СТАТИСТИЧЕСКОЕ МОДЕЛИРОВАНИЕ РАДИОТЕХНИЧЕСКИХ СИСТЕМ И УСТРОЙСТВ С ИСПОЛЬЗОВАНИЕМ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++

Лектор доц. каф. ТОРС ПГУТИ Алышев Ю. В.

Литература

1. **Теория электрической связи: Учебник для вузов/ А.Г. Зюко, Д.Д. Кловский, В.И. Коржик, М.В. Назаров; под ред. Д.Д. Кловского. – М.: Радио и связь, 1998.**
2. **Прокис Дж. Цифровая связь. Пер с англ. / Под ред. Д.Д. Кловского. – М.: Радио и связь. 2000.**
3. **Язык С. Руководство для начинающих. Уэйт М., Прата С., Мартин Д., М., Мир 1988.**
4. **Объектно-ориентированное программирование на С++. А. Пол Спб., М., «Невский диалект», «Изд. БИНОМ» 1999.**
5. **Справочник по математике для инженеров и учащихся втузов. Бронштейн И. Н., Семендяев К.А. – М. Наука. 1981.**
6. **Компьютерное моделирование передачи и приёма двоичных сигналов в канале с гауссовским шумом с использованием объектно-ориентированного программирования на С++. Методическая разработка к практическим занятиям для студентов дневной формы обучения по дисциплине «объектно-ориентированное программирование на С++» Составители: Алышев Ю. В., Борисенков А. В., Самара 2006.**

Методичка по лабораторным занятиям и конспект лекций:

http://tors.psati.ru

Дополнительная литература

7. **Лагутенко О.И. Модемы. Справочник пользователя. Спб.: «Лань», 1997, – 368 с.**
8. **Мейер Бертран Объектно-ориентированное конструирование программных систем / Пер. с англ. – М.: Издательско-торговый дом «Русская редакция», 2005. – 1232 с.**

Лекция 1

Абстрактная модель цифровой системы связи (ЦСС)

a_i – сообщение на передаче,

$\hat{a}_i$ – оценка сообщения на приёме,

i –– индекс, соответствующий отдельному сообщению (символу) и характеризующий порядковый номер временной последовательности.

Анализ качества ЦСС

Теория: $p_{\text{ош}}$ – вероятность ошибки

	символ на передаче	оценка на приёме
Нет ошибок	0	0
	1	1
Ошибка	0	1
	1	0

Эксперимент: $p'_{\text{ош}} = \dfrac{n_{\text{ош}}}{N}$ – частость ошибки

$n_{\text{ош}}$ – число ошибок, в течение сеанса связи при передаче N символов.

Задачей передатчика является преобразование сообщений в сигналы, с помощью которых эти сообщения можно передавать по каналу связи. В приёмнике делается обратное преобразование и на его выходе ожидается последовательность переданных сообщений. В реальных условиях в канале связи сигналы искажаются под воздействием различных помех. Поэтому на приёмной стороне производится оценивание сигналов.

Приёмное устройство, производящее оценивание, при приёме цифровых *m*-ичных выносит решение в пользу одного из них. Но иногда приёмное устройство может принимать неправильные решения. Как часто и при каких условиях приёмное устройство может выносить неправильное решение? Для ответа на этот вопрос необходим анализ качества рассматриваемой системы связи. А для того чтобы произвести анализ качества, необходимо построить модель системы связи.

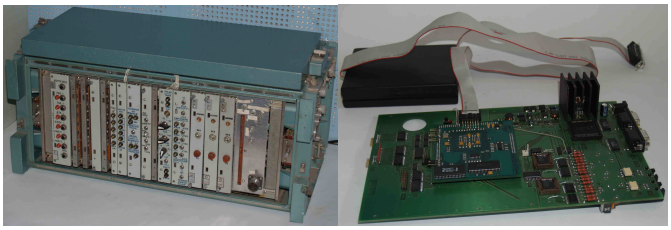
Модели системы связи бывают математические, компьютерные и реальные в виде тестирующих устройств.

Математические модели рассмотрены в курсе «Теория электрической связи».

Реальные модели используют для отладки разрабатываемых макетов устройств связи. Для этого необходимо приобретать соответствующие устройства, например имитаторы радиоканала, устройства частотного сдвига, приборы выявления ошибок и т. п.

Имитатор радиоканала ИРК-2

Приборы выявления ошибок



Радиомодемы

Компьютерные модели представляют собой программную модель, или отдельные программные элементы реальных устройств и радиоканала.

Компьютерная модель – наиболее эффективное средство разработки и отладки и анализа разрабатываемых устройств связи.

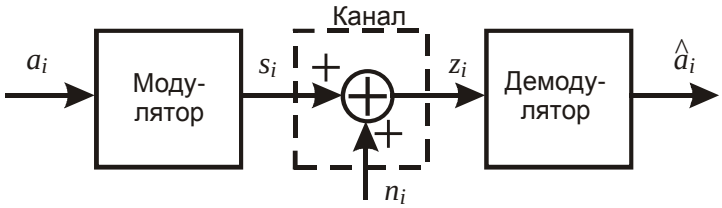
Анализ качества цифровой системы связи

Рассмотрим подробнее процесс передачи цифрового сообщения.

Для передачи по каналам связи цифровое сообщение преобразуют в сигнал. Для того, чтобы этот сигнал передать по радиолинии необходимо принять во внимание его спектр. Спектр сигнала должен быть ограничен практически, то есть иметь конечную ширину. Далее этот сигнал (его спектр) необходимо перенести в область частот, на которой целесообразно передавать его через радиоканал. На приёмной стороне происходит обратное преобразование спектра сигнала. Далее производится анализ принятого колебания, в результате которого выносится решение в пользу символа, наиболее вероятного по заданному критерию качества.

При переносе спектра сигнала вверх на передающей стороне и вниз на приёмной в устройствах происходят линейные преобразования, которые можно не учитывать при постановке задачи анализа качества полученной оценки $\hat{a}_i$. Анализ качества оценки $\hat{a}_i$ можно производить на базе и математической, и компьютерной модели. Но искажения спектра сигнала в радиоканале можно моделировать и не производя переноса его в верхнюю область частот. Для этого, в простейшем случае, искажения в канале могут быть получены с помощью добавления гауссовского шума.

Модель системы связи, содержащей канал с аддитивным гауссовским шумом



модулятор – устройство, в котором параметр переносчика (несущей) меняется по закону первичного (входного) сигнала;

демодулятор – устройство, которое выносит решение в пользу той или иной гипотезы при анализе сигнальной реализации, искажённой
в канале связи.

Под гипотезами понимаются символы, которые были использованы в качестве информационного сообщения на передаче. Количество гипотез обычно соответствует количеству символов на передающей стороне. В простейшем случае рассматривают передачу двоичных символов (бит).

В реальных устройствах связи сигнал на выходе модулятора и на входе демодулятора аналоговый. Однако при моделировании этих устройств на ЭВМ необходимо перейти к дискретному или цифровому сигналу. С учётом того, что над спектром сигнала не производится преобразование вверх и, согласно условию теоремы Котельникова, сигнал на выходе модулятора можно представить в виде последовательности отсчётов с заданной частотой дискретизации.

$$f_д \geq 2 f_в,$$

где $f_в$ – верхняя частота в спектре сигнала.

Конкретную частоту дискретизации выбирают из ряда стандартизованных частот и являющейся наиболее эффективной для поставленной задачи.

В рамках поставленной задачи гауссовский шум также можно представить в виде случайного сигнала, дискретные значения которого имеют гауссовское распределение и следуют с той же частотой дискретизации.

Для систем связи, передающих двоичную информацию, одной из качественных характеристик является *вероятность ошибки*. Для некоторых известных сигналов, использующих той или иной вид модуляции, выведены теоретические значения вероятности ошибки. При моделировании на ЭВМ понятию «вероятность ошибки» соответствует понятие *частость ошибки*, то есть насколько часто появляются ошибки в текущем сеансе связи. Частость ошибки определяется как

$$p'_{ош} = \frac{n_{ош}}{N} \qquad (1)$$

где $n_{ош}$ – число ошибок, которые произошли за текущий сеанс связи при передаче *N* символов.

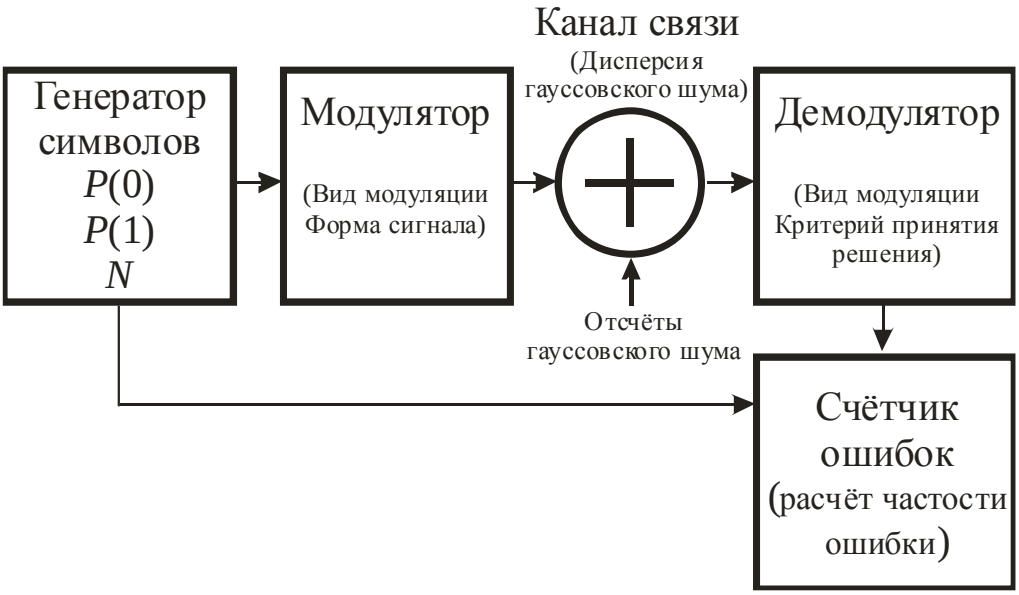
Из (1) видно, что частость ошибки можно определить, лишь зная число ошибок в текущем сеансе связи. Поэтому число ошибок $n_{ош}$ является первичным параметром для процесса моделирования.

Модель для статистических испытаний

При многократном повторении эксперимента с передачей информационной последовательности заданной длины можно получить ряд случайных значений параметра $n_{ош}$. В рамках теории математической статистики речь идёт о проведении статистических испытаний, где число передаваемых символов в каждом эксперименте является численным параметром, характеризующим объём выборки.

Для проведения статистических испытаний должна быть построена соответствующая модель. Эту модель можно эффективно реализовать на персональном компьютере.

Модель для статистических испытаний должна содержать источник сигнала, выдающий двоичные символы, модель модулятора, модель канала связи, вносящего искажения в сигнал, модель демодулятора и счётчик ошибок.



$P(0)$ – вероятность выдачи генератором символа «0»
$P(1)$ – вероятность выдачи генератором символа «1»
N – число переданных символов

Для реализации данной модели на компьютере предлагается воспользоваться наиболее распространённым инструментом для написания технических программ, таким как язык C++. Мотивацией в пользу этого инструмента является то, что устройства обработки цифровых сигналов делаются чаще всего на базе сигнальных процессоров. При этом программное обеспечение для сигнальных процессоров содержит два языка программирования: ассемблер и язык C/C++. Язык C является неким универсальным инструментом, который имеет единую форму для большого количества сигнальных процессоров, которые могут иметь разную производственную платформу. При реализации модели на языке C легче перенести предложенные алгоритмы на реальную цифровую основу.

