МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**Т.Е. Мамонова**

# И Н Ф О Р М А Т И К А

## Общая информатика. Основы языка С++

*Рекомендовано в качестве учебного пособия
Редакционно-издательским советом
Томского политехнического университета*

Издательство
Томского политехнического университета
2011

ББК 32.973.2я73
УДК 681.3(075.8)
М22

**Мамонова Т.Е.**

М22 Информатика. Общая информатика. Основы языка С++: учебное пособие / Т.Е. Мамонова; Томский политехнический университет. – Томск: Изд-во Томского политехнического университета, 2011. – 206 с.

**В авторской редакции**

В учебном пособии в краткой форме изложены теоретические вопросы курса «Информатика», в том числе основные определения и основные технологии кодирования информации и программирования. Выделены важнейшие положения по программированию на языке высокого уровня С++. По каждой теме представлено большое количество тренировочных задач, включен справочный материал.

Пособие подготовлено на кафедре интегрированных компьютерных систем управления, соответствует программе дисциплины и предназначено для студентов ИДО, обучающихся по направлению 220700 «Автоматизация технологических процессов и производств».

**ББК 32.973.2я73**
**УДК 681.3(075.8)**

*Рецензенты*

Доктор технических наук,
профессор кафедры интегрированных
компьютерных систем управления ИК ТПУ
*А.М. Малышенко*

Кандидат технических наук,
доцент кафедры интегрированных
компьютерных систем управления ИК ТПУ
*В.Н. Шкляр*

# ОГЛАВЛЕНИЕ


**ВВЕДЕНИЕ....6**
**1. ОБЩИЕ ВОПРОСЫ ИНФОРМАТИКИ....8**
**1.1. Определение информатики....8**
**1.2. Технические средства информатики....9**
1.2.1. История развития вычислительной техники....9
1.2.2. Поколения ЭВМ....11
1.2.3. Архитектура ЭВМ....13
1.2.3.1. Классическая архитектура ЭВМ и принцип фон Неймана....13
1.2.3.2. Совершенствование и развитие внутренней структуры ЭВМ....15
1.2.3.3. Основной цикл работы ЭВМ....16
1.2.3.4. Система команд ЭВМ и способы обращения к данным....16
1.2.4. Типы и назначение компьютеров....19
1.2.5. Магистрально-модульный принцип построения компьютера....21
1.2.6. Периферийные и внутренние устройства....22
1.2.6.1. Центральный процессор....23
1.2.6.2.Оперативная память....24
1.2.6.3. Устройства хранения информации....25
1.2.6.4. Устройства ввода....26
1.2.6.6. Устройства связи....28
1.2.7. Программный принцип управления компьютером....29
**1.3. Компьютерные вирусы....29**
1.3.1. Основные признаки появления в системе вируса....31
1.3.2. Правовая охрана программ и GPL....32
**1.4. Операционные системы и сети....34**
1.4.1. Операционные системы....34
1.4.1.1. Операционная система MS DOS....35
1.4.1.2. Microsoft Windows....37
1.4.1.3.Операционная система Linux....39
**1.5. Обработка документов....40**
1.5.1. Текстовый процессор MS Word....41
**1.6 Вопросы для самоконтроля....53**
**2. АРИФМЕТИЧЕСКИЕ ОСНОВЫ ПОСТРОЕНИЯ ЭВМ....54**
**2.1. Единицы измерения информации....54**
**2.2. Системы счисления....56**
2.2.1. Двоичная система счисления....58
2.2.2. Восьмеричная и шестнадцатеричная системы счисления....60
2.2.3. Перевод чисел из одной системы счисления в другую....61
**2.3. Двоичное кодирование информации....64**
**2.4 Вопросы для самоконтроля....68**
**3. ОСНОВЫ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ ЗАДАЧ....70**
**3.1. Технология программирования и основные этапы её развития....70**
**3.2. Источники ошибок в программных средствах....73**

3.2.1. Интеллектуальные возможности человека....74
3.2.2. Неправильный перевод как причина ошибок в программных средствах....75
3.2.3. Основные пути борьбы с ошибками....78
**3.3. Понятие алгоритма, свойства алгоритмов....78**
**3.4. Языки программирования....81**
**3.5. Структурное программирование....82**
**3.6. Объектно-ориентированное программирование....84**
3.6.1. История....84
3.6.2. Основные понятия....86
3.6.3. Основные концепции ООП....86
3.6.4. Особенности реализации....89
3.6.5. Подходы к проектированию программ в целом....91
3.6.6. Родственные методологии....92
3.6.7. Производительность объектных программ....93
3.6.8. Критика ООП....95
3.6.9. Объектно-ориентированные языки....96
**3.7. Обобщённое программирование....97**
3.7.1. Общий механизм....98
3.7.2. Способы реализации....99
3.7.3. Обобщённое программирование в языке C++....100
**3.8 Вопросы для самоконтроля....100**
**4. ОСНОВЫ ЯЗЫКА C++....101**
**4.1. Типичная среда C++ программирования....103**
**4.2. Структура программы на C++....104**
**4.3. Базовые средства языка C++....107**
4.3.1. Состав языка C++....107
4.3.1.1. Константы в C++....108
4.3.2. Типы данных в C++....110
4.3.3. Переменные....112
4.3.4. Знаки операций в C++....114
4.3.5. Выражения....116
4.3.6. Ввод и вывод данных....117
**4.4. Основные операторы языка C++....119**
4.4.1. Базовые конструкции структурного программирования....119
4.4.2. Оператор «выражение»....119
4.4.3. Составные операторы....120
4.4.4. Операторы выбора....120
4.4.5. Операторы циклов....122
4.4.6. Операторы перехода....124
**4.5. Примеры решения задач с использованием основных операторов C++....125**
4.5.1. Программирование ветвлений....127
4.5.2. Программирование арифметических циклов....129
4.5.3. Итерационные циклы....131
4.5.4. Вложенные циклы....133
**4.6. Составные типы данных в C++....134**
4.6.1. Массивы....134

Определение массива в C/C++....134
4.6.2. Указатели....144
4.6.3. Ссылки....149
4.6.4. Указатели и массивы....150
**4.7. Символьная информация и строки....154**
**4.8. Функции в C++....159**
4.8.1. Объявление и определение функций....160
4.8.2. Прототип функции....162
4.8.3. Параметры функции....163
4.8.4. Локальные и глобальные переменные....165
4.8.5. Функции и массивы....166
4.8.5.1. Передача одномерных массивов как параметров функции....166
4.8.5.2. Передача строк в качестве параметров функций....169
4.8.5.3. Передача многомерных массивов в функцию....169
4.8.6. Функции с начальными (умалчиваемыми) значениями параметров....171
4.8.7. Подставляемые (inline) функции....171
4.8.8. Функции с переменным числом параметров....172
4.8.9. Перегрузка функций....174
4.8.10. Шаблоны функций....175
4.8.11. Указатель на функцию....177
4.8.12. Ссылки на функцию....179
**4.9. Типы данных, определяемые пользователем....180**
4.9.1. Переименование типов....180
4.9.2. Перечисления....181
4.9.3. Структуры....181
4.9.5. Битовые поля....184
4.9.6. Объединения....185
**4.10. Динамические структуры данных....186**
4.10.1. Линейный однонаправленный список....186
4.10.2. Работа с двунаправленным списком....190
**4.11. Ввод-вывод в C++....194**
4.11.1. Потоковый ввод-вывод....194
4.11.1.1 Открытие и закрытие потока....195
4.11.2. Стандартные файлы и функции для работы с ними....198
4.11.3. Символьный ввод-вывод....198
4.11.4. Строковый ввод-вывод....199
4.11.5. Блоковый ввод-вывод....200
4.11.6. Форматированный ввод-вывод....201
4.11.6.1 Прямой доступ к файлам....202
4.11.6.2 Удаление и добавление элементов в файле....203
**4.12 Вопросы для самоконтроля....204**
**СПИСОК ЛИТЕРАТУРЫ....206**
**ПРИЛОЖЕНИЕ....207**

# ВВЕДЕНИЕ

Данное учебное пособие предназначено для изучения курса «Информатика» для студентов ИДО, обучающихся по классической форме обучения (КЗФ) и с использованием дистанционных образовательных технологий (ДОТ) направления 220700 «Автоматизация технологических процессов и производств». В данном пособии представлены также дополнительные материалы, которые могут использоваться при выполнении лабораторных и курсовой работы по дисциплине «Информатика».

В этом пособии рассматриваются основные понятия и определения информатики, приведен материалы для изучения систем счисления, используемых в ЭВМ, привелдены практические примеры, при решении которых студент подготовится к сдаче зачёта по дисцииплине «Информатика». Также в данном пособии представлены примеры написания программ на языке программирования С++.

Весь материал разбит на разделы, в первом из которых описаны основы общие вопросы по информатике, включающие в свой состав такие вопросы, как история развития вычислительной техники, устройство и принцип работы компьютера. Во втором разделе представлен материал об арифметико-логическом устройстве, устройствах управления, память, логическое устройство компьютера, а также программное обеспечение ЭВМ. В третьем разделе кратко описаны основные технологии программирования, их достоинства и недостатки. Последний раздел посвящён основам программирования на языке высокого уровня С++, данный раздел предназначен для знакомства с языком и приобретения практических навыков для выполнения лабораторных и курсовой работ.

Настоящее пособие посвящено изучению «Информатики» – новой научной дисциплины и новой информационной индустрии, связанных с использованием персональных компьютеров и сети Интернет. Развитие бизнеса, образования, промышленности и общества в целом учеными, политиками, бизнесменами во многом связывается с широким использованием информационных ресурсов Интернет и нарастающими интеллектуальными возможностями вычислительных машин.

Наиболее распространенным видом современной вычислительной техники стали персональные компьютеры IBM PC. По этим причинам в пособии изучаются основные возможности наиболее современных программных средств персональных компьютеров IBM PC – операционной системы Windows, а также редактора текстов Word.

Представление информатики как научной дисциплины связано с рассмотрением проблем организации вычислений и обработки информации с помощью ЭВМ и внутри ЭВМ.

Особенностью информатики как учебной дисциплины является практикум на ЭВМ, который может проводиться в вузе или дома. Для прохождения такого практикума необходимо иметь персональный компьютер или доступ к нему, а также необходимые пакеты программ – редакторы текстов, базы данных, электронные таблицы и др.

Настоящее пособие построено так, что по нему можно изучать информатику даже дома с использованием персональной ЭВМ, и последующей сдачей зачета с помощью Интернет (для студентов, обучающихся с применением ДОТ). Такая возможность требует наличия на домашних ЭВМ перечисленных пакетов программ с операционными системами, редакторами текстов и системами программирования.

# 1. ОБЩИЕ ВОПРОСЫ ИНФОРМАТИКИ

## 1.1. Определение информатики

*Информатика* – это техническая наука, систематизирующая приёмы создания, хранения, воспроизведения, обработки и передачи данных средствами вычислительной техники, а также принципы функционирования этих средств и методы управления ими [1].

*Информатика* – молодая научная дисциплина, изучающая вопросы, связанные с поиском, сбором, хранением, преобразованием и использованием информации в самых различных сферах человеческой деятельности. Генетически информатика связана с вычислительной техникой, компьютерными системами и сетями, так как именно компьютеры позволяют порождать. Хранить и автоматически перерабатывать информацию в таких количествах, что научный подход к информационным процессам становится одновременно необходимым и возможным [9].

Рассмотрим состав ядра современной информатики [9]. Каждая из этих частей может рассматриваться как относительно самостоятельная научная дисциплина; взаимоотношения между ними примерно такие же, как между алгеброй, геометрией и математическим анализом в классической математике – все они хоть и самостоятельные дисциплины, но, несомненно, части одной науки.

