда они ссылаются на один и тот же объект, например, массив.

Сравнивать указатели, если их базовый тип различается, нельзя.

5. Присваивание указателей

Переменной–указателю можно присвоить значение другого указателя того же типа данных, а также, адрес переменной того же типа.

Пример. Операция присваивания указателей, в результате которой оба указателя `px` и `py` ссылаются на область памяти, в которой размещена переменная `x`.

```
int x=1000; int y=2000;
int *px=&x; int *py=&y;
cout<<"\n px= "<<px<<" *px= "<<*px;
cout<<"\n py= "<<py<<" *py= "<<*py;
py=px;/*указателю присвоили значение другого указателя */
cout<<"\n py= "<<py<<" *py= "<<*py;
```

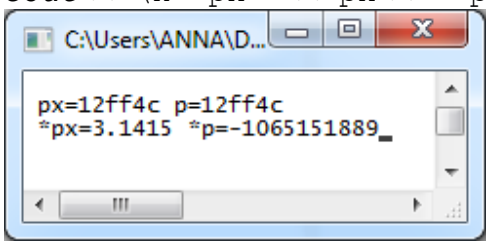
```
px= 12ff50 *px= 1000
py= 12ff4c *py= 2000
py= 12ff50 *py= 1000
```

Преобразование типа указателя

Указатель на один тип данных путем преобразования можно рассматривать как указатель на другой тип данных. Обычно подобное преобразование выполняют для указателей, имеющих тип `void*`, но разрешено выполнять преобразование и для указателей других типов, однако это может иметь

Пример. Как не стоит делать преобразование типа. Указателю `p` на `int` присвоено значение указателя `px` на `double` и выполнено необходимое преобразование типа. В результате, обе переменные `px` и `p` имеют одинаковые значения, но ссылаются на разные данные, – первый указатель ссылается на блок памяти, занятый значением типа `double`, а указатель `p` ссылается только на первые 4 байта этого блока.

```
double x=3.1415; double *px;
px = &x;
int *p;
p=(int *)px;
cout<<"\n px="<<px<<" p="<<p;
cout<<"\n *px="<<*px<<" *p="<<*p;
```



6. Указатели и массивы

Так как имя массива – это константный указатель на первый байт первого элемента массива, то, используя операцию разадресации (*), можно выполнить доступ к любому из элементов массива. Таким образом, будут полностью эквивалентны ссылки на ***i-й*** элемент массива ***x x[i]*** и ****(x+i)***. Следующие выражения являются тождествами:

```
x= =&x[0],
((x+i)= =&x[i])
```

Однако использование указателей для доступа к элементам массива дает несколько более короткий код программы и в ряде случаев позволяет обойтись без дополнительных переменных, используемых в качестве индексов.

Пусть *mas[6]* – массив из шести элементов, тогда записи *mas* и *&mas[0]* эквивалентны и определяют адрес 1-го элемента массива, то есть имя массива – это указатель его первого элемен-

та. Оба значения являются константами типа указатель, поскольку они не изменяются на протяжении всей работы программы. Однако эти значения можно присваивать переменным типа указатель:

```
int mas[6];
int *pm;
. . . .
pm=mas;
```

где ***pm*** указывает на первый элемент массива

Чтобы получить доступ к третьему элементу массива, можно записать:

```
mas[2];
```

или

```
*(pm+2);
```

Следующие записи эквивалентны:

```
a[i] ~ *(a+i);
a[i][j] ~ *(a[i]+j) ~ *(*(a+i)+j)
```

Пример. Составить программу, в которой в одномерный массив *a* из десяти элементов поместить последовательные числа от 0 до 9. Вывести содержимое массива на экран. Использовать доступ к элементам массива по указателю.

Решение:

```
int _tmain()
{
int a[10];
int t;
for(t=0;t<10;t++)
*(a+t)=t ;
for(t=0;t<10;t++)
cout<<"a["<<t<<"]="<<*(a+t)<<"\n";
getch();
}
```

Если, например, `w` имя массива, то `*w` – значение первого элемента массива, а обращение `w[i]` к `i`-му элементу массива есть сокращенная форма операции с указателями `*(w+i)`. Указатели также можно индексировать, используя запись `v[i]` вместо `*(v+i)`.

Пример.

```
const int N=10;
int w[N]=// массив
{0,10,20,30,40,50,60,70,80,90};
int *v; // указатель
v=w+5;
cout<<"\n w[0]="<<*w;
cout<<"\n w[9]="<<*(w+9);
cout<<"\n *(v+4)="<<v[4];/* указатель можно
индексировать */
```

Хотя указатели и массивы тесно связаны, между ними есть существенное различие – значение указателя можно изменять, а имя массива является константой.