ВЕРОЯТНОСТЬ ОШИБКИ ДЛЯ ФАЗОВОЙ МОДУЛЯЦИИ (ФМ), МОДУЛЯЦИИ ОРТОГОНАЛЬНЫМИ СИГНАЛАМИ (МОС), ЧАСТОТНОЙ МОДУЛЯЦИИ (ЧМ) И АМПЛИТУДНОЙ МОДУЛЯЦИИ (АМ) ПРИ КОГЕРЕНТНОМ ПРИЁМЕ В ОДНОЛУЧЕВОМ КАНАЛЕ:

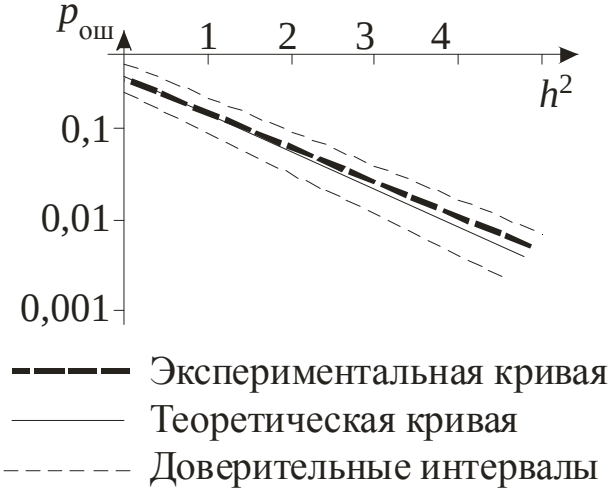
$$p_{\text{ФМ}} = Q\left(\sqrt{2h^2}\right)$$

$$p_{\text{МОС или ЧМ}} = Q\left(\sqrt{h^2}\right)$$

$$p_{\text{АМ}} = Q\left(\sqrt{h^2/2}\right)$$

$$Q(x) = \frac{1}{\sqrt{2\pi}}\int_x^{\infty} \mathrm{e}^{-t^2/2}dt \ - \text{функция ошибок}$$

ГРАФИК КРИВЫХ ПОМЕХОУСТОЙЧИВОСТИ



ДОВЕРИТЕЛЬНЫЕ ИНТЕРВАЛЫ:

$$p_{1,2} = p \pm \Phi^{-1}\left(P_{\text{дов}}\right)\sqrt{\frac{p(1-p)}{n}},$$

Где доверительная вероятность: $P_{\text{дов}} = 0,95,$

Число испытаний: n = 100000,

P – рассчитанная вероятность ошибки.

$\Phi^{-1}(x)$ – функция, обратная

Функции крампа: $\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_{-x}^{x} \mathrm{e}^{-t^2/2}dt$

Для нахождения значений функций $Q(x)$ и $\Phi(x)$ воспользуйтесь литературой [5, стр. 76–77], где табулированы значения функции $\Phi_0(x)$. При этом

$$\Phi_0(x) = \frac{1}{\sqrt{2\pi}}\int_0^x e^{-t^2/2}dt = \frac{1}{2}\Phi(x) = \frac{1}{2} - Q(x).$$

Лекция 2

Исторические сведения

Объектно-ориентированное программирование (ООП) – стиль программирования, который фиксирует поведение реального мира, скрывая детали его реализации, что позволяет решать проблемы в терминах предметной области.

ООП – это основная методология программирования 90-х годов, она представляет собой продукт 35 лет практики и опыта начиная с языка Simula 67 затем SmallTalk, Lisp, Clu, Actor, Eiffel, Objective C и C++. Это стиль программирования, который фиксирует поведение реального мира таким способом, при котором детали его реализации скрыты. Это позволяет решать проблемы в терминах предметной области.

Развитие языка

Simula 67 → SmallTalk → Lisp → Clu → Actor → Eiffel → Objective C → C++

C++ создан Б.Страуструпом в начале 80-х. Основная идея сделать C++ совместимым с C и расширить его конструкциями ООП, основанных на конструкциях типа классов из языка Simula67.

Язык Си разработан Д. Ритчи в начале 70-х годов.

Ему предшествовали и оказали на него серьёзное влияние язык BCPL, разработанный М.Ричардсоном и язык Би(B), созданный К.Томпсоном.

В C++ из BCPL введены коментарии типа //.

Из языка Simula67 позаимствована концепция класса вместе спроизводными классами и функциями членами. Это сделано, чтобы способствовать модульности благодаря использованию виртуальных функций.

Из языка Алгол68 использована возможность перегрузки операций и свобода в расположении описаний.

Ранние версии языка C++ именовались как “Си с классами”.

Название C++ придумал Р.Масситти. C++ означает эволюционную природу перехода от Си к C++. ++ – операция приращения. (Лучше бы ++C). Названия Ди язык не получил, поскольку он является расширением Си.

Летом 1983 года создан комитет по созданию ANSI-стандарта языка C.

Сравнение некоторых языков и принадлежность их к определённому уровню

C называют языком среднего уровня.

Высокий уровень	Ада
	Модула-2
	Паскаль
	COBOL
	Фортран
	Бейсик
Средний уровень	C++
	C
	FORTH
	Макро Ассемблер
Нижний уровень	Ассемблер

Язык С позволяет манипулировать битами, байтами и адресами – основными элементами с которыми работает компьютер.
Языки высокого уровня поддерживают концепцию типов данных. Тип данных определяет набор значений, которые переменная может хранить, и набор операций, которые могут выполняться над переменной.

С не делает проверку выхода за границы массива. Эта проверка возлагается на программиста.

Алфавит и числа

_ A a B b … Z z – *различаемые символы*
Идентификатор – имя переменной или функции
различаются первые **32** символа
15 десятичные
0x1F шестнадцатеричные
054 восьмеричные

Перевод чисел из одной системы исчисления в другую

Преобразование двоичного числа в десятичное можно выполнить следующим образом:

$$10011_2 = 1\times2^4 + 0\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 16 + 0 + 0 + 2 + 1 = 19_{10}$$

Обратное преобразование десятичных чисел в двоичные производится непрерывным делением преобразуемого числа на 2 с одновременным слежением за получающимися остатками, например:

9 ÷ 2 = 4 остаток 1
4 ÷ 2 = 2 остаток 0
2 ÷ 2 = 1 остаток 0
1 ÷ 2 = 0 остаток 1

$9_{10} = 1\,0\,0\,1_2$

Преобразование из двоичной в восьмеричную делается по таблице

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

100010_2

Например: 42_8

Преобразование из двоичной в шестнадцатеричную делается по таблице

```
0000 0
0001 1
0010 2
0011 3
0100 4
0101 5
0110 6
0111 7
1000 8
1001 9
1010 A
1011 B
1100 C
1101 D
1110 E
1111 F
```

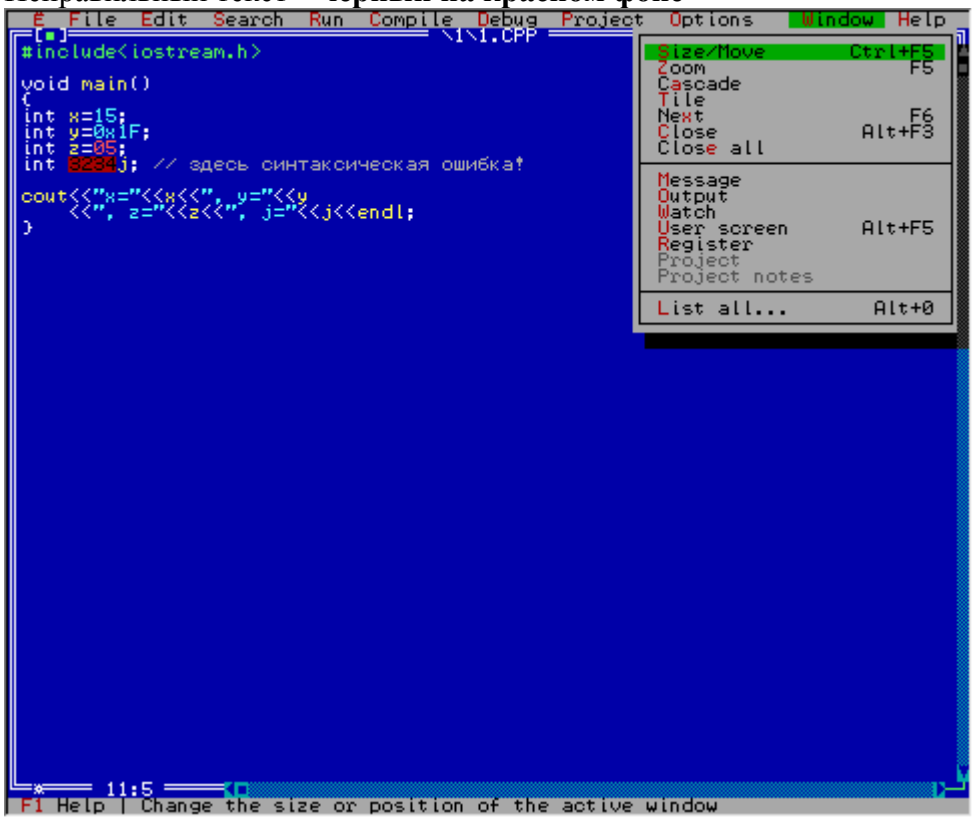
01011111_2

Например: $5F_{16}$

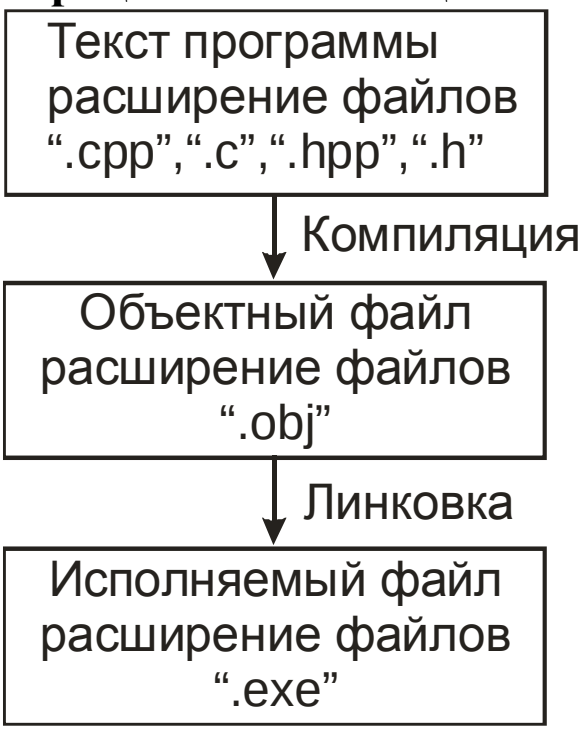
Преобразование из шестнадцатеричной (восьмеричной) в десятичную и обратно удобно делать через перевод в двоичную систему исчисления.