*Теоретическая информатика* – часть информатики, включающая часть математических разделов. Она опирается на математическую логику и включает такие разделы как теория алгоритмов и автоматов, теория информации и теория кодирования, теория формальных языков и грамматик, исследование операций и др. Этот раздел информатики использует математические методы для изучения процессов обработки информации.

*Вычислительная техника* – раздел информатики, в котором разрабатываются принципы построения вычислительных систем. Речь идёт не о технических деталях и электронных схемах (это лежит за пределами информатики как таковой), а о принципиальных решениях на уровне так называемой *архитектуры* вычислительных (компьютерных) систем, определяющей состав назначение, функциональные возможности и принципы взаимодействия устройств. Примеры принципиальных, ставших классическими решений в этой области – неймановская архитектура компьютеров первых поколений, шинная архитектура ЭВМ старших поколений, архитектура параллельной (многопроцессорной) обработки информации.

*Программирование* – деятельность, связанная с разработкой систем программного обеспечения. Например, это – создание системного программного обеспечения и создание прикладного программного обеспечения. Среди системного – разработка новых языков программирования и компиляторов к ним, разработка интерфейсных систем, (пример – общеизвестная операционная оболочка и система Windows). Среди прикладного программного обеспечения – системы обработки текстов, электронные таблицы, системы управления базами данных.

## 1.2. Технические средства информатики

### 1.2.1. История развития вычислительной техники

Всё началось с идеи научить машину считать или хотя бы складывать многоразрядные целые числа. Ещё около 1500 г. Леонардо да Винчи разработал эскиз 13-разрядного суммирующего устройства, что явилось первой дошедшей до нас попыткой решить указанную задачу [4].

**1642** г. – француз Блез Паскаль (физик, математик, инженер) построил 8-разрядную суммирующую машину – прообраз арифмометров, использовавшихся вплоть до середины 70-х годов XX века.

**1822** г.– английский математик Чарльз Бэббидж сконструировал и почти 30 лет строил машину, которая вначале называлась «разностной», а затем – «аналитической». В эту машину были заложены принципы, ставшими фундаментальными для вычислительной техники:

1) автоматическое выполнение операций;

2) автоматический ввод программы (записывалась на перфокарты);

3) наличие специального устройства (памяти) для хранения данных.

На основе механической техники эти идеи реализовать не удалось.

**1944** г. – под руководством Говарда Айкена (американского математика и физика) на фирме IBM (International Business Machines) была запущена машина «Марк-1», впервые реализовавшая идеи Бэббиджа. Для представления чисел в ней были использованы механические элементы (счётные колёса), для управления – электромеханические.

**1945–1946** гг. – под руководством Джона Моучли и Преспера Эккерта в США создана первая электронная вычислительная машина (ЭВМ) ENIAC. В ней использовалось 18000 электронных ламп, энергопотребление равнялось 150 кВт.

**1949** г. – в Великобритании была построена первая ЭВМ с хранимой программой (EDSAC). Принцип хранимой программы требует, чтобы программы закладывались в память машины так же, как в неё закладывается исходная информация.

**1951** г. – в СССР под руководством Сергея Александровича Лебедева создана МЭСМ – малая электронно-счётная машина.

**1964** г. – появление интегральных схем

**1965** г. – первый миникомпьютер

При создании компьютеров используются специалисты различных направлений – математики, физики, техники, программисты и т.д. В этом смысле информатика определялась как совокупность дисциплин изучающих свойства информации, а так же процессы передачи, накопления, обработки информации с помощью технических средств.

Выделяется часть науки, которая занимается проблемами применения средств вычислительной техники для работы с информацией. В Англии и США это Computer Science (наука о вычислительной технике), во Франции – informatique (информатика). В 60-е годы происходит становление информатики, как фундаментальной естественной науки изучающей процессы обработки, передачи и накопления информации. Данная дисциплина создана на стыке точных и естественных наук. Ядро информатики – информационные технологии.

*Информационная технология* – совокупность технических и программных средств, с помощью которых обрабатывается информация. Центральное место в информационных технологиях занимает компьютер.

**1970-е** г. – создание БИС (большой интегральной схемы).

**1970** г. – создана саморазмножающаяся программа для одной из первых компьютерных сетей – ARPnet. Программа Creepeer, которая по некоторым данным была написана Бобом Томасом, путешествовала по сети, обнаруживая свое появление сообщением «Я КРИППЕР... ПОЙМАЙ МЕНЯ, ЕСЛИ СМОЖЕШЬ».

**1971** г. – создан первый микрокомпьютер Kenback1

**1972** г. – 31-летний специалист по системному программированию из фирмы Bell Labs Деннис Ритчи разработал язык программирования С.

**1972** г. – опубликована работа Эдсгера Дайкстры «Заметки по структурному программированию», содержащая блестящее описание основных идей структурного программирования

**1973** г. – швейцарский специалист по программированию Никлаус Вирт опубликовал «Пересмотренное сообщение», определившее точный стандарт языка Pascal. Строгий стиль языка Pascal был с восторгом принят приверженцами структурного программирования.

**1975** г. – год образования фирмы Microsoft.

**1977** г. – первый микрокомпьютер Уозняка и Джобса, выпущенный фирмой Apple.

**1980** г. – создан центральный процессор на одном кремниевом кристалле.

**1980-е** г. – появление СБИС.

*ЭВМ* – это программируемое электронное устройство обработки и накопления информации [4].

### 1.2.2. Поколения ЭВМ

В истории вычислительной техники существует периодизация ЭВМ по поколениям. В её основу был положен физико-технологический принцип: машину относят к тому или иному поколению в зависимости от используемых в ней физических элементов или технологии их изготовления. Границы поколений во времени размыты, так как в одно и то же время выпускались машины разного уровня. Когда приводят даты, относящиеся к поколениям, то скорее всего имеют в виду период промышленного производства.

В настоящее время физико-технологический принцип не является единственным при определении принадлежности той или иной ЭВМ к поколению. Следует считаться и с уровнем программного обеспечения, с быстродействием, другими факторами, основные из которых сведены в таблицу (Приложение).

Опишем поколения ЭВМ более подробно [5]:

**1 поколение (1944–1958).** Ламповые машины с быстродействием порядка 10–20 тыс. операций в секунду, программы писались на машинном языке (рис. 1).

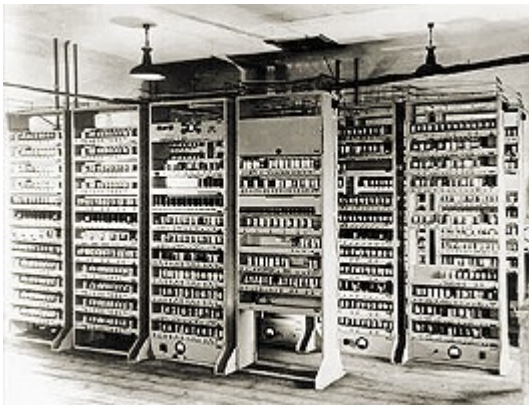


Рис. 1. ЭВМ EDSAC

**2 поколение (1959 – 1963).** Полупроводниковые машины на транзисторах. Быстродействие 100 тыс. операций в секунду. Имеются программы перевода с алгоритмических языков на машинный язык. Есть набор стандартных программ (рис. 2).

Рис. 2. ЭВМ ENIAK

**<u>3 поколение (1964 – 1970)</u>.** Миникомпьютеры на интегральных схемах (рис. 3). Отличаются большей надежностью и малыми размерами. Быстродействие 10 млн. оп/с. Образуют системы программно-совместимых устройств.

Рис. 3. Миникомпьютер
на интегральных схемах

**<u>4 поколение (1971 – до сегодняшнего дня)</u>.** Вычислительные системы на больших интегральных схемах (БИС). Имеют большой объем памяти, позволяют подключать большое количество устройств ввода и вывода информации (рис. 4). Для ввода данных и команд используется клавиатура. Микропроцессор, разработанный, в 1971 году позволил создать центральный процессор на одном чипе.

Рис. 4. Миникомпьютер «Электроника» на БИС

**5 поколение (настоящее и будущее).** Еще создается, предполагается развитие искусственного интеллекта на основе оптико-лазерных технологий и применения СБИС. Планируется создать компьютер с большим быстродействием, огромным по мощности процессором и неограниченной виртуальной памятью.

В качестве основного элемента для электрических цепей будет использован арсенид галия. Работа этих компьютеров будет основана на параллельных вычислениях.

При определении признаков пятого поколения среди специалистов нет единодушия. В середине 80-х годов считалось, что основной признак этого будущего поколения – полновесная реализация принципов искусственного интеллекта. Эта задача оказалась значительно сложнее, чем виделось в то время. И ряд специалистов снижают планку требований к этому этапу (и даже утверждают, что он уже состоялся). Указанные в верхней строчке таблицы даты соответствуют первым годам выпуска ЭВМ.

### 1.2.3. Архитектура ЭВМ

*Архитектура* – это наиболее общие принципы построения ЭВМ, реализующие программное управление работой и взаимодействие основных её функциональных узлов [9].

Наиболее общие принципы построения ЭВМ, которые относятся к архитектуре:

1) структура памяти ЭВМ;
2) способы доступа к памяти и внешним устройствам;
3) возможность изменения конфигурации компьютера;
4) система команд;
5) организация интерфейса.

#### 1.2.3.1. Классическая архитектура ЭВМ и принцип фон Неймана

Основы учения об архитектуре вычислительной машины заложил американский математик Джон фон Нейман. В 1946 г. он с соавторами изложил свои принципы построения вычислительных машин в статье «Предварительное рассмотрение логической конструкции электронно-вычислительного устройства».

В статье обосновывается использование двоичной системы для представления чисел (ранее все ВМ хранили обрабатываемые числа в десятичном виде). В дальнейшем ЭВМ стали обрабатывать и нечисловые виды информации – текстовую, графическую, звуковую и др., но

двоичное кодирование данных по-прежнему составляет информационную основу компьютера.

Революционным является предложенный Нейманом принцип «хранимой программы». Первоначально программа задавалась путём установки перемычек на специальной коммутационной панели. Это было трудоёмким занятием (изменение программы для машины ENIAC требовало несколько дней). Нейман догадался, что программа может также храниться в виде набора нулей и единиц, причём в той же самой памяти, что и обрабатываемое ею число. Отсутствие принципиальной разницы между программой и данными дало возможность ЭВМ самой формировать для себя программу в соответствии с результатами вычислений.

Фон Нейман не только выдвинул принципы логического устройства ЭВМ, но и предложил её структуру, которая воспроизводилась в течение первых двух поколений ЭВМ. Основными блоками по Нейману являются устройство управления (УУ) и арифметико-логическое устройство (АЛУ) (обычно объединённые в центральный процессор), память, внешняя память, устройства ввода и вывода. Схема устройства такой ЭВМ представлена на рис. 5.

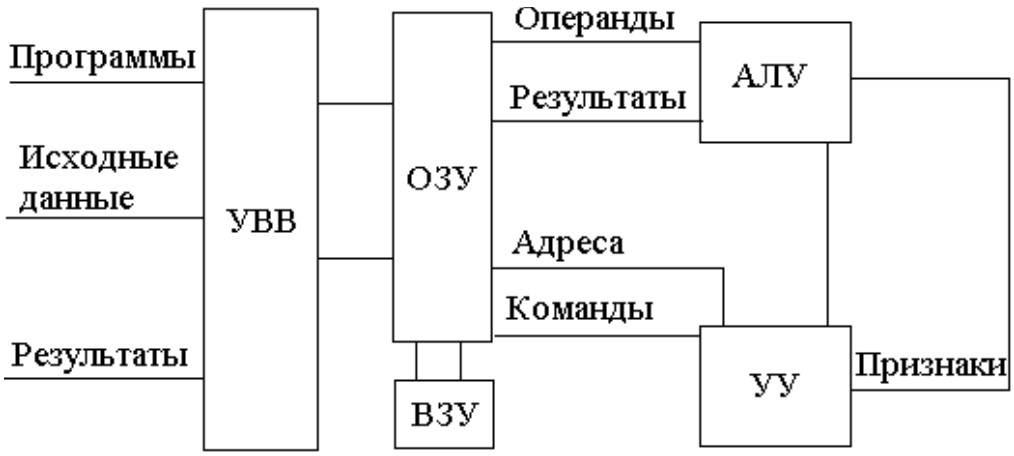



Рис. 5. Архитектура ЭВМ, построенной по принципам фон Неймана.