Пример. Указателю b присвоено значение указателя a. Однако, операция присваивания a=p будет воспринята компилятором как ошибка, поскольку, в отличие от b указатель a является именем массива.

```
const int N=5;
int *b;
b = new int [N];
int a[N]={10,20,30,40,50};
for(int i=0; i<N; i++) b[i] = i;
cout<<"\n a: ";
for(int i=0; i<N; i++) cout<<a[i]<<'\t';
cout<<"\n b: ";
for(int i=0; i<N; i++) cout<<b[i]<<'\t';
b=a; // так можно
cout<<"\n *(b+2)="<<b[2];
/* a=b; // так нельзя */
```

Указатели и многомерные массивы

В многомерных массивах имя массива также указывает на первый элемент массива. Обращение к элементу массива, указанием его индексов, является сокращенной формой выражения с указателями. Пусть a – двумерный массив размера n*m, где n, m – заданные константы, тогда выражение a[i][j] преобразуется компилятором в операцию с указателями *(*(a+i)+j). Выражение a[i] является указателем на первый элемент строки с индексом i.

Пример.

```
const int N=3;
const int M=5;
int a[N][M]={{0,1,2,3,4},
             {1,2,3,4,5},
             {2,3,4,5,6}};
cout<<"\n Первый элемент= "<<*(*a);
cout<<"\n a[2][3]="<<*(*(a+2)+3);
int *p;
p=a[2];// адрес первого элемента строки 2
cout<<"\n p[3]="<<*(p+3);
```

Пример. Многоуровневая адресация. Обращение к одним и тем же данным с помощью массива и указателя на указатель. Указатели a и b имеют разные значения, но адреса строк a[i] и b[i] одинаковые, как следствие, обращения a[i][j] и b[i][j] дают одинаковый результат.

```
const int N=3;
const int M=5;
int a[N][M]={{0,1,2,3,4},{1,2,3,4,5},{2,3,4,5,6}};
int **b;
b= new int*[N];
for(int i=0; i<N; i++) b[i]=a[i];
cout<<"\n b: \n";
```

```
for(int i=0; i<N; i++){
for(int j=0; j<M; j++) cout<<b[i][j]<<" ";
cout<<"\n";
}
cout<<"\n a="<<a;
cout<<"\n b="<<b;
cout<<"\n a[0]="<<a[0];
cout<<"\n b[0]="<<b[0];
```

```
 b:
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6

 a=12fedc
 b=11ffed0
 a[0]=12fedc
 b[0]=12fedc
```

7. Динамическое распределение памяти

Выделение памяти для массива в процессе компиляции называется ***статическим связыванием***. Память под массив выделяется на этапе компиляции. С помощью оператора `new` можем создавать массив во время выполнения программы, размер массива также определяется на этапе выполнения программы. Такой массив называется динамическим, а процесс его создания – ***динамическим связыванием.***

Механизм динамического распределения памяти позволяет программе получать необходимую для хранения данных память в процессе своего выполнения. Для получения доступа к динамически выделяемым областям памяти и их типизации служат указатели.

Для динамически размещаемого одномерного массива используется следующая форма оператора `new`:

```
p=new type [size]
```

Для удаления динамически размещаемого одномерного массива используется оператор

```
delete [] p;
```

Пример. Размер массива устанавливается во время выполнения.

```
int i,size;
cout<<"Size=";
cin>>size;
int * pz= new int[size];
for(i=0;i<size;i++){
cout<<"pz["<<i<<"]=";
cin>>pz[i];
}
int sum=0;
for(i=0;i<size;i++) sum+=pz[i];
cout<<"summa="<<sum<<"\n";
delete [] pz; // освобождаем память
```

Двумерные динамические массивы

Пример. Размер матрицы вводится с клавиатуры во время выполнения программы. Память для размещения данных выделяется с помощью оператора `new`.

```
// Двумерный динамический массив n*m
double **a;
int n, m;
setlocale(LC_CTYPE, "rus");//русификация консоли
cout<<"\n Число строк=";
cin>>n;
cout<<"\n Число столбцов=";
cin>>m;
// Память для размещения данных:
a=new double* [n];
for (int i=0;i<n;i++) a[i]=new double [m];
// Заполняем случайными числами:
for (int i=0;i<n;i++)
for (int j=0;j<m;j++) a[i][j] = (rand()%100) * 0.1;
// Печатаем:
for(int i=0; i<n; i++){
for(int j=0; j<m; j++) cout<<a[i][j]<<" ";
cout<<"\n";
}
// освобождение памяти
for(int i=0;i<n;i++)  delete a[i];
delete   []  a;
```

Пример . Составить программу, в которой использовать одномерный динамический массив *a*, размерность и значения элементов которого ввести с клавиатуры. Вывести содержимое массива на экран.