16-битный редактор от фирмы Borland v3.1

Цвет символов в тексте программы:
Имена переменных и функций – **жёлтый**
Ключевые слова, знаки – **белый**
Комментарии – **серый**
Обращение к препроцессору – **зелёный**
Цифры десятичные, шестнадцатеричные, символьные и строковые константы – **голубой**
Цифры восьмеричные – **магента**
Неправильный текст – **чёрный на красном фоне**



Блок-схема процесса компиляции и линковки



Кодировка с обратным слешем

Код	Значение
\b	Забой
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\”	Двойная кавычка
\’	Одинарная кавычка (апостроф)
\0	Нулевой символ (NULL)
\\	Обратный слэш
\a	Звонок
\N	Восьмеричная константа (N – восьмеричное значение)
\xN	Шестнадцатеричная константа (N – шестнадцатеричное значение)

Специальные символы

;

– символ окончания выражения

//

– комментарий до конца строки

/* … */

– комментарий, позволяющий объединять несколько строк

=

– оператор присваивания

ВСТРОЕННЫЕ ТИПЫ ДАННЫХ

Все допустимые комбинации базовых типов и модификаторов для 16-битных слов

Тип	Длина в битах	Диапазон
Целочисленные типы ЗНАК ЦЕЛОЧИСЛЕННАЯ МАНТИССА 1 | Длина в битах-1		
char signed char	8	-128…127
unsigned char	8	0…255
unsigned int (unsigned) unsigned short int	16	0…65535
int signed int short int signed short int	16	-32768…32767
long int (long) signed long int	32	-2147483648… 2147483647
unsigned long int (unsigned long)	32	0…4294967295
Типы с плавающей точкой ЗНАК ПОРЯДОК ДРОБНАЯ ЧАСТЬ МАНТИССЫ Z | N_1 | N_2 Для long double: ЗНАК ПОРЯДОК ЦЕЛАЯ И ДРОБНАЯ ЧАСТЬ МАНТИССЫ Z | N_1 | $1+N_2$		
Тип	**Бит**	**Диапазон**
Float 1 | 8 | 23 стандарт IEEE-754	32	1.18e-38… …3.40e+38 (точность 7 знаков)
Double 1 | 11 | 52	64	2.23e-308… …1.79e+308 (точность 15 знаков)
long double 1 | 15 | 1 | 63	80	3.37e-4932… …1.18e+4932 (точность 19 знаков)

$$2^{23+1} = 1\underbrace{6777216}_{7}$$

$$\lfloor \lg 16777216 \approx 7,225 \rfloor = 7$$

$$2^{52+1} = 9\underbrace{007199254740992}_{15}$$

$$\lfloor \lg 9007199254740992 \approx 15,9546 \rfloor = 15$$

$$2^{63+1} = 18\underbrace{446744073709551616}_{19}$$

$$\lfloor \lg 18446744073709551616 \approx 19,266 \rfloor = 19$$

$$2^8 = 256$$

$$\lg 2^{256} \approx 77 \rightarrow \pm 38$$

$$2^{11} = 2048$$

$$\lg 2^{2048} \approx 616 \rightarrow \pm 308$$

$$2^{15} = 32768$$

$$\lg 2^{32768} \approx 9864 \rightarrow \pm 4932$$

Представление двоичных целых

Класс	Знак	Мантисса
Положительные		
(Больше)	0	11…11
	…	…
(Меньше)	0	00…01
Нуль	0	00…00
Отрицательные		
(Меньше)	1	11…11
	…	…
(Больше/Неопределённость)	1	00…00
	Слово:	15 бит
	Короткое:	31 бит
	Длинное:	64 бита

Представление мантиссы числа для нормализованных чисел с плавающей точкой

float

$$(1-2Z)2^{N_1-127}\left(1+\frac{N_2}{2^{23}}\right)$$

double

$$(1-2Z)2^{N_1-1023}\left(1+\frac{N_2}{2^{52}}\right)$$

long double

$$(1-2Z)2^{N_1-16383}\left(1+\frac{N_2}{2^{63}}\right)$$

Пример программы проверяющей соответствие формата с плавающей точкой заявленным битовым позициям

```
#include<fstream.h>
union lf
{
long l;
float f;
};
union ld
{
long l[2];
double d;
};
union sl
{
short s[5];
long double l;
};
void main()
{
lf y;
ld x;
sl z;
y.l=0x007fffff;//1.175494e-38
y.l=0x00000001;//1.401298e-45
y.l=0x7f800000;//+INF
y.l=0x7f800001;//sNaN
y.l=0x7fdfffff;//sNaN
y.l=0x7fe00000;//QNaN
y.l=0x7fffffff;//QNaN
y.l=0x3fc00000;//3.402823e+38
y.l=0x40800000;//3.402823e+38
y.l=0xc0800000;//3.402823e+38
y.l=0x7f7fffff;//3.402823e+38
y.l=0x00800000;//1.175494e-38
cout<<y.f<<endl;
x.l[1]=0x7fefffff;x.l[0]=0xffffffff; //1.797693134862315e+308
x.l[1]=0x00100000;x.l[0]=0x00000000; //2.225073858507201e+308
cout.precision(15);
cout<<x.d<<endl;
z.s[4]=0x7ffe;z.s[3]=0xffff;z.s[2]=0xffff;z.s[1]=0xffff;
z.s[0]=0xffff;//1.18973149535723176e+4932
z.s[4]=0x0001;z.s[3]=0x8000;z.s[2]=0x0000;z.s[1]=0x0000;
z.s[0]=0x0000;//3.3621031431120934965e-4932
z.s[4]=0x4000;z.s[3]=0x8000;z.s[2]=0x0000;z.s[1]=0x0000;
z.s[0]=0x0000;//2
cout.precision(19);
cout<<z.l<<endl;
}
```

float	double	long double
max=3,402823e+38 min=1,175494e-38 min'=1,401298e-45	max=1,797693134862316e+308 min=2,225073858507201e-308 min'=4,940656458412465e-324	max=1,189731495357231765e+4932 min=3,362103143112093506e-4932 min''=1,681051571556046753e-4932 min'=3,645199531882474603e-4951

max – максимально возможное для данного типа
min – ненулевое минимально возможное нормализованное для данного типа
min’’ – ненулевое минимально возможное ненормализованное для данного типа
min’ – ненулевое минимально возможное денормализованное для данного типа

Представление коротких (short-real) и длинных (long-real) вещественных

Класс	Знак	Порядок	Мантисса ff-ff
Положительные «не числа» (NaN)			
Quiet (QNaN)	0 … 0	11…11 … 11…11	11…11 … 10…00
Signaling (SNaN)	0 … 0	11…11 … 11…11	01…11 … 00…01
Бесконечность (+INF)	0	11…11	00…00
Положительные вещественные			
Нормализованные	0 … 0	11…10 … 00…01	11…11 … 00…00
Денормализованные	0 … 0	00…00 … 00…00	11…11 … 00…01
Нуль	0	00…00	00…00
Отрицательные вещественные			
Нуль	1	00…00	00…00
Денормализованные	1 … 1	00…00 … 00…00	00…01 … 11…11
Нормализованные	1 … 1	00…01 … 11…10	00…00 … 11…11
Бесконечность (–INF)	1	11…11	00…00
Отрицательные «не числа» (NaN)			
Signaling (SNaN)	1 … 1	11…11 … 11…11	00…01 … 01…11
(Неопределённость QNaN) Quiet (QNaN)	1 … 1	11…11 … 11…11	10…00 … 11…11
Короткое:		8 бит	23 бита
Длинное:		11 бит	52 бита

Представление временных вещественных

Класс	Знак	Порядок	Мантисса ff-ff
Положительные «не числа» (NaN)			
Quiet (QNaN)	0 … 0	11…11 … 11…11	111…11 … 110…00
Signaling (SNaN)	0 … 0	11…11 … 11…11	101…11 … 100…01
Бесконечность (+INF)	0	11…11	100…00
Положительные вещественные			
Нормализованные	0 … 0	11…10 … 00…01	111…11 … 100…00
Ненормализованные	0 … 0	00…00 … 00…00	111…11 … 100…01
Денормализованные	0 … 0	00…00 … 00…00	011…11 … 000…01
Нуль	0	00…00	000…00
Отрицательные вещественные			
Нуль	1	00…00	000…00
Денормализованные	1 … 1	00…00 … 00…00	000…01 … 011…11
Ненормализованные	1 … 1	00…00 … 00…00	100…01 … 111…11
Нормализованные	1 … 1	00…01 … 11…10	100…00 … 111…11
Бесконечность (–INF)	1	11…11	100…00
Отрицательные «не числа» (NaN)			
Signaling (SNaN)	1 … 1	11…11 … 11…11	100…01 … 101…11
(Неопределённость QNaN) Quiet (QNaN)	1 … 1	11…11 … 11…11	110…00 … 111…11
		15 бит	64 бита

Лекция 3

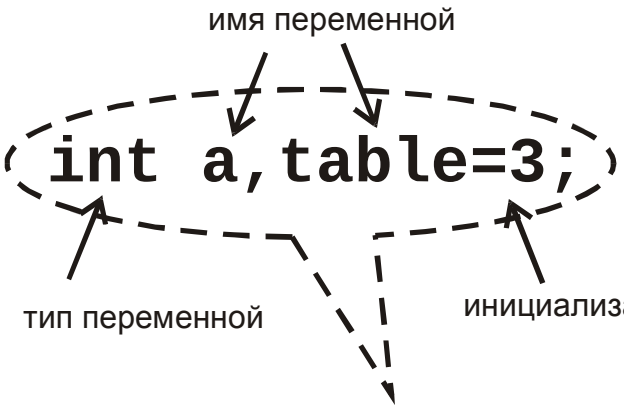
Объявление переменных

объявление целочисленной переменной

```
int a;
```

объявление двух переменных с плавающей точкой

```
float x,y;
```



Объявление двух целочисленных переменных. Это означает, что в памяти ЭВМ под эти переменные отводится место, равное размеру типа этих переменных, умноженного на их количество. Для каждой переменной в памяти машины закрепляется определённый адрес.

В данном случае под эти две переменные выделяется 4 байта. По адресу в памяти машины, закрепленного за переменной **`table`** записывается число 3

Объявление констант

```
const int dozen=12;//целочисленная константа
const char* = "Слово";// строковая константа
const char='A'; // символьная константа
const float E=2.71828;
```

Перечислимые типы

enum – особый целочисленный тип с набором именованных целых констант или их называют перечислимыми константами.

Пример:

```
enum boolean { false, true };
enum answer { yes, no, maybe=-1 };
enum signal { off, on } variable;
```

Пустой тип

```
void
```

ПРЕОБРАЗОВАНИЯ ТИПОВ

1. Любое `char`, `short` или `enum` преобразуются в `int`. Составные значения непредставимые как `int`, преобразуются в `unsigned`.

2. Если выражение имеет смешанный тип, тогда операнд более низкого типа преобразуется в операнд более высокого типа, согласно иерархии, и значение выражения имеет этот же тип.

```
int < unsigned < long < unsigned long < float < double < long double
```

Примеры:

```
float z;
int x=3,y=2;
z=x/y;
cout<<z<<endl; // Результат 1
```

```
z=float(x)/y;
z=x/float(y);
z=float(x)/float(y);
// функциональный вид
// явного преобразования
```

```
z=(float)x/y;
z=x/(float)y;
z=(float)x/(float)y;
// операторный вид
// явного преобразования
```

```
z=3/y; // Результат 1
z=3./y; // Результат 1,5
```

Библиотечные функции

Некоторые функции и константы определённые в математической библиотеке `math.h`

`double sin(double x);` – синус;

`double cos(double x);` – косинус;

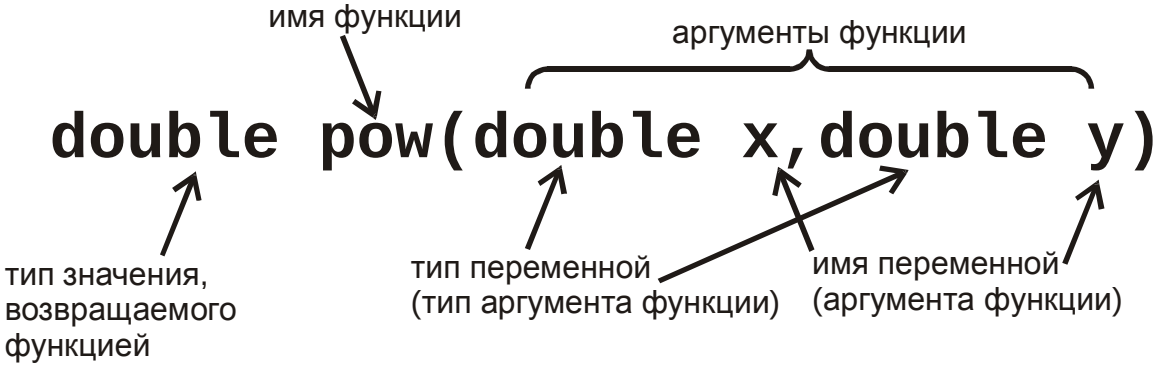
`double exp(double x);` – экспонента

`double pow(double x, double y);` – возведение x в степень y;

double sqrt(double x); – квадратный корень

double tan(double x); – тангенс

double atan2(double x,double y); – арктангенс от двух аргументов x/y (результат от $-\pi$ **до** $+\pi$);