УУ и АЛУ в современных компьютерах объединены в один блок – процессор, являющийся преобразователем информации, поступающей из памяти и внешних устройств. Сюда относятся выборка команд из памяти, кодирование и декодирование, Выполнение различных, в том числе и арифметических, операций, согласование работы узлов компьютера.

Память (ЗУ) хранит информацию (данные) и программы. Запоминающее устройство у современных компьютеров «многоярусно» и включает:

– ОЗУ, хранящее ту информацию, с которой компьютер работает непосредственно в данное время (исполняемая программа, часть необходимых для неё данных, некоторые управляющие программы);

– ВЗУ гораздо большей ёмкости, чем ОЗУ, но с существенно более медленным доступом.

На ОЗУ и ВЗУ классификация устройств памяти не заканчивается. Определённые функции выполняют и сверхоперативное запоминающее устройство (СОЗУ), и постоянное запоминающее устройство (ПЗУ), и другие виды компьютерной памяти.

В построенной по описанной схеме ЭВМ происходит последовательное считывание команд из памяти и их выполнение. Номер (адрес) очередной ячейки памяти, из которой будет извлечена следующая команда программы, указывается специальным устройством – счётчиком команд в УУ. Его наличие также является одним из характерных признаков рассматриваемой архитектуры.

Подавляющее большинство вычислительных машин на сегодняшний день имеют фон-неймановскую архитектуру. Исключение составляют отдельные разновидности систем для параллельных вычислений, в которых отсутствует счётчик команд, не реализована классическая концепция переменной и т.д.

По-видимому, значительное отклонение от фон-неймановской архитектуры произойдёт в результате развития идея машины пятого поколения, в основе обработки информации в которых лежат не вычисления, а логические выводы.

#### 1.2.3.2. Совершенствование и развитие внутренней структуры ЭВМ

Появление третьего поколения ЭВМ было обусловлено переходом от транзисторов к интегральным микросхемам. Возникло противоречие между высокой скоростью обработки информации внутри машины и медленной работой устройств ввода-вывода, в большинстве своём содержащих механические движущиеся части. Процессор, руководивший работой внешних устройств, значительную часть времени был бы вынужден простаивать в ожидании информации из «внешнего мира». Для решения этой проблемы возникла тенденция к освобождению центрального процессора от функций обмена и к передаче их специальным электронным схемам управления работой внешних устройств – контроллерам.

Наличие интеллектуальных контроллеров внешних устройств стало важной отличительной чертой машин третьего и четвёртого поколений.

Контроллер можно рассматривать как специализированный процессор, управляющий работой внешнего устройства по специальным встроенным программам обмена.

### 1.2.3.3. Основной цикл работы ЭВМ

Каждая программа состоит из отдельных машинных команд. Каждая машинная команда делится на ряд элементарных унифицированных составных частей, которые принято называть тактами. В зависимости от сложности команды она может быть реализована за разное число тактов.

При выполнении каждой команды ЭВМ проделывает определённые стандартные действия:

1. Согласно содержимому счётчика адреса команд, считывается очередная команда программы. Её код обычно заносится на хранение в специальный регистр УУ, который носит название регистра команд.

2. Счётчик команд автоматически изменяется так, чтобы в нём содержался адрес следующей команды. В простейшем случае для этой цели достаточно к текущему значению счётчика прибавить некоторую константу, определяющуюся длиной команды.

3. Считанная в регистр команд операция расшифровывается, извлекаются необходимые данные и над ними выполняются требуемые действия.

Затем во всех случаях, за исключением команды останова или наступления прерывания, все описанные действия циклически повторяются.

Если требуется изменить порядок вычислений для реализации развилки или цикла, достаточно в счётчик команд занести требуемый адрес.

### 1.2.3.4. Система команд ЭВМ и способы обращения к данным

Система команд любой ЭВМ обязательно содержит следующие группы команд обработки информации [8].

1. Команды передачи данных (перепись), копирующие информацию из одного места в другое.

2. Арифметические операции. К основным арифметическим действиям обычно относятся сложение и вычитание. Умножение и деление во многих ЭВМ выполняются по специальным программам.

3. Логические операции, позволяющие компьютеру анализировать обрабатываемую информацию (И, ИЛИ, НЕ, < и т.д.).

4. Сдвиги двоичного кода влево или вправо.

5. Команды ввода и вывода информации для обмена с внешними устройствами.

6. Команды управления, реализующие нелинейные алгоритмы, например, условный и безусловный переход.

Команда ЭВМ обычно состоит из двух частей – операционной и адресной. Операционная часть (иначе она ещё называется кодом операции КОП) указывает, какое действие необходимо выполнить с информацией. Адресная часть описывает, где используемая информация хранится. У нескольких команд адресная часть может отсутствовать, например, в команде останова. Операционная часть имеется всегда.

Код операции можно представить как некоторый условный номер в общем списке системы команд, например, команда 010 – сложение. Адресная часть обладает значительно бóльшим разнообразием.

Команды могут быть одно- , двух- и трёхадресные в зависимости от участвующих в них операндов.

Первые ЭВМ имели наиболее простую и наглядную трёхадресную систему команд. Например,

010 A1 A2 A3

означает, что надо взять числа из адресов памяти A1 и A2, сложить их и сумму поместить в адрес A3. Если для операции требовалось меньшее число адресов, то лишние просто не использовались. Например, в операции переписи указывались лишь ячейки источника и приёмника информации A1 и A3, а содержимое A2 не имело никакого значения.

Трёхадресная команда легко расшифровывалась и была удобна в использовании, но с ростом объёмов ОЗУ её длина становилась непомерно большой. Длина команды складывается из длины трёх адресов и кода операции. Отсюда следует, например, что для ОЗУ из 1024 ячеек только для записи адресной части одной команды требуется 3*10=30 двоичных разрядов, что для технической реализации не очень удобно. Поэтому появились двухадресные машины, длина команды в которых сокращалась за счёт исключения адреса записи результата. В таких ЭВМ результат операции оставался в специальном регистре (сумматоре) и был пригоден для использования в последующих вычислениях. В некоторых машинах результат записывался вместо одного из операндов.

Дальнейшее упрощение команды привело к созданию одноадресных машин. Рассмотрим систему команд такой ЭВМ на примере. Пусть надо сложить числа, хранящиеся в ячейках с адресами ОЗУ A1 иA2, а сумму поместить в ячейку с адресом A3. Для решения этой задачи одноадресной машине потребуется выполнить три команды:

1) извлечь содержимое ячейки A1 в сумматор;

2) сложить сумматор с числом из A2;

3) записать результат из сумматора в A3.

Рассмотрим теперь вопрос об адресации элементов ОЗУ более подробно. Наиболее просто была организована память в ЭВМ первых двух поколений. Она состояла из отдельных ячеек, содержимое каждой из которых считывалось или записывалось как единое целое. Каждая ячейка памяти имела свой номер, который и получил название адреса. Адреса соседних ячеек ОЗУ являются последовательными числами, т.е. отличаются на единицу. В рассмотренных ЭВМ использовались данные только одного типа (вещественные числа), причём их длина равнялась длине машинной команды и совпадала с разрядностью памяти (36 двоичных разрядов) и всех остальных устройств машины.

Очень часто программа предназначалась для обработки по одним и тем же формулам определённого количества содержимого последовательно расположенных ячеек (массивов). В ЭВМ первых двух поколений были предусмотрены особые механизмы циклической обработки массивов. С этой целью в машинных командах помимо обычных адресов можно было использовать модифицируемые, у которых специальный управляющий бит был установлен в единицу. К помеченным таким образом модифицируемым адресам при выполнении команды прибавлялось значение из специальных индексных ячеек. Меняя содержимое индексных ячеек, можно было получить доступ к различным элементам массива. Формирование результирующего адреса осуществлялось в УУ в момент исполнения команды, поэтому исходная команда в ОЗУ сохранялась без изменений.

Описанный механизм модификации адресов существенно упрощал написание циклических программ, таких как нахождение суммы последовательных ячеек ОЗУ, копирование отдельных участков памяти и т.д.

В ЭВМ третьего поколения идеология построения памяти существенно изменилась: минимальная порция информации для обмена с ОЗУ была установлена равной 8 двоичных разрядов, т.е. один байт. Стало возможным обрабатывать несколько типов данных: символы текста (1 байт), целые числа (2 байта), вещественные числа обычной или двойной точности (4 или 8 байт соответственно). В связи с этим была введена новая условная единица измерения информации – машинное слово. Оно равнялось 4 байтам и соответствовало длине стандартного вещественного числа. Все объёмы информации начали измеряться в единицах, кратных слову: двойное слово, полуслово и т.п. Естественно, что адрес (номер ячейки ОЗУ) в машинах с байтовой организацией стал относиться к отдельному байту. Байты памяти имеют возрастающие на единицу номера. Слово состоит из нескольких последовательно расположенных байтов. В качестве адреса слова удобно принимать адрес одного из образующих его байтов (обычно используется младший байт,

имеющий наименьший номер). Таким образом, адреса слов меняются уже не через единицу. Их приращение зависит от длины машинного слова в байтах и равняется четырём.

### 1.2.4. Типы и назначение компьютеров

Существование различных типов компьютеров определяется различием задач, для решения которых они предназначены. С течением времени появляются новые типы задач, что приводит к появлению новых типов компьютеров. Поэтому приведенное ниже деление очень условно [12].

Различают:

1) суперкомпьютеры;

2) специализированные компьютеры-серверы;

3) встроенные компьютеры-невидимки (микропроцессоры);

4) персональные компьютеры.

Для выполнения изначального назначения компьютеров – вычислений – на рубеже 60–70-х годов были созданы специализированные ЭВМ, так называемые *суперкомпьютеры*.

*Суперкомпьютеры* – специальный тип компьютеров, создающихся для решения предельно сложных вычислительных задач (составления прогнозов, моделирования сложных явлений, обработки сверхбольших объемов информации). Принцип работы суперкомпьютера заключается в том, что он способен выполнять несколько операций параллельно.

Одной из ведущих компаний мира в производстве суперкомпьютеров является компания Cray Research. Ее основатель, человек-легенда Сеймур Крей, уже в середине 70-х годов построил компьютер **Cray-1**, который поражал мир своим быстродействием: десятки и даже сотни миллионов арифметических операций в секунду.

Как известно, скорость распространения любого сигнала не превышает скорости света в вакууме – 300 тысяч километров в секунду, или 300 миллионов метров в секунду. Если компьютер выполняет 300 миллионов операций в секунду, то за время выполнения одной операции сигнал успевает пройти не более одного метра. Отсюда следует, что расстояние между частями суперкомпьютера, выполняющими одну операцию, не может превосходить нескольких десятков сантиметров. И действительно, суперкомпьютеры компании Cray были очень компактны и выглядели как «бублик» диаметром менее двух метров. Этот «бублик» занимался только вычислениями. Для общения с человеком и доставки данных для вычислений к «бублику» были подключены несколько достаточно производительных обычных компьютеров.