Для решения задачи необходимо выделить память под массив, а затем созданный динамический массив надо удалить, очистив память.

Решение:

```
int _tmain()
{
int i, n;
cin>>n;                    // n - размерность массива
int *a=new int[n];  // выделение памяти под массив a
 for(i=0;i<n;i++)   //ввод элементов массива с клавиатуры
 cin>>*(a+i);
 cout<<"massiv:\n";
 for(i=0;i<n;i++)    //вывод элементов массива на экран
 cout<< *(a+i)<<"\t";
delete a;           // освобождение памяти
getch();
}
```

В данном примере *a* является указателем на массив из *n* элементов. Оператор *int *a=new int[n];* производит два действия: *объявляется* переменная типа указатель, а затем *указателю присваивается адрес* выделенной области памяти в соответствии с заданным типом объекта. Для лучшего понимания работы с динамическими массивами эти две операции можно разделить.

Для рассмотренного примера можно задать следующую эквивалентную последовательность операторов:

```
int n, *a;
cin>>n;          // n – размерность массива
a=new int[n];    // выделение памяти под массив
delete a;        // освобождение памяти
```

Если с помощью операции *new* невозможно выделить требуемый объем памяти, то результатом операции *new* является значение 0.

Иногда при программировании возникает необходимость создания *многомерных динамических объектов.* Начинающие программисты по аналогии с описанным способом создания одномерных динамических массивов для двумерного динамического массива размерности n×k могут пытаться выделить память с помощью следующей записи операции *new:*

```
mas=new int[n][k];   // Неверно! Ошибка!
```

Такой способ выделения памяти не дает верного результата. Будет выдаваться ошибка о несоответствии типов указателей. Попытка устранить подобную ошибку с помощью операции явного преобразования типов приводит к ошибкам при обращении к элементам массива из-за неверного типа выделенной области памяти.

Пример 6. Составить программу, в которой создать двумерный динамический массив целых чисел *mas*, размерность ($n \times к$) которого ввести с клавиатуры. Значения элементов рассчитать как сумму соответствующего номера строки и столбца ($i + j$). Вывести содержимое массива на экран.

Решение:

```
int _tmain()
{
 int n,k,i,j,**mas;
 cin>>n;              // n – число строк массива
 cin>>k;              // k - число столбцов массива
//выделение памяти под n указателей на строку
 mas=new int*[n];
// выделение памяти для каждой
// строки по числу столбцов к
 for(i=0;i<n;i++)
 mas[i]=new int[k];
// строки по числу столбцов к
for(i=0;i<n;i++)
{
    for(j=0;j<k;j++)
    {
       *(*(mas+i)+j)=i+j;
       cout<<mas[i][j]<<"\t";
    }
    cout<<"\n";
}
// освобождение памяти
for(i=0;i<n;i++)  delete mas[i];
delete   []  mas;
getch();
}
```

Для решения рассмотренного выше примера сначала необходимо с помощью операции *new* выделить память под *n* указателей. Выделенная память будет представлять собой вектор, элементом которого является указатель. При этом все указатели располагаются в памяти последовательно друг за другом. После этого необходимо в цикле каждому указателю присвоить адрес выделенной области памяти размером, равным второй границе массива.

Таким образом, в отличие от обычного двумерного массива, когда при объявлении выделяется сплошной участок памяти размером, равным произведению его границ, для двумерного динамического массива память выделяется с помощью нескольких операций *new* и не является сплошным участком (рис. 1).