M_PI – Константа π

M_PI_2 – Константа π/2

Константа π определена в библиотечном файле math.h следующим образом:

```
#ifndef M_PI
 #define M_PI        3.14159265358979323846
#endif
```

Пример использования функций:

$y = \frac{\sin\left(e^{-x^3}\right)}{\sqrt{2\pi}}$ – пример некоторой функции

```
y=sin(exp(-pow(x,3)))/sqrt(2*M_PI); // запись её в программе
```

Операторы

Опе-ратор	Описание	Пример
Арифметические операторы		
+	Сложение	x = x + z
–	Вычитание	x = x – z
*	Умножение	x = x * z
/	Деление	x = x / z
%	Взятие по модулю (остаток от целочисленного деления)	x = x % z
Битовые операторы		
&	Оператор **И**	y = x & 0x1F;
|	Оператор **ИЛИ**	y = x | 0x1F;
^	Оператор **Исключающее ИЛИ**	y = x ^ 0x1F;
~	Дополнение (Поразрядное **НЕ**)	x = ~0x02 (Результат:x=0xFD)
>>	Арифметический сдвиг вправо	y = x>>3; $(y = x/2^3)$
<<	Арифметический сдвиг влево	y = x<<5; $(y = x \cdot 2^5)$

Операторы присваивания		
=	Присваивание	x = 10;
+=	Сложение с присваиванием (накопление)	x += 10; (то же, что и x = x + 10;)
-=	Вычитание с присваиванием	x -= 10; (то же, что и x = x - 10;)
*=	Умножение с присваиванием	x *= 10; (то же, что и x = x * 10;)
/=	Деление с присваиванием	x /= 10; (то же, что и x = x / 10;)
%=	Взятие по модулю с присваиванием	x %= 10; (то же, что и x = x % 10;)
&=	Поразрядное **И** с присваиванием	x &= 10; (то же, что и x = x & 10;)
|=	Поразрядное **ИЛИ** с присваиванием	x |= 10; (то же, что и x = x | 10;)
^=	Поразрядное **Исключающее ИЛИ** с присваиванием	x ^= 10; (то же, что и x = x ^ 10;)
>>=	Сдвиг вправо с присваиванием	x >>=5; (то же, что и x = x>>5;)
<<=	Сдвиг влево с присваиванием	x <<=5; (то же, что и x = x<<5;)
Операция присваивания		
=	Позволяет объединять операторы присваивания.	a=b=0; (то же, что и a=0; b=0;)
Операция запятая		
,	Объединяет несколько операторов в единый оператор	a=1, b=2; (то же, что и { a=1; b=2; }
Унарные операторы		
*	Косвенная адресация (разыменовывание)	int x = *y;
&	Взятие адреса	int* x = &y;
++	Инкремент	x++; ++x; (то же, что и x = x + 1;)
--	Декремент	x--; --x; (то же, что и x = x - 1;)

Логические операторы		
!	Логическое **НЕ**	if(!valid) {…}
&&	Логическое **И**	if(x&&0xFF) {…}
||	Логическое **ИЛИ**	if(x||0xFF) {…}
Операторы отношения		
!=	Не равно	if(x!=10) {…}
if(x) {…} то же что и **if(x!=0) {…}**		
==	Равно	if(x==10) {…}
if(!x) {…} то же что и **if(x==0) {…}**		
<	Меньше	if(x<10) {…}
>	Больше	if(x>10) {…}
<=	Меньше или равно	if(x<=10) {…}
>=	Больше или равно	if(x>=10) {…}
Оператор ?		

?:	Позволяет выбирать по условию.	a>b? x=a: x=b;
		или
		x = a>b? a: b;

Операторы классов и структур		
::	Разрешение области видимости	Class::Fun();
->	Косвенный доступ	Class->Fun();
.	Прямой доступ	Class.Fun();
Оператор sizeof		
	Возвращает длину в байтах переменной или типа, помещённого в скобки	x= **sizeof** a; x= **sizeof**(tp)
Операторы () []		
()	Для изменения приоритета выполнения операторов	x=(a+b)*c;
[]	Для индексации массивов	x[0]=1;

Оператор typedef

```
typedef int Miles; //Miles – это int
typedef char* Cstring;//Указатель на char
typedef void* Gen_ptr;//Обобщённый указатель
typedef int (*F)(int); /Указатель на функцию
```

Приоритеты операторов

Выс-ший	`() [] -> .`
	`! ~ + - ++ -- & * siseof(тип)`
	Префик. Адрес
	`* / % (`*арифметические*`)`
	`+ - (`*бинарные*`)`
	`<< >>`
	`< <= > >=`
	`== !=`
	`& (`*логические*`)`
	`^ (`*логические*`)`
	`| (`*логические*`)`
	`&&`
	`||`
	`?:`
	`= *= /= %= += -= &= ^= |= <<= >>=`
	`++ --  ,`

Низ-ший	*Постфик.*

Составной оператор

– ряд операторов, окружённых фигурными скобками { и }.

Операторы `if` и `if-else`

```
if(выражение)
    оператор;
```

```
if(выражение)
    оператор 1;
else
    оператор 2;
```

Оператор `while`

```
while(выражение)
    оператор;
```

Оператор `do`

```
do
    оператор;
while(выражение);
```

Оператор `for`

```
for(выражение1;выражение2;выражение3)
оператор;
```

выражение1 – поле инициализации

выражение2 – условие существования цикла

выражение3 – поле приращения

Операторы передачи управления

Оператор `break`

```
for(i=0;i<10;++i)
 {
 if(i>5) break;
 }
// break – продолжение работы здесь
```

Оператор `continue`

```
for(i=0;i<100;++i)
 {
 if(i>50) continue;
 i++;
 //continue – продолжение работы здесь
 }
```

Оператор `switch`

```
switch(выражение)
    оператор;
```

Пример:

```
switch(byte)
 {
 case 0x30: case 0x20:
  выбор ' ', '0' и '1'
 case 0x31:
  выбор '1'
  break;
 case 0x32:
  выбор '2'
  break;
 default: // по умолчанию
 }
```

Оператор `goto`

`goto` метка	

```
for(i=0;;++i)
 {
 for(j=0;j<10;++j)
  {
  if(j*i>50) goto end;
  if(i==5) break;
  }
 // переход по break
 }
end: // переход по goto
```

Потоковый ввод/вывод

`<<`	«поместить в» выходной поток (перегруженный оператор)
`>>`	«получить из» входного потока (перегруженный оператор)
`cout`	стандартный вывод (объект)
`cin`	стандартный ввод (объект)
`cerr`	стандартная ошибка (объект)
`endl`	то же, что и `'\n'` – символ новой строки (модификатор)

Для обеспечения ввода/вывода C++ полагается на внешнюю стандартную библиотеку. Информация, необходимая программе для использования этой библиотеки, размещена
в файле **`iostream.h`**

Для подключения этой и других библиотек используется директива **`include.`** Для обработки набора директив, таких как **`include`**, C++ использует препроцессор, чтобы преобразовать программу из предварительной формы в форму синтаксиса чистого C++. Эти директивы начинаются с символа #.

Пример:

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

Объявление констант и макросов с помощью команды #define для препроцессора

```
#define XXI 21
#define MEM_ERR cout<<"ошибка \
      памяти!"<<endl
```

Здесь «\» – символ переноса строки

Макрос вычисляющий абсолютное значение

```
#define ABS(X) (((X)<0)?-(X):X)
int y=-3;
cout<<ABS(y)<<endl;
```

Результат: **3**

Лекция 4

Структура программы

Программа C++ состоит из объявлений, возможно, находящихся в различных файлах. В программе объявляются константы, переменные, массивы, структуры и функции. Эти объявления могут быть расположены в разных текстовых файлах и чаще всего представляют собой библиотечные файлы, имеющие расширение «.h» или «.hpp». Текстовые файлы с расширением «.c» или «.cpp» действуют как модули и могут быть откомпилированы отдельно.

Константы и переменные представляют собой данные.

Массивы и структуры являются составными данными.

Для написания программы используются функциональные конструкции. Функция в C++ – первичная единица структурирования программы. Каждая функция находится на внешнем или глобальном уровне и не может быть объявлена вложенным способом. Одни функции объединяются в другие и в последствии используются в main(). Функция main() используется как начальная отметка для выполнения программы.

Область видимости

```
#include <iostream.h>
int a=9;// глобальная переменная
void main()
{//внешний блок
int a=2;//локальная переменная
cout<<::a<<endl;//выводит 9
 {//вход во внутренний блок
 cout<<a<<endl;//выводит 2
 int a=7;//локальная переменная
 cout<<::a<<endl;//выводит 9
 cout<<a<<endl;//выводит 7
 }//выход из внутреннего блока
cout<<a<<endl;//выводит 2
}
```

Каждый блок имеет собственную спецификацию. Имя из внешнего блока допустимо в том случае, если внутренний блок его не переопределяет. Если же оно переопределено, то имя внешнего блока скрывается или маскируется во внутреннем блоке. Внутренние блоки могут быть вложены на произвольную глубину, определённую ограничениями системы.

В C++ контекст видимости идентификатора начинается в конце его объявления и продолжается до конца самого внутреннего блока, который охватывает это объявление.

Классы памяти

auto
extern
register
static

Переменные, объявленные внутри тел функций, по умолчанию являются автоматическими (располагаются в стековой памяти)

Класс памяти register – переменные размещаются в скоростных регистрах памяти. Если регистров не хватает компилятор создаёт автоматические переменные

Класс памяти extern – переменные объявлены вне функции, память для них распределена постоянно, используются для связи между функциями, в том числе находящихся в разных файлах

Класс памяти static – переменные, сохраняющие свои значения при повторном входе в блок, используются для связи между функциями, находящихся в одном и том же файле

Пример программы на языке C++

```
//программа наибольшего общего делителя
#include <iostream.h> //подключение библиотеки ввода/вывода
int gcd(int m, int n) //объявление и описание функции
{
int r;   //объявление остатка
while(n!=0) //не равно
 {
 r=m%n; //оператор получения остатка
 m=n;   //присваивание
 n=r;
 }      //конец условного цикла
return(m) //выход из gcd со значением m
}
main() //начало программы
{
int x, y; //локальные переменные
cout<<"\nGcdProgram";
do
 { //вход в тело цикла do
 cout<<"\n Введите два числа: ";
 cin>>x>>y; //получает данные со стандартного ввода
 cout<<"\nGCD ("<<x<<","<<y<<")="<<gcd(x,y)<<endl; //помещает ответы в стандартный вывод
 }
while(x); //выход по условию x==0
}
```

Функции в C++. Объявление функции

```
// программа подающая сигнал в динамик компьютера (звонок)
#include <iostream.h> // подключение библиотеки ввода/вывода
const char BELL = '\a'; // объявление символьной константы
void ring() //объявление и описание функции ring
{
cout<<BELL; // подать сигнал в динамик компьютера
}
void main() //объявление и описание главной функции
{
ring(); // вызов функции ring
}
```

Прототипы функций

```
#include <iostream.h> // подключение библиотеки ввода/вывода
const char BELL = '\a'; // объявление символьной константы
void ring(); //прототип функции (объявление функции ring)
void main() //объявление и описание главной функции
{
ring();// вызов функции ring
}
void ring() // описание функции ring
{
cout<<BELL; // подать сигнал в динамик компьютера
}
```