Компьютер, работающий в локальной или глобальной сети, может специализироваться на оказании информационных услуг другим компьютерам, на обслуживании других компьютеров. Такой компьютер называется *сервером* от английского слова *serve* (в переводе – обслуживать, управлять). В локальной сети один из компьютеров может выполнять функции *файлового сервера*, т.е. использоваться для долговременного хранения файлов.

Основная задача, решаемая файловыми серверами, – организация хранения, доступа и обмена данными (информацией) между компьютерами, людьми и другими источниками и поставщиками информации. Требования к серверам иные, чем к суперкомпьютеру. Важно наличие у них устройств хранения информации (типа магнитных дисков) большой емкости, скорость же обработки информации не столь критична.

В классе серверов выделяется подкласс *суперсерверов*, необходимых в тех случаях, когда, с одной стороны, желательна централизация данных, а с другой стороны, к этим данным необходимо обеспечить доступ очень большому количеству потребителей.

Кроме привычных компьютеров с клавиатурами, мониторами, дисководами, сегодняшний мир вещей наполнен компьютерами-невидимками. *Микропроцессор* представляет собой компьютер в миниатюре. Кроме обрабатывающего блока, он содержит блок управления и даже память (внутренние ячейки памяти). Это значит, что микропроцессор способен автономно выполнять все необходимые действия с информацией. Многие компоненты современного персонального компьютера содержат внутри себя миниатюрный компьютер. Массовое распространение микропроцессоры получили и в производстве, там где управление может быть сведено к отдаче ограниченной последовательности команд.

Микропроцессоры незаменимы в современной технике. Например, управление современным двигателем – обеспечение экономии расхода топлива, ограничение максимальной скорости движения, контроль исправности и т.д. – немыслимо без использования микропроцессоров. Еще одной перспективной сферой их использования является бытовая техника – применение микропроцессоров придает ей новые потребительские качества.

В 1975 году появился первый *персональный компьютер*. С самого начала их выпуска стало ясно, что невысокая цена и достаточные вычислительные возможности этого нового класса компьютеров будут способствовать их широкому распространению.

Персональные компьютеры совершили компьютерную революцию в профессиональной деятельности миллионов людей и оказали огромное влияние на все стороны жизни человеческого общества. Компьюте-

ры этого типа стали незаменимым инструментом работы инженеров и ученых. Особо велика их роль при проведении научных экспериментов, требующих сложных и длительных вычислений.

В последние годы появилась разновидность персонального компьютера – так называемый *домашний компьютер*. По сути, он ничем не отличается от персонального, только используется для других целей: развлекательных и образовательных.

Идея *сетевого компьютера***,** работающего только в сети и представляющего собой упрощенный вариант персонального компьютера, все больше занимает умы разработчиков. Такому компьютеру не нужно хранить программы, он в любой момент может получить их по сети.

### 1.2.5. Магистрально-модульный принцип построения компьютера

Под архитектурой компьютера понимается его логическая организация, структура, ресурсы, т. е. средства вычислительной системы, которые могут быть выделены процессу обработки данных на определенный интервал времени. Архитектура современных ПК основана на *магистрально-модульном принципе.*

Модульный принцип позволяет потребителю самому подобрать нужную ему конфигурацию компьютера и производить при необходимости его модернизацию. Модульная организация системы опирается на магистральный (шинный) принцип обмена информации. Магистраль или *системная шина* – это набор электронных линий, связывающих воедино по адресации памяти, передачи данных и служебных сигналов процессор, память и периферийные устройства.

Обмен информацией между отдельными устройствами ЭВМ производится по трем многоразрядным шинам, соединяющим все модули, – *шине данных, шине адресов* и *шине управления*.

Подключение отдельных модулей компьютера к магистрали на физическом уровне осуществляется с помощью контроллеров, а на программном обеспечивается драйверами. Контроллер принимает сигнал от процессора и дешифрует его, чтобы соответствующее устройство смогло принять этот сигнал и отреагировать на него. За реакцию устройства процессор не отвечает – что функция контроллера. Поэтому внешние устройства ЭВМ заменяемы, и набор таких модулей произволен.

Разрядность *шины данных* задается разрядностью процессора, т. е. количеством двоичных разрядов, которые процессор обрабатывает за один такт.

Данные по шине данных могут передаваться как от процессора к какому-либо устройству, так и в обратную сторону, т.е. шина данных является двунаправленной. К основным режимам работы процессора с использованием шины передачи данных можно отнести следующие: запись/чтение данных из оперативной памяти и из внешних запоминающих устройств, чтение данных с устройств ввода, пересылка данных на устройства вывода.

Выбор абонента по обмену данными производит процессор, который формирует код адреса данного устройства, а для ОЗУ – код адреса ячейки памяти. Код адреса передается по *адресной шине*, причем сигналы передаются в одном направлении, от процессора к устройствам, т.е. эта шина является однонаправленной.

По *шине управления* передаются сигналы, определяющие характер обмена информацией, и сигналы, синхронизирующие взаимодействие устройств, участвующих в обмене информацией.

Внешние устройства к шинам подключаются посредством *интерфейса*. Под интерфейсом понимают совокупность различных характеристик какого-либо периферийного устройства ПК, определяющих организацию обмена информацией между ним и центральным процессором. В случае несовместимости интерфейсов (например, интерфейс системной шины и интерфейс винчестера) используют *контроллеры*.

Чтобы устройства, входящие в состав компьютера, могли взаимодействовать с центральным процессором, в IBM-совместимых компьютерах предусмотрена *система прерываний (Interrupts)*. Система прерываний позволяет компьютеру приостановить текущее действие и переключиться на другие в ответ на поступивший запрос, например, на нажатие клавиши на клавиатуре. Ведь с одной стороны, желательно, чтобы компьютер был занят возложенной на него работой, а с другой – необходима его мгновенная реакция на любой требующий внимания запрос. Прерывания обеспечивают немедленную реакцию системы.

### 1.2.6. Периферийные и внутренние устройства

Прогресс компьютерных технологий идет семимильными шагами. Каждый год появляются новые процессоры, платы, накопители и прочие периферийные устройства. Рост потенциальных возможностей ПК и появление новых более производительных компонентов неизбежно вызывает желание модернизировать свой компьютер. Однако нельзя в полной мере оценить новые достижения компьютерной технологии без сравнения их с существующими стандартами.

Разработка нового в области ПК всегда базируется на старых стандартах и принципах. Поэтому знание их является основополагающим фактором «для» (или «против») выбора новой системы.

В состав ЭВМ входят следующие компоненты [7]:

1) центральный процессор (*CPU*);
2) оперативная память (*memory*);
3) устройства хранения информации (*storage devices*);
4) устройства ввода (*input devices*);
5) устройства вывода (*output devices*);
6) устройства связи (*communication devices*).

### 1.2.6.1. Центральный процессор

Во всех вычислительных машинах до середины 50-х годов устройства обработки и управления представляли собой отдельные блоки, и только с появлением компьютеров, построенных на транзисторах, удалось объединить их в один блок, названный *процессором.*

*Процессор* – это мозг ЭВМ. Он контролирует действия всех остальных устройств (*devices*) компьютера и координирует выполнение программ. Процессор имеет свою внутреннюю память, называемую *регистрами,* управляющее и арифметико-логическое устройства.

Процесс общения процессора с внешним миром через устройства ввода-вывода по сравнению с информационными процессами внутри него протекает в сотни и тысячи раз медленнее. Это связано с тем, что устройства ввода и вывода информации часто имеют механический принцип действия (принтеры, клавиатура, мышь) и работают медленно. Чтобы освободить процессор от простоя при ожидании окончания работы таких устройств, в компьютер вставляются специализированные микропроцессоры-контроллеры (от англ. *controller* – управляющий). Получив от центрального процессора компьютера команду на вывод информации, контроллер самостоятельно управляет работой внешнего устройства. Окончив вывод информации, контроллер сообщает процессору о завершении выполнения команды и готовности к получению следующей.

Число таких контроллеров соответствует числу подключенных к процессору устройств ввода и вывода. Так, для управления работой клавиатуры и мыши используется свой отдельный контроллер. Известно, что даже хорошая машинистка не способна набирать на клавиатуре больше 300 знаков в минуту, или 5 знаков в секунду. Чтобы определить, какая из ста клавиш нажата, процессор, не поддержанный контроллером, должен был бы опрашивать клавиши со скоростью 500 раз в секун-

ду. Конечно, по его меркам это не бог весть какая скорость. Но это значит, что часть своего времени процессор будет тратить не на обработку уже имеющейся информации, а на ожидание нажатий клавиш клавиатуры.

Таким образом, использование специальных контроллеров для управления устройствами ввода-вывода, усложняя устройство компьютера, одновременно разгружает его центральный процессор от непроизводительных трат времени и повышает общую производительность компьютера.

### 1.2.6.2.Оперативная память

Существует два типа оперативной памяти – *память с произвольным доступом* (*RAM* или *random access memory*) и *память*, *доступная только на чтение* (*ROM* или *read only memory*). Процессор ЭВМ может обмениваться данными с оперативной памятью с очень высокой скоростью, на несколько порядков превышающей скорость доступа к другим носителям информации, например дискам.

Оперативная *память с произвольным доступом* (*RAM*) служит для размещения программ, данных и промежуточных результатов вычислений в процессе работы компьютера. Данные могут выбираться из памяти в произвольном порядке, а не строго последовательно, как это имеет место, например, при работе с магнитной лентой. *Память, доступная только на чтение* (*ROM*) используется для постоянного размещения определенных программ (например, программы начальной загрузки ЭВМ). В процессе работы компьютера содержимое этой памяти не может быть изменено.

Оперативная память – временная, т.е. данные в ней хранятся только до выключения ПК. Для долговременного хранения информации служат дискеты, винчестеры, компакт-диски и т.п. Конструктивно элементы памяти выполнены в виде модулей, так что при желании можно сравнительно просто заменить их или установить дополнительные и тем самым изменить объем общей оперативной памяти компьютера. Основными характеристиками элементов (микросхем) памяти являются: тип, емкость, разрядность и быстродействие.

В настоящее время отдельные микросхемы памяти не устанавливаются на материнскую плату. Они объединяются в специальных печатных платах, образуя вместе с некоторыми дополнительными элементами модули памяти (SIMM- и DIMM-модули).

### 1.2.6.3. Устройства хранения информации

*Устройства хранения информации* используются для хранения информации в электронной форме. Любая информация – будь это текст, звук или графическое изображение, – представляется в виде последовательности нулей и единиц. Ниже перечислены наиболее распространенные устройства хранения информации.

*Винчестеры* (*hard discs*)

Жесткие диски – наиболее быстрые из внешних устройств хранения информации. Кроме того, информация, хранящаяся на винчестере, может быть считана с него в произвольном порядке (диск – устройство с произвольным доступом).

Емкость диска современного персонального компьютера составляет десятки гигабайт. В одной ЭВМ может быть установлено несколько винчестеров.

*Оптические диски (cdroms)*

Лазерные диски, как их еще называют, имеют емкость около 600 мегабайт и обеспечивают только считывание записанной на них однажды информации в режиме произвольного доступа. Скорость считывания информации определяется устройством, в которое вставляется компакт-диск (cdrom drive).

*Магнито-оптические диски*

В отличие от оптических дисков магнито-оптические диски позволяют не только читать, но и записывать информацию.

*Флоппи диски (floppy discs)*

В основе этих устройств хранения лежит гибкий магнитный диск, помещенный в твердую оболочку. Для того чтобы прочитать информацию, хранящуюся на дискете, ее необходимо вставить в дисковод (floppy disc drive) компьютера. Емкость современных дискет всего 1.44 мегабайта. По способу доступа дискета подобна винчестеру.