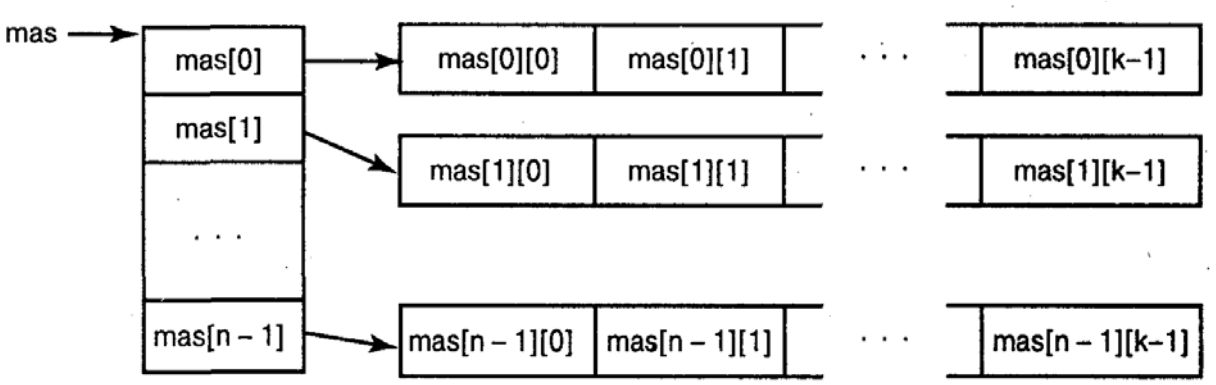


Рисунок 1. Размещение динамического двумерного массива в памяти

В C++ определено понятие «вектор». Любой массив при объявлении представляет собой ***вектор.*** Динамически память может быть выделена как вектор. Из этого следует невозможность явного выделения памяти под многомерный массив. Поэтому при работе с многомерными динамическими объектами следует пользоваться вышеописанным способом.

8. Массивы указателей

Указатели, подобно данным других типов, могут храниться в массивах. Вот, например, как выглядит объявление 10-элементного массива указателей на **int** − значения.

```
int *pi[10];
```

Здесь каждый элемент массива `pi` содержит указатель на целочисленное значение. Чтобы присвоить адрес `int`-переменной с именем `var` третьему элементу этого массива указателей, запишите следующее.

```
int var;
pi[2]=&var;
```

Помните, что здесь `pi` − массив указателей на целочисленные значения. Элементы этого массива могут содержать только значения, которые представляют собой адреса переменных целочисленного типа, а не сами значения. Вот поэтому переменная `var` предваряется оператором "&".

Чтобы узнать значение переменной `var` с помощью массива `pi` , используйте такой синтаксис.

```
*pi[2]
```

Поскольку адрес переменной `var` хранится в элементе `pi[2]`, применение оператора "*" к этой индексированной переменной и позволяет получить значение переменной `var`.

Подобно другим массивам, массивы указателей можно инициализировать.

```
int *pparint [10]; //массив указателей на int
int a=0;
int b=1;
pparint [1]=&a;
pparint [2]=&b;
*pparint[1]= *pparint[2];
cout<<pparint[1] <<"  "<< pparint[2]<<"\n";
cout<<*pparint[1]<<"  "<<*pparint[2];
```

```
12ff50  12ff4c
1  1
```

Многоуровневая непрямая адресация

Можно создать указатель, который будет ссылаться на другой указатель, а тот − на конечное значение. Эту ситуацию называют цепочкой указателей, *многоуровневой непрямой адресацией* (multiple indirection) или использованием *указателя на указатель*. Значение обычного указателя (при одноуровневой непрямой адресации) представляет собой адрес переменной, которая содержит некоторое значение. В случае применения указателя на указатель первый содержит адрес второго, а тот ссылается на переменную, содержащую определенное значение.

При использовании непрямой адресации можно организовать любое желаемое количество уровней, но, как правило, ограничиваются лишь двумя, поскольку увеличение числа уровней часто ведет к возникновению концептуальных ошибок.

Переменную, которая является указателем на указатель, нужно объявить соответствующим образом. Для этого достаточно перед ее именем поставить дополнительный символ "звездочка"(*). Например, следующее объявление сообщает компилятору о том, что balance − это указатель на указатель на значение типа int .

int **balance;

Необходимо помнить, что переменная balance здесь − не указатель на целочисленное значение, а указатель на указатель на int-значение.

Чтобы получить доступ к значению, адресуемому указателем на указатель, необходимо дважды применить оператор "*", как показано в следующем примере.

```
int x, *p, **q;
x = 10;
// Присваиваем p адрес переменной x
p = &x;
// Присваиваем q адрес переменной p
q = &p;
cout << **q;
//Выводим значение переменной x.
//Доступ к значению переменной x получаем
//через  указатель q. Обратите внимание на
// использование двух символов "*"
```

Здесь переменная p объявлена как указатель на int-значение, а переменная q — как указатель на указатель на int-значение. При выполнении этой программы на экран будет выведено значение переменной x, т.е. число 10.