Передача параметров в функции

```
// Многократное звучание звонка
#include <iostream.h> // подключение библиотеки ввода/вывода
const char BELL = '\a'; // объявление символьной константы
void ring(int k) //объявление и описание функции ring
{ // подать сигнал в динамик компьютера k раз
for(int i=0;i<k;++i) cout<<BELL;
}
void main() //объявление и описание главной функции
{
int n; // объявление целочисленной переменной
cout<<"Введите число: 1…100";
cin>>n; // консольный ввод числа повторяющихся гудков в динамике компьютера
ring(n);
}
```

Оператор return

```
// Программа нахождения минимума
#include <iostream.h> // подключение библиотеки ввода/вывода
int min(int x, int y) // объявление и описание функции min
{
if(x<y) return (x); // выбор минимального значения
else return y;
}
main()
{
int j,k,m; // объявление целочисленных переменных
cout<<"Введите 2 целых числа: ";
cin>>j>>k; // консольный ввод двух целых чисел
m=min(j,k); // вызов функции поиска минимума и запись возвращаемого результата в переменную m
cout<<m<<" является минимальным\      // перенос строки
 из "<<j<<" и "<<k<<endl;
}
```

Аргументы по умолчанию

```
// Быстрая степень (k - аргумент, значение которого по умолчанию = 2)
float powq(float a,int k=2)
{
float b=1;
while(k)
 {
 if(k%2) b*=a; k/=2; a*=a;
 }
return b;
}
// где-то в программе:
...

cout<<powq(3); // Возведение в квадрат
cout<<powq(3,3); // Возведение в куб
...
```

Встраивание

```
inline double cube(double x)
{
return (x*x*x);
}

cout<<cube(z)<<endl; // здесь вызова функции нет,
// есть просто перемножение z*z*z, то есть строчка равносильно:
cout<<z*z*z<<endl;
```

Перегружаемые функции

Все три функции рассмотренные ниже объявлены в одной программе

```
// 1 функция
double powq(double a,int k)
{
float b=1;
while(k)
 {
 if(k%2) b*=a; k/=2; a*=a;
 }
return b;
}
// 2 функция
double powq(double a)
{
return a*a;
}
// 3 функция
double powq(float a)
{
return a*a;
}

void main()
{
double x=1.2;
float z=1.1;
cout<<powq(x,3)<<endl; // вызов 1-й функции x^3
cout<<powq(x)<<endl; // вызов 2-й функции x^2
cout<<powq(z)<<endl; // вызов 3-й функции x^2
}
```

Перегружаемые функции различаются по типу и/или по количеству аргументов

Указатели

Указатели используются в программах для организации доступа к памяти и манипуляции адресами.

Если `v` – переменная, тогда `&v` – адрес или местоположение сохранённого значения в памяти.

`int p;` – объявление переменной типа `int`;

`int* p;` или

`int *p;` – объявление указателя на тип `int`;

Если `v` – указатель, тогда `*v` –значение в памяти по адресу содержащемуся в указателе `v` (операция разыменование).

```
int x=12;
int y=3;
int* p;
p=&x;
y=*p;
cout<<p<<endl;      // выдаёт адрес переменной x
cout<<*p<<endl;     // выдаёт значение 12
cout<<y<<endl;      // выдаёт значение 12
```

Нулевой адрес или запись в указатель пустого значения:

```
p=NULL;
p=0;
```

Операции с указателями

```
int* p=4;
p+=2
cout<<p<<endl; // результат 8 (для 16-битных систем)

double* t=40;
t--
cout<<t<<endl; // результат 32
```

Объявление ссылок и вызов по ссылке

```
int n;
int& nn=n; // nn – альтернативное имя для n
double a[10];
double& last=a[9]; // last – псевдоним для a[9]
---------------------------
Сортировка в порядке убывания
В аргументе функции используются ссылки
void order(int& q, int& p) // сортировка в порядке убывания
{
int temp;
if(p>q) temp=p, p=q, q=temp;
}

void main()
{
int i=3, j=7;
cout<<"До вызова order: "<<endl;
cout<<"i="<<i<<"\tj="<<j<<endl;
order(i,j);
cout<<"После вызова order:"<<endl;
cout<<"i="<<i<<"\tj="<<j<<endl;
}
---------------------------
или через указатели:
void order(int* q, int* p)
{
int temp;
if(*p>*q) temp=*p, *p=*q, *q=temp;
}
```

Строки

```
// Вычисление длины строки
size_t strlen(const char* s)
{
int i;
for(i=0;s[i];++i);
return i;
}
```

```
// Сравнение строк. (равно 0 если одинаковы)
int strcmp(const char* s1, const char* s2)
{
int i;
for(i=0; s1[i]&&s2[i]&&(s1[i]== s2[i]);++i);
return (s1[i]-s2[i]);
}
// Копирование строки s2 в s1
int strcpy(char* s1,const char* s2)
{
int i;
for(i=0; s1[i]=s2[i];++i);
return s1;
}
```

Автоматические массивы

(размещаются в стековой памяти)

```
int a[100]; // одномерный массив
int b[3][5]; // двухмерный массив
int c[2][3][4]; // трёхмерный массив
```

Инициализация массивов

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

или

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

Инициализация символьного массива

```
char name[3][9]={"Laura","Michele","Pohl"};
```

Операторы new и delete

Для управления свободной памятью используют операторы выделения свободной памяти new и освобождения памяти delete

```
int* p; // указатель на тип int
new_type* a; // указатель на тип new_type
int size=5;
p=new int[size]; // динамический массив
a=new new_type; // новый объект
...
delete [] p; // освобождение памяти, занятой динамическим массивом
delete a; // освобождение памяти, занятой объектом
```

Лекция 5

Реализация абстрактных типов данных

Абстрактный тип данных (АТД) – тип данных, который определяется пользователем. Большая часть объектно-ориентированного процесса проектирования заключается в разработке АТД, которые позволяют наиболее эффективно решать поставленную задачу.

Структуры

Тип структуры позволяет программисту объединить компоненты в переменную с одним именем. Структура содержит индивидуально именованные компоненты, называемые *членами* или *элементами* (structure members).

Поскольку члены структуры бывают различных типов, программист может создавать агрегаты (данных), позволяющие описывать сложные данные.

ОПРЕДЕЛЕНИЕ СТРУКТУРНЫХ ПЕРЕМЕННЫХ

Слово «структура» используется двояко:

1) в смысле «структурного шаблона»

2) в создании «структурной переменной»;

Шаблон является схемой без содержания; он сообщает компилятору, как делать что-либо, но не вызывает никаких действий в программе.

Примеры:

Установка структурного шаблона, описывающего комплексную переменную

```
struct complex
{
double real;
double image;
};
```

Объявление и инициализация комплексной переменной

```
complex x;
x.real=1;
x.image=2;
```

Вывод значения комплексного сигнала в консольную строку:

```
cout<<"("<<x.real<<", "<<x.image<<")"<<endl;
```

Установка структурного шаблона men

```
struct men
{
char name[15];  // имя
char surname[20];  // фамилия
unsigned year_of_birth;  // год рождения
};
void main()
{
men group[25];
int n=5;
strcpy(group[n].name,"Иван");
strcpy(group[n].surname,"Петров");
group[n].year_of_birth=1980;
cout<<"Имя: "<<group[n].name<<endl;
cout<<"Фамилия: "<<group[n].surname<<endl;
cout<<"Год рождения: "<<group[n].year_of_birth<<endl;
}
/* инвентаризация книги */
#include <stdio.h>
#define MAXTIT 41 /*макс.длина названия +1 */
# define MAXAUT 31 /*макс.длина фамилии автора+1*/
struct book
{
char title [MAXTIT]; /* символьный массив для названия */
char author [MAXAUT]; /* символьный массив для фамилии автора */
float value; /* переменная для хранения цены книги */
}; /*конец шаблона структуры */
void main()
{
struct book libry; /* описание переменной типа book */
printf("Введите название книги:\n");
gets(libry.title); /*доступ к элементу title */
printf("Введите фамилию автора:\n");
gets(libry.author);
printf("Введите цену:\n");
scanf("%f",&libry.value);
printf("%s:\"%s\"\(%p.2f\)\n", libry.author, libry.title, libry.value);
}
```

Образец работы программы:

```
Введите название книги:
Искусство программирования
Введите фамилию автора:
И. Пол
Введите цену:
5.67
И. Пол: "Искусство программирования" (5р.67к.)
```

Объектно-ориентированное программирование (ООП)

ООП – взгляд на программирование, сосредоточенный на данных. Данные и их поведение жёстко связаны. Они представлены в виде классов, экземпляры, которых – объекты. *Объекты* – переменные класса.

Для удобства создания нового типа из уже существующих типов, определённых пользователем ООП используется механизм *наследования*.

Класс – обеспечивает механизм инкапсуляции для реализации АТД.

Инкапсуляция включает как детали внутренней реализации специфического типа, так и доступные извне операции и функции, которые могут оперировать объектами этого типа. Детали реализации могут быть скрыты от пользователя, который использует тип.

Класс – расширение идеи относительно *struct*, основанной в С. Это способ инкапсуляции типов данных и связанных функций. Концепция *class* – это, фактически, *struct* с видимостью private, но умолчанию.

Сокрытие данных позволяет создавать легче отлаживаемый и сопровождаемый код, потому что ошибки и модификации локализованы в нём. Программы клиента должны знать только спецификацию интерфейса типа.

Термин ООП включает в себя следующие концепции:

а) моделирование деятельности мира;
б) наличие типов, определяемых пользователем;
в) сокрытие деталей реализации;
г) повторное использование кода через наследование;
д) разрешение интерпретации вызова функции во время выполнения.

Советы от Б. Страуструпа по созданию классов:

а) если вы считаете «это» отдельным понятием, сделайте его классом;
б) если вы считаете «это» отдельным объектом, сделайте его объектом некоторого класса;
в) если два класса имеют общим нечто существенное, сделайте «это» базовым классом.

СОКРЫТИЕ ДАННЫХ И ФУНКЦИИ-ЧЛЕНЫ

Пользователь АТД – *клиент*

Поставщик АТД – *изготовитель*

Сокрытие подробностей реализации — в интересах изготовителя:

– меньше объяснений клиенту об АТД;

– внутренние усовершенствования не касаются вопросов использования АТД клиентом;

– снижается риск небрежного использования со стороны клиента.

Объявление функции включается в объявление структуры и вызывается с использованием методов доступа к членам структуры.

Функциональные возможности, типа данных struct, должны непосредственно включаться в объявление структуры.

В терминологии ООП функция-член – это *метод.*

Видимость *private*, *protected* и *public*.

Концепция struct расширяется в C++ для того, чтобы позволить функциям иметь общие, частные и защищенные члены.

Для того, чтобы избежать вмешательства во внутренние детали реализации АТД, ООП использует механизм сокрытия данных. В C++ оно обеспечивается ключевым словом *class* и ключевыми словами, связанными с правами доступа: *public*, *private*, и *protected*.

Ключевое слово *protected*: его реальное использование связывается с наследованием.

Внутри *struct* использование ключевого слова *private*, сопровождаемого двоеточием, ограничивает доступ к членам, которые следуют за этой конструкцией. Члены private могут использоваться только несколькими категориями функций, в привилегии которых входит доступ к этим членам, например функции члены struct.

Удобно считать private часть используемой только разработчиком, а public часть – спецификацией интерфейса, которую может использовать клиент. Позже разработчик может изменять private часть, не влияя при этом на правильность использования клиентом рассматриваемого типа.

По правилам хорошего тона члены-данные должны помещаться в private часть структуры, и доступ к ним должен осуществляется с использованием функций-членов. Такая дисциплина доступа гарантирует, что пользователь не сможет вмешаться или неправильно использовать реализованный АТД.