*Zip and Jaz Iomega discs*

Это относительно новые носители информации, которые призваны заменить гибкие магнитные диски. Их можно рассматривать, как быстрые и большие по емкости (100 мегабайт – Zip, 1 гигабайт – Jaz) дискеты.

*Магнитные ленты* (*magnetic tapes*)

Современные магнитные ленты, хранящие большие объемы информации (до нескольких гигабайт), внешне напоминают обычные магнитофонные кассеты и характеризуются строго последовательным доступом к содержащейся на них информации.

### 1.2.6.4. Устройства ввода

*Устройства ввода* передают информацию в ЭВМ от различных внешних источников. Информация может быть представлена в весьма различных формах: текст – для клавиатуры (*keyboard*), звук – для микрофона (*microphone*), изображение – для сканера (scanner).

*Клавиатура* – одно из самых распространенных на сегодня устройств ввода информации в компьютер. Она позволяет нажатием клавиш вводить символьную информацию.

Ключевой принцип работы клавиатуры заключается в том, что она воспринимает нажатия клавиш и преобразует их в двоичный код, индивидуальный для каждой клавиши.

Но указывать место на экране монитора, в котором компьютер что-то должен изменить, с помощью клавиатуры неудобно. Для этого существует специальное устройство ввода – *мышь*.

Принцип ее действия основан на измерении направления и величины поворота шарика, находящегося в нижней части мыши. Когда мы перемещаем мышь по поверхности стола, шарик поворачивается. Специальные датчики измеряют поворот шарика. После преобразования результатов измерения в двоичный код они передаются в компьютер. По ним процессор выводит на экран условное изображение указателя (обычно в форме стрелки). Существуют разновидности этого устройства – *оптические мыши*, принцип действия которых основан на отслеживании перемещения луча света. Часто для них требуется специальный металлический коврик.

Мышь не позволяет вводить числовую и буквенную информацию, но удобна для работы с графическими объектами, изображенными на экране.

*Сканер* – устройство ввода графической информации. Его особенность – способность считывать изображение непосредственно с листа бумаги.

Принцип действия сканера напоминает работу человеческого глаза. Освещенный специальным источником света, находящимся в самом сканере, лист бумаги с текстом или рисунком «осматривается» микроскопическим «электронным глазом». Диаметр участка изображения, воспринимаемого таким «глазом», составляет 1/20 миллиметра и соответствует диаметру человеческого волоса. Яркость считываемой в данный момент точки изображения кодируется двоичным числом и передается в компьютер. Для того чтобы осмотреть стандартный лист бумаги, «электронному глазу» приходится строку за строкой обходить его, передавая закодированную информацию об освещенности каждой точки изображения в компьютер.

### 1.2.6.5. Устройство вывода

*Монитор* – устройство вывода на экран текстовой и графической информации. Мониторы бывают *цветными* и *монохромными*. Они могут работать в одном из двух режимов: текстовом или графическом.

В *текстовом режиме* экран монитора условно разбивается на отдельные участки – *знакоместа*, чаще всего на 25 строк по 80 символов (знакомест). В каждое знакоместо может быть выведен один из 256 заранее определенных символов. В число этих символов входят большие и малые латинские буквы, цифры, определенные символы, а также псевдографические символы, используемые для вывода на экран таблиц и диаграмм, построения рамок вокруг участков экрана и так далее. В число символов, изображаемых на экране в текстовом режиме, могут входить и символы кириллицы.

На цветных мониторах каждому знакоместу может соответствовать свой цвет символа и фона, что позволяет выводить красивые цветные надписи на экран. На монохромных мониторах для выделения отдельных частей текста и участков экрана используется повышенная яркость символов, подчеркивание и инверсное изображение.

*Графический режим* предназначен для вывода на экран графической информации (рисунки, диаграммы, фотографии и т.п.). Разумеется в этом режиме можно выводить и текстовую информацию в виде различных надписей, причем эти надписи могут иметь произвольный шрифт, размер и др.

В графическом режиме экран состоит из точек, каждая из которых может быть темной или светлой на монохромных мониторах и одного или нескольких цветов – на цветном. Количество точек на экране называется *разрешающей способностью* монитора в данном режиме. Следует заметить, что разрешающая способность не зависит напрямую от размеров экрана монитора.

*Принтер* – устройство для вывода результатов работы компьютера на бумагу. Само название произошло от английского слова *printer*, означающего «печатник» (печатающий).

Первые принтеры создавали изображение из множества точек, получающихся под действием иголок, ударяющих через красящую ленту по бумаге и оставляющих на ней след. Иголки закреплены в печатающей головке и приводятся в движение электромагнитами. Сама же головка движется горизонтально, печатая строку за строкой. Количество иголок составляет 8 или 24 при одной и той же высоте печатающей головки. Во втором случае их делают тоньше, а получаемое изображение оказывается более «мелкозернистым».

Такой принтер преобразует электрические сигналы, выдаваемый компьютером, в движение иголок. Принтеры, использующие для получения изображения механический (ударный) принцип, называют *матричными*.

Матричные принтеры создают сильный шум и требуют частой замены красящей ленты, поэтому в 80-х годах был предложен другой способ печати на бумаге – *струйный*.

Принцип, лежащий в основе струйной печати с использованием жидких чернил, состоит в нанесении капелек чернил непосредственно на поверхность бумаги, пленки или ткани. Импульсная печатающая головка струйного принтера, подобно головке матричного принтера, состоит из вертикального ряда камер, способных нанести на бумагу одну или несколько вертикальных полосок. Число камер, входящих в состав головки, может достигать 48. Это позволяет получать очень качественное изображение.

Существуют как черно-белые, так и цветные струйные принтеры. Последние, кроме головки с черными чернилами, имеют еще печатную головку с чернилами трех цветов.

Кроме матричных и струйных принтеров, широкое распространение получили и, так называемые, *лазерные* принтеры. Принцип их работы достаточно сложен и требует глубокого знания физики, поэтому нами рассматриваться не будет. Эти принтеры при своей относительно высокой стоимости очень экономичны в эксплуатации и намного менее требовательны к качеству бумаги, по сравнению со струйными принтерами.

#### 1.2.6.6. Устройства связи

Устройства связи необходимы для организации взаимодействия отдельных компьютеров между собой, доступа к удаленным принтерам и подключения локальных сетей к общемировой сети Интернет. Примерами таких устройств являются *сетевые карты* (*ethernet cards*) и *модемы* (*modems*). Скорость передачи данных устройствами связи измеряется в битах в секунду (а также в Кбит/с и Мбит/с). Модем, используемый для подключения домашнего компьютера к сети Интернет, обычно обеспечивает пропускную способность до 56 Кбит/с, а сетевая карта – до 100 Мбит/с.

### 1.2.7. Программный принцип управления компьютером

В XIX веке английским математиком и инженером Чарльзом Бэббиджем был разработан проект вычислительной машины, которая пред-

назначалась для автоматического проведения длинных цепочек вычислений. Конструкция его аналитической машины включала 50 тысяч деталей: зубчатых колес, рычагов и пружин, взаимодействовавших определенным образом. Совершенствуя и уточняя конструкцию машины, Бэббидж первым смог выделить необходимые для ее работы части:

1) *устройство* для *хранения* чисел, как исходных, так и получающихся в результате вычисления;

2) специальный вычислительный блок – *процессор*;

3) *устройство* для *ввода* и *вывода* информации.

В качестве средства хранения информации в аналитической машине использовалась *перфокарта* – картонная прямоугольная пластина с рядами пробитых в ней дырочек. Каждый ряд состоял из двух частей, разделенных столбцом, содержащим отверстия во всех рядах. Первая часть представляла собой запись числа, вторая – код команды, указывающей, что делать с числом.

В созданной Бэббиджем аналитической машине присутствовала хранимая в памяти машины программа ее работы. Меняя программу (перфокарту), можно было изменять порядок вычислений, т.е. переходить от одной задачи к другой.

Главной особенностью конструкции этой машины является *программный принцип* работы.

Принцип программы, хранимой в памяти компьютера, считается важнейшей идеей современной компьютерной архитектуры. Суть идеи заключается в том, что

1) программа вычислений вводится в память ЭВМ и хранится в ней наравне с исходными числами;

2) команды, составляющие программу, представлены в числовом коде по форме ничем не отличающемся от чисел.

## 1.3. Компьютерные вирусы

Как уже отмечалось, компьютер работает исключительно под управлением программ (программного обеспечения). Это делает его по-настоящему универсальным устройством, которое может выполнять роль музыкального центра, телевизора, пишущей машинки и т.д. Программы пишут программисты и у некоторых из них появляется желание придумать что-то эдакое. Иногда это – невинные шалости, в других случаях они имеют явную зловещую направленность. До тех пор, пока человек, сидящий за компьютером, мог контролировать работу всех программ и знал, что и зачем он запустил, все было нормально. Но потом появились программы, которые, не спрашивая ничьего разрешения,

запускались, копировались в разные места диска и «заражали» другие программы (заменяли часть полезного кода рабочей программы своим или изменяли его). С этого момента и нужно начинать разговор о «компьютерных вирусах» [7].

Отдельно хочется подчеркнуть, что практически все вирусы функционируют в операционных системах семейства MS Windows и в MS DOS.

*Компьютерным вирусом* называется программа (некоторая совокупность выполняемого кода/инструкций), которая способна создавать свои копии (не обязательно полностью совпадающие с оригиналом) и внедрять их в различные объекты/ресурсы компьютерных систем, сетей и т.д. без ведома пользователя. При этом копии сохраняют способность дальнейшего распространения.

Компьютерные вирусы, как и биологические, ставят перед собой три задачи:

1) заразить;

2) выполнить;

3) размножиться.

Заражается компьютер «снаружи», когда человек запускает на исполнение некую программу, которая либо заражена вирусом (т.е. при ее выполнении запускается и вирус), либо сама является вирусом.

Поведение вирусов разнообразно. Некоторые вирусы просто «осыпали» буквы с монитора или рисовали безобидные рисунки. Такие считаются наиболее безвредными. Другие могут переименовывать файлы на диске, стирать их. Эти, без сомнения, гораздо опаснее. А вирус «Win95.CIH» может испортить микросхему BIOS компьютера. Трудно сказать, что хуже – потеря информации или выход из строя компьютера.

И, наконец, вирус размножается, т.е. дописывает себя везде, где он имеет шанс выполниться.

Есть вирусы, которые достаточно один раз запустить, после чего они постоянно при загрузке компьютера активно включаются в работу и начинают заражать все исполняемые файлы.

Появились вирусы, использующие возможности внутреннего языка программ серии Microsoft Office. Они содержаться в файлах, подготовленных в редакторе Word или в электронных таблицах Excel. Для заражения компьютера достаточно открыть такой документ.

Так как все больше людей использует Интернет, то последний все чаще становится рассадником заразы. Теперь достаточно зайти на некий сайт и нажать на кнопку формы, чтобы заполучить какой-нибудь вирус.

В последнее время широко распространился вид почтовых вирусов, играющих на любопытстве людей. Например, вам приходит письмо с признанием в любви и приложенными фотографиями. Первое движе-

ние – посмотреть содержимое письма. И как результат, – все фотографии и музыка на вашей машине пропали, а вместо них – злобный вирус «I Love You» (или подобный ему). Кроме того, он еще и пошлет себя всем, кто записан в вашей адресной книге.

Троянские программы отличаются от вирусов тем, что они вместо разрушительных действий собирают и отправляют по известным им адресам пароли и другую секретную информацию пользователя. Такая программа может давать злоумышленнику полный доступ к вашим программам и данным.

Методы борьбы с вирусами и троянцами описаны во многих местах. К сожалению, единственный действенный метод – не включать компьютер вовсе. Можно еще посоветовать ничего не устанавливать и ничего не запускать. Только тогда какой смысл иметь компьютер?

Поэтому широко используются *антивирусы* – программы, призванные обнаруживать и удалять известные им «нехорошие программы». Наиболее представительными являются DrWeb, Antiviral Tolkit Pro (AVP), ADInf. При использовании таких программ главное – постоянное обновление антивирусных баз.

И все-таки очень важно не запускать неизвестно что. Или установить *антивирусный монитор* (который отличается от *антивирусного сканера*, занимающегося тотальной проверкой файлов). Когда вы запускаете тот же DrWeb на проверку дисков – это антивирусный сканер. А в комплекте с ним идет некий Spider – вот это антивирусный монитор.

Однако при борьбе с вирусами не стоит впадать в крайность и стирать все подряд. При этом вы можете случайно удалить важные системные файлы, что приведет к невозможности работы на компьютере. На этом построено действие «психологических» вирусов, рассчитанных именно на то, что пользователь своими руками разрушит систему.

### 1.3.1. Основные признаки появления в системе вируса

Основными признаками появления в системе вируса являются:

1) замедление работы некоторых программ;

2) увеличение размеров файлов (особенно выполняемых), хотя это достаточно сложно заметить (попробуйте Adinf);

3) появление не существовавших ранее «странных» файлов, особенно в каталоге Windows или корневом;

4) уменьшение объема доступной оперативной памяти;

5) внезапно возникающие разнообразные видео и звуковые эффекты;

6) заметное снижение скорости работы в Интернете (вирус или троянец могут передавать информацию по сети);

7) жалобы от друзей (или провайдера) о том, что к ним приходят всякие непонятные письма – вирусы любят рассылать себя по почте.

В операционной системе Linux вирусы были выявлены только в лабораторных условиях. Несмотря на то, что некоторые образцы Linix-вирусов действительно обладали всеми необходимыми способностями к размножению и автономной жизни, ни один из них так и не был зафиксирован в «диком» виде.

Использование ОС Linux защищает от вирусов гораздо лучше, чем любые антивирусные программы в MS Windows.

### 1.3.2. Правовая охрана программ и GPL

Говоря о создании и распространении программного обеспечения, следует отметить две основных стратегии, применяемых в этой области. Одна – стратегия *copyright,* подразумевающая оплату при покупке каждой копии программного продукта и запрет на распространения этих копий. В законодательствах многих стран имеются законы, охраняющие авторское право на программные продукты и данные. Наиболее известен Digital Millennium Copyright Act (DMCA), принятый правительством США, хотя многие считают, что он уже стал орудием торможения развития компьютерной среды.

Но, программирование – это такая же наука, как и химия, физика, математика. Все достижения в этих областях обнародованы. Не нужно открывать еще раз теорему Пифагора и изобретать заново колесо. Если человек живет в обществе, то все его открытия должны стать достоянием этого общества, ведь именно так происходит прогресс. То же можно сказать и о программном обеспечении. Развитие программного обеспечения невозможно, если мы не можем разделить свои достижения с другими специалистами, чтобы они продолжили наше дело, чтобы развивали и исходили уже из того, что развили другие. Эта точка зрения отражена в лицензии GPL, в соответствии с которой разрабатывалась и развивалась ОС Linux. Говоря о такой стратегии часто используют термин *copyleft*. Свободное программное обеспечение часто более надежно, чем несвободное.

В семидесятых годах XX века программное обеспечение зачастую разрабатывалось свободными объединениями программистов и бесплатно передавалось другим нуждающимся в нем пользователям. Нередко этим занимались даже крупные фирмы. К 1983 году положение изменилось – наступила эра персональных компьютеров, коммерческие программы и операционные системы (в частности, DOS от Microsoft) начали свое победное шествие по миру. Чуть позже идея коммерциали-

зации проникла и в мир «больших» машин и «серьезного» программирования. Ричард Столлмен, один из основателей Unix, основал проект GNU (www.gnu.org), целью которого было вернуть прежние взаимоотношения производителей и потребителей программного обеспечения. GNU (расшифровывается как «GNU is not Unix») – не Unix, потому что GNU не ограничивает свободу.

В манифесте GNU отличию свободных программ от бесплатных уделено очень много места – по-русски же это можно сказать гораздо короче, поскольку эти понятия не обозначаются, как в английском, одним словом *free.* Получив в пользование или купив свободную программу, вы можете:

1) сколько угодно *копировать*, как угодно широко ее *распространять*;

2) *изменять* или совершенствовать ее исходный код (программа, распространяемая по «публичной лицензии» GNU, всегда поставляется вместе с исходным кодом разработчика – этой самой строго охраняемой и никогда не раскрываемой частью коммерческих программ);

3) свободно *распоряжаться* измененной версией – хоть раздавать ее даром, хоть запрашивать за нее миллиард.

Только на одну вещь пользователь такого программного обеспечения не имеет права ни в коем случае. Он не может при дальнейшем распространении *скрыть исходный код программы,* объявив себя его «владельцем», и остановить таким образом свободное совершенствование и развитие программы. Специально для таких программ проект GNU ввел в обиход понятие **copyleft** (в отличие от copyright, когда создатель продукта сохраняет на него практически все авторские и имущественные права при любых обстоятельствах – даже если и распространяет его совершенно бесплатно).

Итак, *свобода программного обеспечения* состоит из следующих пунктов:

1) свободы читать (изучать) код;

2) свободы писать (модифицировать) код;

3) свободы распространять (публиковать, тиражировать) код.

Очевидно, что проблемы пиратства в случае со свободными программами просто не существует.

## 1.4. Операционные системы и сети

Одной из первых операционных систем персонального компьютера была MS DOS. Лишенная графического интерфейса, обладающая очень ограниченными возможностями, она практически завершила свое суще-

ствование с появлением Windows. Сначала графическая оболочка для MS DOS, а затем полноценная система – MS Windows на некоторое время практически полностью захватила нишу ОС для персонального компьютера.

Почти одновременно с Windows появилась и начала завоевывать своих приверженцев ОС Linux, перенявшая от ОС UNIX идеологию командной строки. С течением времени под давлением требований пользователей Linux обогатился графическим интерфейсом, не только не уступающим, но во многом превосходящим возможности оконной системы Windows. Сейчас все большее число пользователей персонального компьютера предпочитают эту бесплатную, гармонично развивающуюся ОС программному обеспечению от фирмы Microsoft.

Принципы работы с графическими оболочками MS Windows и Linux практически одинаковы: окна, щелчки мыши, контекстные меню. Далее мы ознакомимся с особенностями вышеперечисленных ОС и научимся понимать различия в их функционировании.

### 1.4.1. Операционные системы

*Операционная система* – это программа, которая управляет аппаратными и программными средствами компьютера, предназначенными для выполнения задач пользователя. [12]

ЭВМ предоставляет различные ресурсы для решения задачи, но чтобы сделать эти ресурсы легкодоступными для человека и его программ, требуется операционная система. Она скрывает от пользователя сложные и ненужные подробности и предоставляет ему удобный интерфейс для работы. Операционная система осуществляет загрузку в оперативную память всех программ, передает им управление в начале их работы, выполняет различные действия по запросу выполняемых программ и освобождает занимаемую программами оперативную память при их завершении.

Кроме перечисленного выше, операционные системы могут предоставлять и другие возможности, делающие ЭВМ еще более удобной для использования: одновременное выполнение множества различных программ (мультизадачность); средства защиты информации, хранящейся на дисках ЭВМ; работа нескольких пользователей на одной ЭВМ (многопользовательский режим); возможность подключения ЭВМ к сети, а также объединение вычислительных ресурсов нескольких машин и совместное их использование (кластеризация).

Кроме операционных систем для работы необходимы некоторые другие компоненты. Среди них *базовая система ввода-вывода* (BIOS),

постоянно находящаяся в памяти компьютера. Эта система «встроена» в материнскую плату компьютера. Ее назначение состоит в выполнении элементарных действий, связанных с осуществлением операций ввода-вывода. BIOS содержит также тест функционирования компьютера, проверяющий работу памяти и устройств компьютера при включении электропитания. Кроме того, базовая система ввода-вывода содержит программу вызова загрузчика операционной системы.

*Загрузчик операционной системы* – это специальная программа, предназначенная для инициирования процесса загрузки операционной системы.

В настоящее время трудно себе представить работу на компьютере без использования операционной системы. Обзор операционных систем мы начнем с MS DOS – одной из первых ОС, завоевавших широкую популярность среди пользователей персональных ЭВМ. Затем рассмотрим версии Windows: от Windows 3.11 – графической оболочки для MS DOS, до современных ОС Windows 9X и Windows 2000. Наиболее полно будет рассмотрена ОС Linux, которая является UNIX-подобной ОС для персональных компьютеров. Эта система уже более семи лет является базовой при обучении студентов и школьников информатике и информационным технологиям в Московском государственном индустриальном университете.

#### 1.4.1.1. Операционная система MS DOS

**MS DOS** – первая операционная система для персональных компьютеров, которая получила широкое распространение. Со временем она была практически вытеснена новыми, современными операционными системами, типа Windows и Linux, но в ряде случаев MS DOS остается удобной и единственно возможной для работы на ЭВМ (устаревшая техника, давно написанное программное обеспечение и т.п.)

Работа пользователей с операционной системой DOS осуществляется с помощью командной строки, ведь собственный графический интерфейс у нее отсутствует. Предпринималось множество попыток упростить общение с системой и самое удачное решение предложил Питер Нортон (Peter Norton). У многих пользователей работа в операционной системе DOS ассоциируется именно с его программой – *Norton Commander*. Оболочка NC скрывает от пользователя множество неудобств, возникающих при работе с файловой системой MS DOS, например, такие, как необходимость набирать команды из командной строки. Простота и удобство в использовании – вот что делает оболочки типа NC популярными и в наше время (к ним можно отнести QDos,

PathMinder, XTree, Dos Navigator, Volkov Commander и др.). Принципиально отличаются от них графические оболочки Windows 3.1 и Windows 3.11. В них применяется концепция так называемых «окон», которые можно открывать, перемещать по экрану и закрывать.

В MS DOS используется файловая система **FAT.** Одним из ее недостатков являются жесткие ограничения на имена файлов и каталогов. *Имя* может состоять не более чем из восьми символов. *Расширение* указывается после точки и состоит из не более чем трех символов. Присутствие расширения в имени файла не является обязательным, оно добавляется для удобства, так как расширение позволяет узнать, какая программа создала его и тип содержимого файла. DOS не делает различий между одноименными строчными и прописными буквами. Кроме букв и цифр имя и расширение файла могут состоять из следующих символов: **-, _, $, #, &, @, !, %, (, ), {, }, ', ^** . Примеры имен файлов в MS DOS: doom.exe, referat.doc.

Так как MS DOS была создана довольно давно (известно, как стремительно развиваются и устаревают компьютеры и, как следствие, программы для них), она совершенно не соответствует требованиям, предъявляемым к современным операционным системам. Она не может напрямую использовать большие объемы памяти, устанавливаемые в современные ЭВМ. В файловой системе используются только короткие имена файлов (8 символов в имени и 3 в расширении), плохо поддерживаются разные устройства типа звуковых карт, видео-ускорителей и т.д.

В MS DOS совершенно не реализована мультизадачность, т.е. она не может естественным образом выполнять несколько задач (работающих программ) одновременно. Поэтому она не может использоваться в качестве основной операционной системы для полноценной многопользовательской работы в сети. MS DOS не имеет никаких средств контроля и защиты от несанкционированных действий программ и пользователя, что привело к появлению огромного количества так называемых вирусов.