ОБЛАСТЬ ВИДИМОСТИ КЛАССА

Одно из предназначений классов – обеспечение технологии инкапсуляции, то есть чтобы все имена, объявленные внутри класса, обрабатывались внутри их собственного пространства имен, отлично от любых внешних имен, имен функций или имен прочих классов. Это создает потребность в операторе разрешения контекста.

ОПЕРАТОР РАЗРЕШЕНИЯ КОНТЕКСТА ::

– является оператором с самым высоким приоритетом в языке. Применяется в двух формах:

1) `::i` *одноместный оператор – указывает на внешнюю область видимости*, используется для раскрытия или обращения к имени, относящемуся ко внешнему контексту и скрытому локальным контекстом или контекстом класса.

2) `name::i` *двухместный оператор – указывает на область видимости класса*, используется для устранения неоднозначности имён, которые повторно используются в пределах класса, его применение существенно при наследовании.

Объявление структуры и класса для описания комплексных чисел

Использование `struct`:

```
struct complex // начало описания структуры
{
void assign(double r, double i);
void print() // вывод в консоль, inline-функция
 {
 cout<<real<<"+"<<imag<<"i";
 }
private:
double real, imag; // скрытые данные
}; // конец описания структуры

void complex::assign(double r, double i=0.0)
{
real=r ; imag=i;
}
```

Использование `class`

```
class complex // начало описания класса
{ // по умолчанию private
double real, imag; // скрытые данные
public:
void assign(double r, double i);
void print() // вывод в консоль, inline-функция
 {
 cout<<real<<"+"<<imag<<"i";
 }
}; // конец описания класса

void complex::assign(double r, double i=0.0)
{
real=r; imag=i;
}
```

ШАБЛОНЫ

Для обеспечения параметрического полиморфизма в C++ используется ключевое слово *template*. С помощью параметрического полиморфизма можно применить один и тот же код к разным типам, причём тип является параметром тела кода.

ШАБЛОНЫ ФУНКЦИЙ

Пример копирования массива

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

С помощью макроса:

```
#define COPY(A,B,N){int i; for\ // пример переноса на другую строку описания макроса
(i=0;i<(N);++i)(A)[i]=(B)[i];} // не безопасно для типов
```

С помощью шаблона:

```
template<class TYPE> void copy(TYPE a[],TYPE b[],int n)
{
for(int i=0;i<n;++i)
 a[i]=b[i]; //a и b должны быть одного типа
}
```

Обобщённая структура копирования

```
template<class T1, class T2> void copy(T1 a[],T2 b[],int n)
{
for(int i=0;i<n;++i)
 a[i]=b[i];
}
```

Алгоритм вызова перегруженной функции

1. Строгое соответствие для нешаблонных функций
2. Строгое соответствие для шаблонов функций
3. Обычное разрешение аргументов для нешаблонных функций

ДРУЖЕСТВЕННЫЕ ФУНКЦИИ

Ключевое слово *friend* (друг) служит спецификатором, уточняющим свойства функции. Оно дает функции-не-члену доступ к скрытым членам класса и предоставляет способ для обхода ограничений сокрытия данных в C++. Однако должна быть веская причина для обхода этих ограничений, поскольку они важны для надежного программирования.

Причины:

– некоторые функции нуждаются в привилегированном доступе к более чем одному классу;

– дружественные функции передают все свои аргументы через список аргументов, при этом каждое значение аргумента должно допускать автоматическое преобразование.

Дружественная функция должна быть объявлена внутри объявления класса, по отношению к которому она является дружественной. Функция предваряется ключевым словом friend и может встречаться в любой части класса; это не влияет на ее смысл.

Функция-член одного класса может быть дружественной другому классу.

Структура compl<Type> – комплексные числа

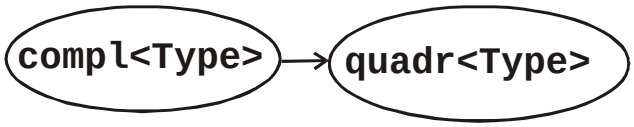
```
#ifndef MATH_H // если не определён макрос-константа MATH_H
 #include <math.h> // подключить математическую библиотеку
#endif // завершение команды #ifndef
template<class T> struct compl // начало описания структуры
{ // данные структуры:
T R; // real – вещественная часть комплексного числа
T I; // image – мнимая часть комплексного числа
compl() { R=I=0; }  // Конструктор по умолчанию
compl(T r) { R=r; I=0; }  // Конструктор с одним аргументом
compl(T r,T i) { R=r,I=i; }  // Конструктор с двумя аргументами
compl(const compl<T>& z) { R=z.R,I=z.I; } // Конструктор копирования
~compl() { } // Деструктор
T& x() { return R; }  // Доступ к элементу R (real) структуры
T& y() { return I; }  // Доступ к элементу I (image) структуры
compl<T>& operator=(const compl<T>& z) // Оператор присваивания
 { if(this!=&z) R=z.R,I=z.I; // если z – не сам объект
 return (*this);
 }
                    // Комплексное сопряжение
compl<T> conj() { compl<T> temp(R,-I); return(temp); }
void operator()(T r,T i) { R=r,I=i; } // Оператор ()
T mod2() { return (R*R+I*I); }  // Квадрат модуля комплексного числа
void print() { cout<<R<<"+j*("<<I<<")";} // Вывод в консоль
friend compl<T> operator-(compl<T> c1,compl<T> c2) // Вычитание
 {
 compl<T> temp(c1.R-c2.R,c1.I-c2.I); return(temp);
 }
friend compl<T> operator*(compl<T> c1,compl<T> c2) // Комплексное умножение
 {
 compl<T> temp(c1.R*c2.R-c1.I*c2.I,c1.R*c2.I+c1.I*c2.R);
 return(temp);
 }
void write_bin(ostream& out); //Запись в бинарный файл (Объявл.функции)
void read_bin(istream& in); //Чтение из бинарного файла(Объявл.функции)
}; // конец описания структуры
template<class T> void compl<T>::write_bin(ostream& out)
{ //Запись в бинарный файл (Описание функции)
int s=sizeof(T); out.write((char*)&R,s); out.write((char*)&I,s);
}
template<class T> void compl<T>::read_bin(istream& in)
{ //Чтение из бинарного файла (Описание функции)
int s=sizeof(T); in.read((char*)&R,s); in.read((char*)&I,s);
}
```

Структура quadr<Type> – квадратурные числа

```
// public compl<T> - наследование свойств от класса compl<T>
template<class T> struct quadr: public compl<T>
{ // начало описания структуры
quadr(T i) { R=i; I=i; } // Конструктор с одним параметром
quadr(quadr<T>& z) { R=z.R; I=z.I; } // Конструктор копирования
~quadr() {} // Деструктор
friend quadr<T> operator*(quadr<T> c1,quadr<T> c2)
 {
 quadr<T> temp(c1.X*c2.X,c1.Y*c2.Y); return(temp);
 }
// Деление квадратурных чисел
friend quadr<T> operator/(quadr<T> c1,quadr<T> c2)
 {
 quadr<T> temp(c1.X/c2.X, c1.Y/c2.Y); return(temp);
 }
}; // конец описания структуры
```

Схема одиночного наследования для класса квадратурных чисел

Одиночное наследование свойств базового класса (структуры) **compl<Type>**



Лекция 6

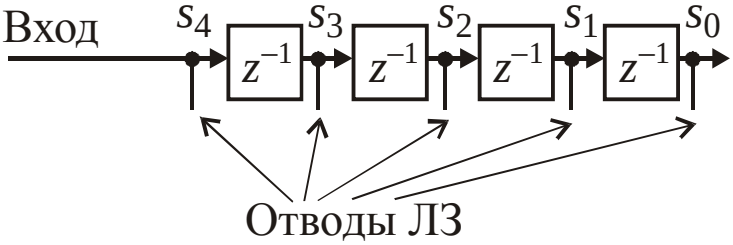
Класс одномерного динамического массива с шаблонной (template) подстановкой vect<Type>

```
// Описание класса vect
#define M_Err if(!p) cerr<<"Свободная память исчерпана\n",exit(1);
#define Size_Err if(n<=0) cerr<<"Неправильный\
                                размер массива"<<n<<endl,exit(1);
#define Index_Err if(i<0||i>up()) cerr<<"Неправильный\
                                      индекс "<<n<<endl, exit(1);
template <class T> class vect
{ // Скрытые данные:
 T* p; // указатель на массив
 int size; // размер массива
public:
 vect();// Конструктор по умолчанию
 vect(int n); // Конструктор с одним параметром на входе,
              // параметр характеризует размер массива
 vect(int n,T a); // Конструктор с двумя параметром на входе,
                  // 2-й параметр – число для инициализации
 vect(const vect& v); // Конструктор копирования
 ~vect(){ delete [] p; p=NULL; } // Деструктор
 int up() const { return(size-1); } // Возвращает индекс последнего элемента массива
 int len() const { return(size); } // Возвращает размер массива
 T& operator[](int i); // доступ к элементу массива
 vect& operator=(const vect& v); // Оператор присваивания
```

```
 void print(); // Вывод в консоль
};
// описание внешних функций класса vect<T>
template <class T> vect<T>::vect() // Конструктор по умолчанию (описание)
{
p=new T[(size=80)]; M_Err
}

template <class T> vect<T>::vect(int n) // Конструктор с 1 параметром на входе (описание)
{
Size_Err
p=new T[(size=n)]; M_Err
for(int i=0;i<size;)
 p[i++]=NULL;
}

template <class T> vect<T>::vect(int n,T a) // Конструктор с указанием границы и
                                            //инициализацией (описание)
{
Size_Err
p=new T[(size=n)]; M_Err
for(int i=0;i<size;p[i++]=a);
}

template <class T> vect<T>::vect(const vect<T>& v) // Конструктор копирования (описание)
{ size=v.size;
p=new T[size]; M_Err
for(int i=0;i<size;++i)
 p[i]=v.p[i];
}

template <class T> T& vect<T>::operator[](int i) // Взять значение элемента массива
{
Index_Err return(p[i]);
}

template <class T> vect<T>& vect<T>::operator=(const vect<T>& v)
{ // Операция присвоения (описание)
int s=(size<v.size)?size:v.size;
if(v.size!=size)
 cerr<<"Произошло копирование массивов различных размеров "
 <<size<<" и "<<v.size<<endl;
for(int i=0;i<s;++i)
 p[i]=v.p[i];
return(*this);
}

template <class T> void vect<T>::print() // Вывод в консоль (описание)
{
for(register i=0;i<size;cout<<p[i++]<<'\t'); cout<<endl;
}
```

Модель линии задержки (ЛЗ)

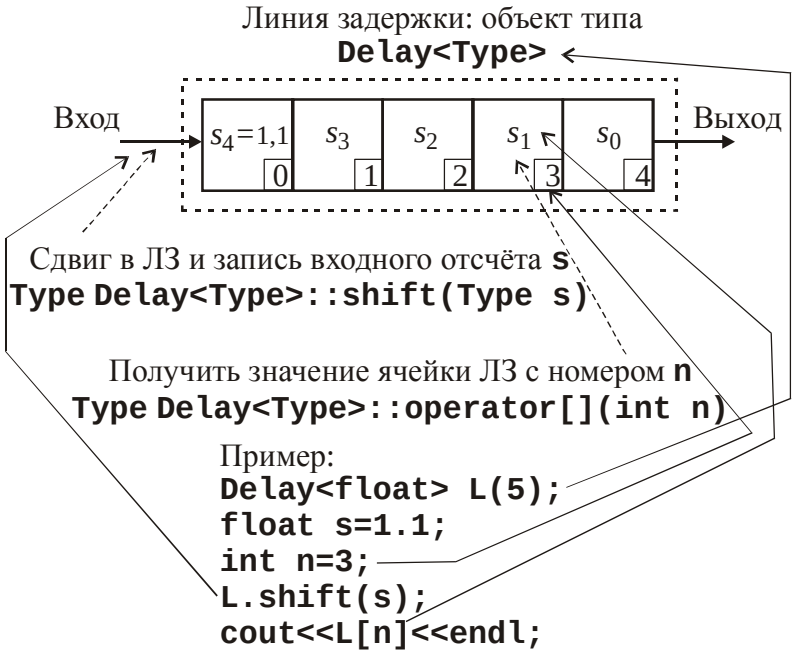
Представление ЛЗ в виде последовательности элементов задержки



Сигнал наблюдается в точках (отводах) между элементами задекржки.

Представление ЛЗ в виде сдвигового регистра

Здесь возможен просмотр содержимого ячеек ЛЗ



Упрощённый класс линии задержки (ЛЗ)

```
// Описание класса ЛЗ
class Delay
{
float* p; // указатель на массив ячеек линии задержки
int size; // размер линии задержки
public:
Delay(int n); // Конструктор, n - размер ЛЗ
float shift(float s); // Сдвиг и запись входного отсчёта сигнала s в ЛЗ
T operator[](int i); // Доступ к ячейке ЛЗ
~Delay() { delete [] p; } // Деструктор
print(); // вывод содержимого ЛЗ в консоль
};
```

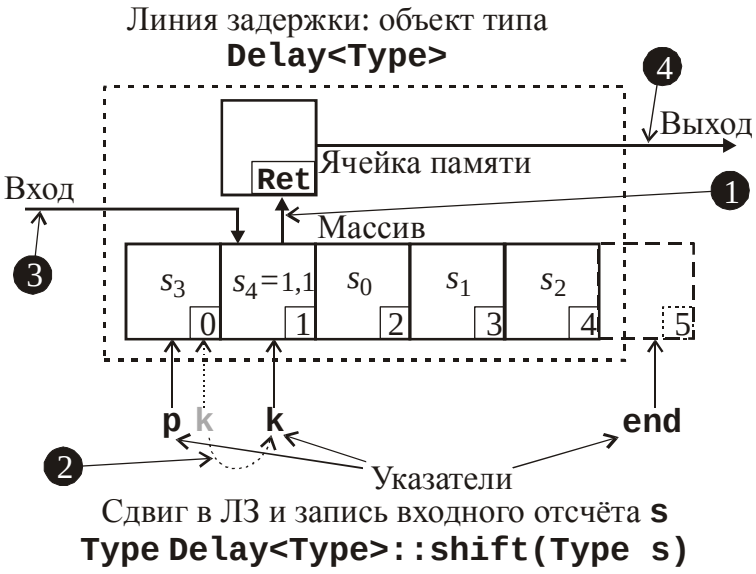
```
Delay::Delay(int n) // Конструктор, n – размер ЛЗ (описание)
 {
 p=new float[size=n];
 for(i=0;i<size;++i)
  p[i]=0; // Обнулить каждую ячейку линии задержки
 }

// Сдвиг и запись входного отсчёта сигнала s в ЛЗ (описание)
float Delay::shift(float s)
 {
 float r=p[size-1]; // сохранить содержимое последней ячейки
 for(i=size-1;i>0;--i)  // цикл по ячейкам линии задержки
  p[i]=p[i-1]; // перезапись для реализации сдвига
 p[0]=s; // записать в 0-ю ячейку входное значение
 return r; // вернуть результат выходного значения линии задержки
 }
```

Класс линии задержки (ЛЗ) с использованием наследования

```
// Описание класса ЛЗ
template<class T> class Delay: public vect<T>
{
public:
Delay(int n):vect<T>(n) { } // конструктор
~Delay() { } // деструктор
T shift(T s);
T operator[](int i) // доступ к ячейке линии задержки
 {
 return vect<T>::operator[](i);
 }
void print(); // просмотр содержимого линии задержки
};

// Сдвиг и запись входного отсчёта сигнала s в ЛЗ
T Delay<T>::shift(T s)
 {
 T r;
 r=vect<T>::operator[](vect<T>::up());
 for(i=vect<T>::up();i>0;--i)
  vect<T>::operator[](i)=
  vect<T>::operator[](i-1);
 vect<T>::operator[](0)=s;
 return r;
 }
```

Лекция 7

Класс линии задержки (ЛЗ)с шаблонной (template) подстановкой Delay<Type> оптимальный по быстродействию

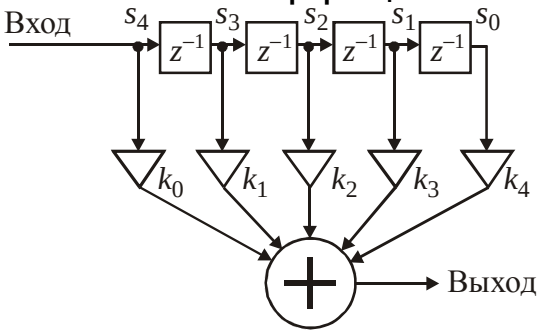



```
// Описание класса Delay
template<class T> class Delay
{
protected:
 T *p; //указатель на начало  массива
 T *k, //указатель на входную  ячейку
 T *end;//указатель на ячейку памяти, следующей, после  последней в ЛЗ
int Size;  // размер ЛЗ
public:
Delay(int n); //конструктор ЛЗ
~Delay() //деструктор ЛЗ
  {
  if(p)
   delete p;
  k=NULL; Size=0;
  }
 T shift(T s); // сдвиг в ЛЗ
 int len() { return Size; }  // Получить размер ЛЗ
 int size() { return Size; }  // Получить размер ЛЗ
 void print();// Вывести содержимое ЛЗ  в консольную строку
 T operator[](int n); // Доступ на чтение к элементу ЛЗ
};
```

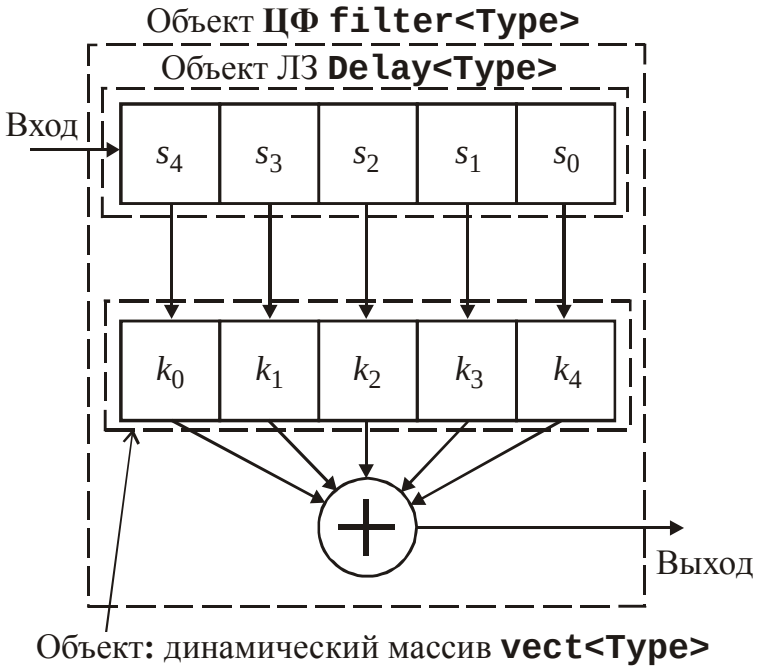
```
// Описание внешних функций класса Delay<T>

template<class T> Delay<T>::Delay(int n) // Конструктор ЛЗ (описание)
{
if(n<1)
 cerr<<"Размер неверный "<<endl,exit(1);
p=new T[Size=n];
if(p==NULL)
 cerr<<"Не достаточно памяти"<<endl,exit(1);
k=p;
end=p+Size;
for(register i=0;i<Size;i++) p[i]=NULL;
}

template<class T> T Delay<T>::shift(T z) // Сдвиг в ЛЗ (описание)
{
if(++k>=end) k=p;
T Ret=*k;
*k=z;
return Ret;
}

template<class T> T Delay<T>::operator[](int n)
{ // Доступ на чтение к элементу ЛЗ (описание)
if(n>=Size||n<0)
 cerr<<"Ошибка: неверный индекс "<<n<<endl,exit(1);
if(k-n<p||k-n>end)
 return *(k-n+Size);
else
 return *(k-n);
}
```

Модель цифрового фильтра (ЦФ)

Схема цифрового нерекурсивного фильтра с элементами задержки и весовыми коэффициентами

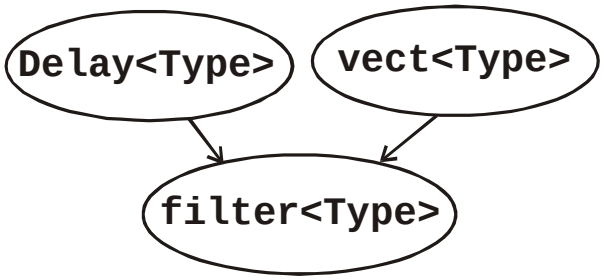


Программная модель нерекурсивного фильтра



Наследование

Множественное наследование свойств базовых объектов **`Delay<Type>`** и **`vect<Type>`**



Согласованный фильтр

Комплексное сопряжение коэффициентов

Ван Трис Г. Теория обнаружения, оценок и модуляции. Том III. Обработка сигналов в радио- и гидролокации и приём случайных гауссовых сигналов на фоне помех. Нью-Йорк, 1971. Пер. с англ. Под ред. проф. В.Т. Горяинова. М., «Сов. Радио», 1977, 664 с. [Стр. 274]

Стейн Джонс. Принципы современной теории связи и их применение [Стр. 252]

Класс цифрового фильтра с шаблонной (template) подстановкой и с использованием множественного наследования

```
#include<vect.h>
#include<delay.h>
// Описание класса filter: наследуются свойства классов Delay<Type> и vect<Type>
template<class T> class filter: public Delay<T>, vect<T>
{
public:
// Происходит вызов конструкторов ЛЗ и динамического массива
filter(vect<T>& n): Delay<T>(n.len()), vect<T>(n.len()) {}

~filter() { } // деструктор

T operator[](int i) // Доступ к ячейкам ЛЗ
 {
 return Delay<T>::operator[](i);
 }

T operator()(int i) // Доступ к весовым коэффициентам
 {
 return vect<T>::operator[](i);
 }

T filtr();// Процесс фильтрации

void print();// Вывод содержимого ЦФ в консольную строку
}

// Фильтрация: результат процесса сложения произведений ячеек ЛЗ ЦФ на весовые коэффициенты ЦФ
template<class T> T filter<T>::filtr()
{
T sum(0); // Обнуление сумматора
for(int i=0;i<vect<T>::len();++i ) // Процесс накопления произведений
 sum+=Delay<T>::operator[](i)*vect<T>::operator[](i)
return sum; // Вернуть результат суммирования
}

template<class T> void filter<T>::print()
{ // Вывод содержимого фильтра в консольную строку
Delay<T>::print(); // Вывод содержимого ЛЗ ЦФ
vect<T>::print(); // Вывод значений весовых коэффициентов
}
```