Перечислим некоторые компоненты операционной системы MS DOS. *Дисковые файлы* IO.SYS и MSDOS.SYS (они могут называться и по-другому, например IBMBIO.COM и IBMDOS.COM для PC DOS) помещаются в оперативную память при загрузке и остаются в ней постоянно. Файл IO.SYS представляет собой дополнение к базовой системе ввода-вывода, а MSDOS.SYS реализует основные высокоуровневые услуги операционной системы.

*Командный процессор* DOS обрабатывает команды, вводимые пользователем. Командный процессор находится в дисковом файле COMMAND.COM на диске, с которого загружается операционная си-

стема. Некоторые команды пользователя, например type, dir или copy, командный процессор выполняет сам. Такие команды называются внутренними или встроенными. Для выполнения остальных (внешних) команд пользователя командный процессор ищет на дисках программу с соответствующим именем и, если находит ее, загружает в память и передает ей управление. По окончании работы программы командный процессор удаляет программу из памяти и выводит сообщение о готовности к выполнению команд (приглашение DOS).

*Внешние команды* DOS – это программы, поставляемые вместе с операционной системой в виде отдельных файлов. Эти программы выполняют действия обслуживающего характера, например форматирование дискет (format.com), проверку состояния дисков (scandisk.exe) и т.д.

*Драйверы устройств* – это специальные программы, которые дополняют систему ввода-вывода DOS и обеспечивают обслуживание новых или нестандартное использование имеющихся устройств. Например, с помощью драйвера DOS ramdrive.sys возможна работа с «электронным диском», т.е. частью памяти компьютера, с которой можно работать так же, как с диском. Драйверы помещаются в память компьютера при загрузке операционной системы, их имена указываются в специальном файле CONFIG.SYS. Такая схема облегчает добавление новых устройств и позволяет делать это, не затрагивая системные файлы DOS.

#### 1.4.1.2. Microsoft Windows

На смену операционной системе MS DOS с ее графическими оболочками Windows 3.1 и Windows 3.11 пришли полноценные операционные системы семейства **MS Windows** (сначала Windows 95, затем Windows 98 и Windows 2000). Их запуск, в отличие от Windows 3.1 и Windows 3.11, производится автоматически после включения компьютера (в том случае, если установлена только одна эта система).

В MS Windows для хранения файлов используется модификация файловой системы FAT – **VFAT**. В ней длина имен файлов и каталогов может достигать 256 символов. При указании имен прописные и заглавные буквы различаются.

В операционной системе Windows при работе с окнами и приложениями широко применяется манипулятор *мышь*. Обычно мышь используется для выделения фрагментов текста или графических объектов, установки и снятия флажков, выбора команд меню, кнопок панелей инструментов, манипулирования элементами управления в диалогах, «прокручивания» документов в окнах.

В Windows активно используется и правая кнопка мыши. Поместив кончик указателя над интересующим вас на экране объектом и сделав щелчок правой кнопкой мыши, вы можете раскрыть так называемое «контекстное меню», содержащее наиболее употребительные команды, применимые к данному объекту.

*Ярлыки* обеспечивают доступ к программе или документу из различных мест, не создавая при этом нескольких физических копий файла. На рабочий стол можно поместить не только пиктограммы (значки) приложений и отдельных документов, но и папок. *Папки* – еще одно название каталогов (directories).

Существенным нововведением в Windows 95 стала *Панель задач* (Taskbar). Несмотря на небольшие функциональные возможности, она делает наглядным механизм многозадачности и намного ускоряет процесс переключения между приложениями по сравнению с предыдущими версиями Windows. Внешне панель задач представляет полосу, обычно располагающуюся в нижней части экрана, на которой размещены кнопки приложений и кнопка *Пуск* (*Start*). В правой ее части обычно присутствуют часы и небольшие пиктограммы программ, активных в данный момент.

Рабочий стол Windows сконструирован так, чтобы максимально облегчить работу пользователя-новичка и в то же время предоставить максимальные возможности его настройки в соответствии с конкретными нуждами опытных пользователей.

При завершении работы нельзя просто выключить компьютер, не завершив работу системы по всем правилам, – это может привести к потере некоторых несохраненных данных. Для правильного завершения работы необходимо сделать следующее.

1. Сохранить данные во всех приложениях, с которыми вы работали.
2. Завершить работу всех ранее запущенных DOS-приложений.
3. Открыть меню кнопки *Пуск* и выбрать команду *Завершение работы* – на экране появится диалоговое окно.
4. Выбрать нужный вариант действий и нажать кнопку *Да.*

### 1.4.1.3.Операционная система Linux

**Linux** – это операционная система для IBM-совместимых персональных компьютеров и рабочих станций. Это многопользовательская ОС с сетевой оконной графической системой X Window System. ОС Linux поддерживает стандарты открытых систем и протоколы сети Интернет и совместима с системами Unix, DOS, MS Windows. Все компоненты системы, включая исходные тексты, распространяются с лицен-

зией на свободное копирование и установку для неограниченного числа пользователей.

Разработал эту операционную систему в начале 90-х годов тогда еще студент университета Хельсинки (Финляндия), Линус Торвальд при участии пользователей сети Интернет, сотрудников исследовательских центров, различных фондов и университетов (в том числе и МГИУ).

Будучи традиционной операционной системой, Linux (произносится «линукс», с ударением на первом слоге) выполняет многие из функций, характерных для DOS и Windows. Однако следует отметить, что эта ОС отличается особой мощью и гибкостью. Система Linux разрабатывалась как ПК-версия операционной системы Unix, которая десятилетиями используется на мэйнфреймах и мини-ЭВМ и является основной ОС для рабочих станций. Linux предоставляет в распоряжение пользователя ПК скорость, эффективность и гибкость Unix, используя при этом все преимущества персональных машин. При работе с мышью активно используются все три кнопки, в частности средняя кнопка используется для вставки фрагментов текста.

С экономической точки зрения Linux обладает еще одним весьма существенным достоинством – это бесплатная система. Linux распространяется по генеральной открытой лицензии GNU в рамках фонда свободного программного обеспечения (Free Software Foundation), что делает эту ОС доступной для всех желающих. Linux защищена авторским правом и не находится в общедоступном пользовании, однако открытая лицензия GNU это почти то же самое, что и передача в общедоступное пользование. Она составлена так, что Linux остается бесплатной и в то же время стандартизированной системой. Существует лишь один официальный вариант ядра Linux.

От Unix операционной системе Linux достались еще две замечательные особенности: она является *многопользовательской* и *многозадачной* системой. Многозадачность означает, что система может выполнять несколько задач одновременно. Многопользовательский режим означает, что в системе могут одновременно работать несколько пользователей, каждый из которых взаимодействует с ней через свой терминал. Еще одним из достоинств этой ОС является возможность ее установки совместно с Windows на один компьютер.

Linux способен любую персональную машину превратить в рабочую станцию. В наше время Linux является операционной системой для бизнеса, образования и индивидуального программирования. Университеты по всему миру применяют Linux в учебных курсах по программированию и проектированию операционных систем. Он стал незаме-

ним в широких корпоративных сетях, а также для организации Интернет-узлов и Web-серверов.

Современный Linux предоставляет возможность использовать несколько разновидностей графического интерфейса: *KDE* (K Desktop Environment), *GNOME* (GNU Network Model Environment) и другие. В каждой из этих оболочек пользователю предоставляется возможность работы сразу с несколькими рабочими столами (в то время как в MS Windows всегда один рабочий стол, который приходится загромождать окнами) [12].

## 1.5. Обработка документов

Одной из наиболее распространенных функций современного персонального компьютера является подготовка разнообразных текстовых документов. В данном разделе рассматриваются программные продукты, функционирующие в MS Windows и ОС Linux и предназначенные для работы с текстами. Мы ознакомимся с принципами создания и редактирования как простых, так и более сложных документов.

Различают две основные группы программ подготовки текстовых документов: *текстовые редакторы* и *текстовые процессоры.*

**Текстовыми редакторами**, в основном, называют программы, создающие текстовые файлы без элементов форматирования (т.е. не позволяющие выделять части текста различными шрифтами и гарнитурами). Редакторы такого рода незаменимы при создании текстов компьютерных программ.

**Текстовые процессоры** умеют форматировать текст, вставлять в документ графику и другие объекты, не относящиеся к классическому понятию «текст». Следует отметить условность такого разделения – разнообразие программ для обработки текста позволяет найти редактор с любым набором функций.

Некоторые текстовые процессоры являются так называемыми *WYSIWYG*-редакторами. Название получено по первым буквам фразы *What You See Is What You Get* – то, что ты видишь, есть то, что ты получишь. Когда говорят, что это WYSIWYG-редактор, то гарантируют полное соответствие внешнего вида документа на экране компьютера и его печатной копии. К редакторам такого типа относятся Word и StarWriter.

Некоторые современные редакторы поддерживают концепцию «почти» WYSIWYG. Вид документа на экране при этом немного отличается от того, как будет выглядеть напечатанный документ, но делается это специально с целью более эффективного использования рабочего окна

документа. Примерами «почти» WYSIWYG-редакторов являются Netscape Composer и KLyX [4].

### 1.5.1. Текстовый процессор MS Word

*Microsoft Word* – это мощный текстовый редактор, получивший широчайшее распространение в среде Windows. Он является удобным инструментом для подготовки разнообразных писем, деловой документации, отчетов. С его помощью удобно создавать как бланки и анкеты, так и статьи, брошюры.

В основе оформления документов в Word лежит система шаблонов и стилей форматирования, которые позволяют достичь единства оформления большинства документов. Word относится к WYSIWYG-редакторам: напечатанный документ выглядит так же, как и на экране (рис. 6).

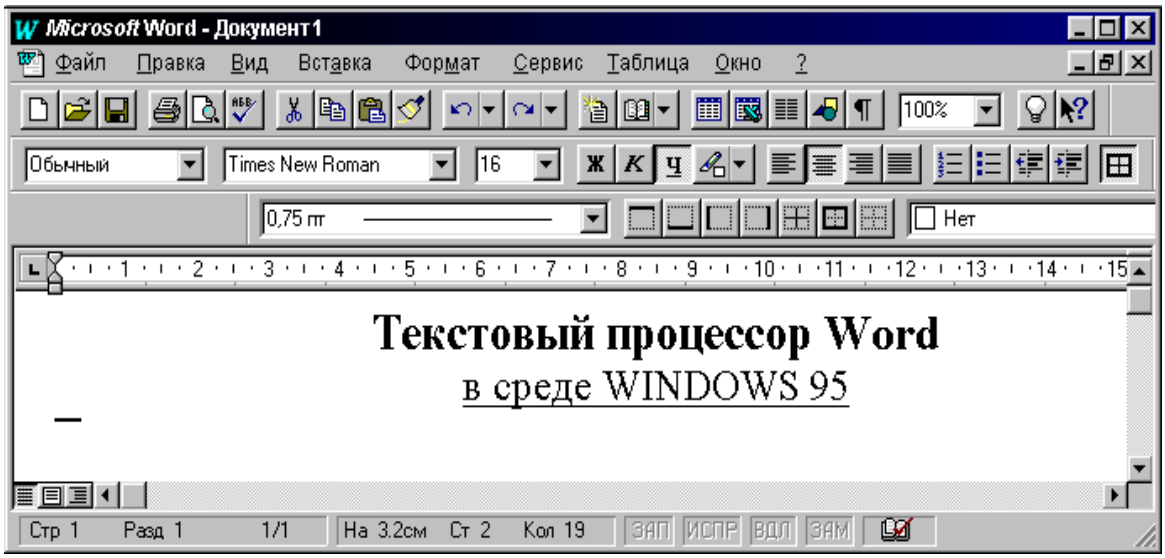



Рис. 6 . Текстовый редактор Microsoft Word

Word по умолчанию сохраняет текстовые файлы в собственном двоичном формате MS Word (соответствующее расширение файла – .doc). Текстовая версия этого формата – RTF-формат (Rich Text Format), документирована фирмой Microsoft и поддерживается текстовыми процессорами некоторых других фирм. Из-за своей текстовой структуры формат RTF намного безопаснее с точки зрения распространения компьютерных вирусов, в то время как файлы формата DOC являются одним из инструментов передачи вирусов между компьютерами. Последние версии процессора могут сохранять файл в формате, включающим элементы разметки гипертекста (.html или .htm).