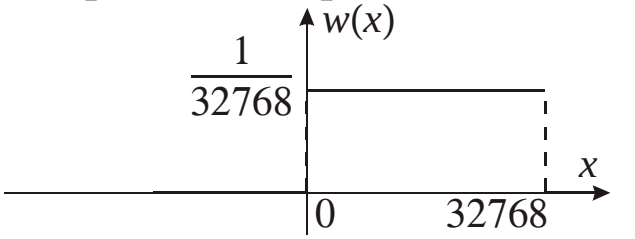
Функция, моделирующая гауссовский шум

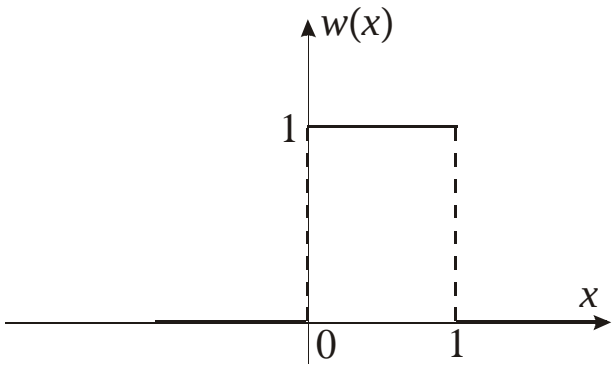
Реализация шума на основе центральной предельной теоремы с ограничением суммы

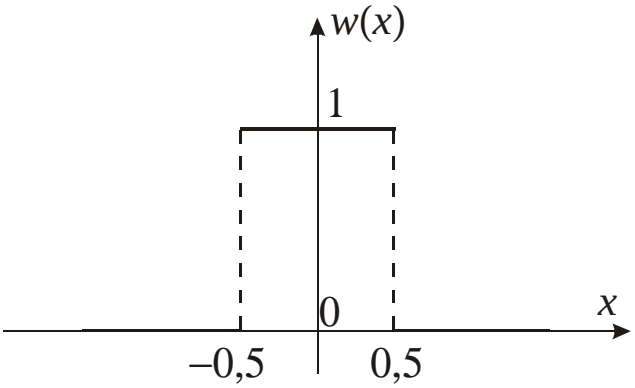
$$\zeta = \sum_{i=1}^{12} \xi_i$$

$$D_\zeta(x) = 1$$

Распределение вероятностей







$$\xi_i = (-0,5 \ldots 0,5)$$

$$D_{\xi_i}(x) = \int_{-\infty}^{\infty} w(x)\, x^2 dx =$$

$$= \int_{-0,5}^{0,5} 1 \cdot x^2 dx = \frac{x^3}{3}\Bigg|_{-0,5}^{0,5} =$$

$$= \frac{1}{3}\left(\frac{1}{8} + \frac{1}{8}\right) = \frac{1}{12}$$

// Неоптимальная реализация

```
float noise(float s)
{
int i;
float n=0;
for(i=0;i<12;++i)
 n+=rand()/32768.-0.5;
return s*n;
}
```

// Оптимально по скорости

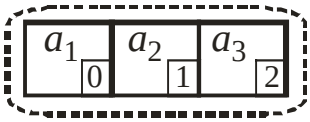
```
#define R x+=rand();
float noise(float s)
{
long x=-196608L;
R R R R R R R R R R R R
return s*x/32768;
}
```

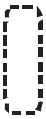
Лекция 8

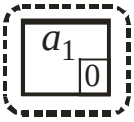
Класс цифрового фильтра с любым числом ветвей, с шаблонной (template) подстановкой и с использованием множественного наследования

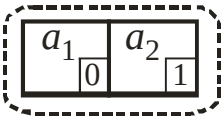
```
#include<vect.h>
#include<delay.h>
template<class T> class filter: public delay<T>
{
 vect<vect<T>*>* p; // Указатель на динамический массив указателей на динамический массив
 int Number;   // Число ветвей
public:
 filter(vect<T>& n); // конструктор с 1 параметром
 ~filter() { remove(); } // деструктор
 int getLen()  // получить размер линии задержки в фильтре
  { return delay<T>::max(); }
 int getNumber() { return Number; } // получить число ветвей фильтра
 void remove(); //
  { if(Number==1) delete [] p; }
 void print();
 T filtr(int n); // получить результат фильтрации n-й ветви
 T filtr() { return filtr(0); } // получить результат фильтрации 0-й ветви
};

template<class T> T filter<T>::filtr(int n) // получить результат фильтрации n-й ветви
{
if(n<0||n>=Number) n=0; // проверка наличия n-й ветви, если неправильный n, то 0-я ветвь
T sum(0); // Обнуление сумматора
for(int i=0;i<=(*p)[n]->up();i++) // суммирование произведений ячеек линии задержки
 sum+=(*(*p)[n])[i]*delay<T>      // на соответствующие им весовые коэффициенты n-й ветви
  ::operator[]((*Begin)[n]
  +i*(*Step)[n]);
return sum; // вернуть результат фильтрации n-й ветви
}

template<class T> int maxDelay(vect<vect<T>*,>& n)
{ // определить максимальный размер массива среди массивов коэффициентов
for(int Max=1,i=0;i<=n.up();i++)  // Цикл по всем ветвям
 if(!n[i]) return 0; // Вернуть код ошибки если найдена ветвь без весовых коэффициентов
 else
  if(n[i]->up()+1>Max) // В Max записать максимальный размер среди массивов с весовыми
   Max=n[i]->up()+1;   // коэффициентами
return Max; // Вернуть максимальный размер массива среди массивов коэффициентов
}

template<class T> filter<T>::filter(vect<T>& n):delay<T>(n.up()+1)
{ // конструктор фильтра по заданному динамическому массиву с коэффициентами
Number=1;
p=new vect<vect<T>*,>[Number];
(*p)[0]=&n; // адрес массива коэффициентов записать в 0-й элемент массива
}

template<class T> void filter<T>::print() // просмотреть содержимое данных в фильтре
{
delay<T>::print(); // просмотр содержимого линии задержки
for(int i=0;i<=(*p)[0]->up();i++)
 cout<<(*(*p)[0])[i]<<'\t'; // просмотр весовых коэффициентов для каждой ветви
}
```

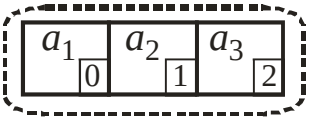
Класс «Очередь» – Queue

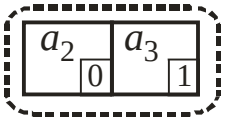
Начало очереди  Конец очереди

 – Очередь пуста

 – В очередь добавлен 1 элемент (a_1)

 – В очередь добавлен ещё 1 элемент (a_2)

 – В очередь добавлен ещё 1 элемент (a_3)

 – Из очереди удалён 1 элемент (a_1)

Элементы очереди образуют динамический массив. Число элементов массива увеличивается на 1 при вставке элемента в очередь, и уменьшается на 1 при удалении элемента из очереди.

Этот класс может быть использован, например, для моделирования алгоритма Витерби. При моделировании алгоритма Витербм самым сложным этапом является построение решётки Витерби. Класс «Очередь» позволяет эффективно решать эту задачу.

Так как для вставки и удаления используется динамическое управление памятью с помощью операций new и delete, то использовать данный объект в высокоскоростных программах не рекомендуется.

```
#define MaxQueue 100 // максимальное количество элементов в очереди
template<class T> class Queue
{
protected:
 int Len; // Длина очереди
 T* P; // Указатель на массив элементов T
public:
 Queue():Len(0),P(NULL) {}  // конструктор по умолчанию
 Queue(Queue<T>* a);        // конструктор копирования
 ~Queue() {if(P) delete [] P; P=NULL;} // деструктор
 int Delete();              // удаление первого элемента в очереди
 void Destroy();            // удаление всех элементов в очереди
 Queue<T>* object() { return this; } // вернуть адрес объекта
 int len() { return Len; }  // вернуть размер очереди
 T operator[](int i);       // вернуть элемент с индексом i из очереди
 T first() { return P[0]; } // вернуть первый элемент из очереди
 int Insert(T elem);        // постановка элемента в очередь
 void print();              // вывод содержимого очереди
};
```

```
// Описание конструктора копирования
template<class T> Queue<T>::Queue(Queue<T>* a)
{
if(a!=NULL) //Если очередь не пуста
 {
 Len=a->Len; // Копировать длину очереди (число элементов в очереди)
 P=new T[Len]; // Создать массив для хранения элементов очереди
 if(!P) // Если возвращён нулевой указатель, то
  cerr<<"Не хватило памяти!\n",exit(1); // Закончилась свободная память
 for(int i=0;i<Len;++i)  // Цикл по всем элементам очереди
  P[i]=a->P[i]; // Копировать элементы очереди
 }
else
 Len=0,P=NULL;
}

// Описание функции постановки элемента в очередь
template<class T>
int Queue<T>::Insert(T elem)
{
if(++Len>MaxQueue) // Увеличить размер очереди на 1. Если превышен лимит очереди
                   // (максимально возможное количество элементов в очереди),то
 {
 --Len; // Вернуть прежний размер очереди, уменьшив его на 1
 return 1;  // очередь переполнена! Код ошибки 1.
 }
T* temp=new T[Len]; // Создаём новый массив для очереди
if(!temp) // Если в указателе NULL
 {
 --Len; // Уменьшить размер очереди на 1
 return 2; // Для нового элемента не хватило памяти! Код ошибки 2.
 }
for(int i=0;i<Len-1;i++) // Цикл по всем элементам в очереди
 temp[i]=P[i]; // Копировать элементы очереди из старого массива в новый
if(P) // Если указатель на старый массив не равен нулю, то
 delete [] P; // удалить старый массив
P=temp; // переписать в указатель p новый адрес массива данных очереди
P[i]=elem; // записать новый элемент в конец очереди
return 0; Вернуть код 0 – успешное завершение функции постановки элемента в очередь
}

// Описание функции удаления первого элемента в очереди
template<class T> int Queue<T>::Delete()
{
if(Len<=0) // если длина очереди < 1, то
 return 1; // очереди нет! Вернуть код ошибки 1.
T* temp=new T[--Len]; // создать новый массив размером меньше на 1
if(!temp) // если не удалость создать массив (возврат значения NULL)
 exit(1); // аварийный выход из-за нехватки памяти
for(int i=0;i<Len;i++) // Цикл по всем элементам очереди
 temp[i]=P[i+1]; // Копируем данные из старого массива в новый
if(P) // если указатель не равен NULL, то
 delete [] P; // удалить старый массив
P=temp; // переписать в указатель p новый адрес массива данных очереди
return 0; // Вернуть код 0 – успешное завершение функции удаления элемента из очереди
}

// Описание функции удаления всех элементов в очереди
template<class T> void Queue<T>::Destroy()
{
while(Len>0) // Выполнять цикл до тех пор, пока длина очереди ненулевая
 Delete(); // Удалить очередной элемент из очереди
}
```

```
// Доступ к ячейке очереди
template<class T> T Queue<T>::operator[](int i)
{
if(i<0||i>=Len) // Если индекс лежит вне диапазона, то
 cerr<<"Ошибка неверный индекс Queue\n",exit(1); // Вывести сообщение о неверном индексе
return P[i]; // Вернуть значение элемента очереди с индексом i
}

// Описание функции вывода содержимого очереди
template<class T> void Queue<T>::print()
{
cout<<"___Queue элемент - Begin:\n"; // Сообщение о начале вывода содержимого очереди
cout<<"this="<<this<<"\nLen="<<Len<<"\nP="<<P<<endl;
if(P>0&&Len>0) // если очередь не пуста
 for(int i=0;i<Len;++i) // цикл по элементам очереди
  cout<<P[i]<<'\t'; // вывести i-й элемент очереди
cout<<"\n~~~Queue элемент - End:\n"; // Сообщение о конце вывода содержимого очереди
}
```

Пример использования класса Queue

```
float x;
Queue<float> Q; // Объявить объект «Очередь» с именем Q, класса Queue<float>
                // каждый элемент очереди будет представлен в виде числа типа float
x=1.1;
Q.Insert(x); // запись числа 1.1 в очередь
Q.print();// 1.1
x=1.2;
Q.Insert(2); // запись числа 1.2 в очередь
Q.print();// 1.1 1.2
Q.Delete(); // удаление числа 1.1 из очереди
Q.print();// 1.2
Q.Destroy(); // очистить содержимое очереди
```