В текстовом процессоре Word используется несколько *панелей инструментов,* которые облегчают работу с документом. Рассмотрим эти панели инструментов и их назначение.

1. **Стандартная** содержит кнопки команд, служащих для работы с файлом и с буфером обмена.
2. **Форматирование** служит для форматирования текста.
3. **VisualBasic** предназначена для создания программ на VisualBasic, совмещенных с документом Word.
4. **Web** преобразует документ Word в Web страницу, т.е. файл, содержащий разметку языка HTML.
5. **WordArt** содержит кнопки вызова команд создания фигурного текста.
6. **Автотекст** автоматическая замена фрагментов или быстрая вставка часто повторяющегося фрагмента.
7. **Базы Данных** предназначена для создания и работы с базами данных, которые используются в таблицах документа.
8. **Настройка Изображения** содержит кнопки вызова команд, служащих для работы с графическим изображением.
9. **Рецензирование** предназначена для вставки сообщений и рецензий.
10. **Рисование** предназначена для вставки в документ графических объектов.
11. **Таблицы и границы** служит для обрамления таблиц.
12. **Формы** содержит кнопки создания форм, таблиц, списков, полей ввода при работе с базами данных.
13. **Элементы управления** вставляет кнопки, переключатели и другие элементы VisualBasic в документ.
14. **Колонтитулы** при помощи этой панели инструментов можно установить различные верхние и нижние колонтитулы.
15. **Главный документ** служит для разработки структуры главного документа.
16. **Настройка объема** устанавливает варианты объема текста в документе.
17. **Настройка тени** позволяет добавить тень как к тексту, так и к рисункам.
18. **Структура** предназначена для установки структуры документа.
19. **Создание и сохранение документа** Word предоставляет несколько шаблонов документов, которые позволят вам создавать специализированные документы, такие как письма или статьи. Один и тот же шаблон можно использовать много раз. Для создания, сохранения,

открытия и закрытия документа можно воспользоваться пунктами меню **Файл** или кнопками на панели инструментов «Стандартная».

Текстовый редактор Word может сохранять документы в некоторых других форматах. Для сохранения документа в формате, отличном от Microsoft Word, нужно в окне сохранения документа в списке «Тип файла» выбрать требуемый формат файла.

### Форматирование текста

Одной из важнейших особенностей текстовых процессоров, в том числе и программы Word, является возможность разнообразного *форматирования* текста. Различают три вида форматирования.

1. *Форматирование символов* – при форматировании символов речь идет, в основном, об изменении шрифта.

2. *Форматирование абзацев* – под форматированием абзацев понимается изменение размеров полей отдельных абзацев в тексте, изменение интервалов между строками и выравнивание абзацев.

3. *Форматирование страниц* – под форматированием страниц понимается выбор размера, ориентации и размеры полей страниц.

Форматирования символов включает в себя изменение цвета, размера, стиля написания текста. Для изменения стиля написания символов могут использоваться кнопки, расположенные на панели инструментов «Форматирование».

Под абзацем в Word понимается часть документа, за которой следует маркер абзаца. При вводе текста абзац всегда заканчивают нажатием на клавишу Enter. Если же требуется перейти на следующую строку без выделения нового абзаца, используйте комбинацию Shift + Enter.

*Процесс форматирования* абзацев включает в себя:

1) выравнивание абзацев;
2) установку абзацных отступов;
3) установку отступа первой строки абзаца;
4) установку расстояния между строками;
5) установку расстояния между абзацами;
6) контроль положения абзаца на странице.

Выравнивание абзацев устанавливается при помощи панели инструментов «Форматирование». По умолчанию Word выравнивает все абзацы влево, при этом строки начинаются как бы с одной вертикальной линии. Можно осуществлять выравнивание как по центру, так и по правому краю, а также блочное выравнивание – выравнивание по ширине листа.

Для установки отступов абзаца и первой строки можно использовать горизонтальную линейку. Выделите абзацы, для которых нужно установить отступ, и с помощью мыши переместите маркеры отступов абзацев и первой строки (бегунки), расположенные на горизонтальной линейке, в нужную позицию. Основные параметры отступов абзаца можно также установить в диалоговом окне **Абзац**, для чего необходимо из меню **Формат** вызвать пункт **Абзац**.

По умолчанию Word устанавливает расстояние между строками в один интервал, т.е. это расстояние точно соответствует высоте одной строки. Однако оно может составлять полтора, два и более интервалов. Расстояние между строками устанавливается в диалоговом окне **Абзац** в списке «Межстрочный».

Любой документ, напечатанный на бумаге, имеет поля. Word позволяет установить величину для каждого из четырех полей (верхнего, нижнего, правого и левого) по отдельности. Для этого следует выбрать пункт **Параметры страницы** из меню **Файл** и ввести требуемые величины в соответствующие поля (рис. 7).

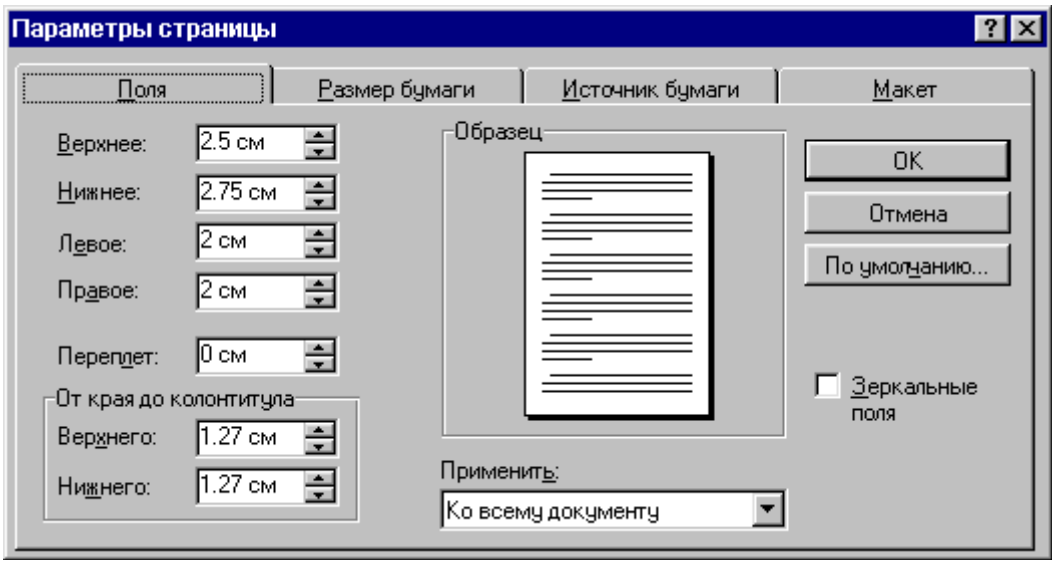



Рис. 7. Окно «Параметры страницы»

Для того чтобы обеспечить автоматическое выполнение переносов слов во всем документе, нужно не только включить режим переноса слов (для чего в пункте **Перенос слов** меню **Сервис** нужно поставить отметку **Автоматический перенос слов в документе**), но также убедиться, что перенос слов не заблокирован ни в одном абзаце. Чтобы снять блокировку слов в отдельном абзаце в меню **Правка** выберите

**Выделить все**, затем в меню **Формат** выберите **Абзац**, щелкните на вкладке **Положение на странице** и удалите отметку **Без переноса слов**.

Для проверки орфографии документа используйте клавишу **F7**. Можно также использовать кнопку **Орфография** на панели инструментов «Стандартная».

## Колонтитулы

При создании многостраничных документов почти всегда в верхней или нижней части страницы помещают дополнительную информацию, называемую колонтитулами. В колонтитулы можно поместить заголовок документа, номер страницы, дату, время и некоторые другие параметры. Меню для работы с колонтитулами вызывается путем выбора пункта **Колонтитулы** из меню **Вид**.

Одна из причин применения разбиения документов на разделы – это необходимость иметь разные верхние и нижние колонтитулы в различных частях документа. Выделив каждую часть в отдельный раздел, вы получаете возможность установить для каждого из разделов колонтитулы, отличающиеся друг от друга.

Чаще всего в колонтитулы помещают номера страниц документа. Для быстрой нумерации страниц выберите пункт **Номера страниц** из меню **Вставка**. Если на первой странице не должно быть номера, то удалите отметку в поле **Номер на первой странице**. В диалоговом окне **Формат номера страницы** Word предлагает вам выбрать один из вариантов расположения номеров страниц.

## Многоколоночный текст

Word позволяет разместить текст в несколько колонок, при этом он располагает средствами, способными установить следующие виды документа с использованием колонок:

1) создание колонок одинаковой ширины,
2) создание двух колонок разной ширины,
3) создание нескольких колонок разной длины,
4) свободный выбор позиции начала новой колонки,
5) изменение ширины и расстояния между колонками,
6) изменение количества колонок в тексте,
7) добавление вертикальной линии между колонками,
8) выравнивание длины колонок,
9) размещение в колонке графических иллюстраций.

Для изменения количества колонок в документе нужно воспользоваться кнопкой **Колонки** на панели инструментов «Стандартная» или

выбрать пункт **Колонки** из меню **Формат**, после чего в окне **Колонки** устанавливаются основные параметры (рис. 8).

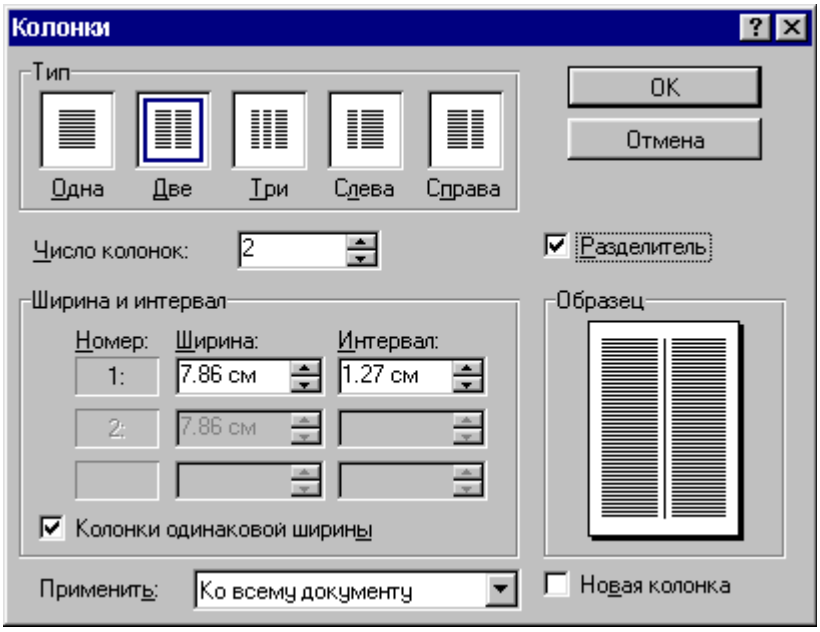



Рис. 8. Окно «Колонки» стандартной панели инструментов

**Пример 1.** Создание документа, название которого размещено через всю полосу, а основной текст в две и более колонок. Для этого вставим маркер конца раздела между заголовком и основным текстом при помощи пункта **Разрыв** из меню **Вставка (**рис. 9).

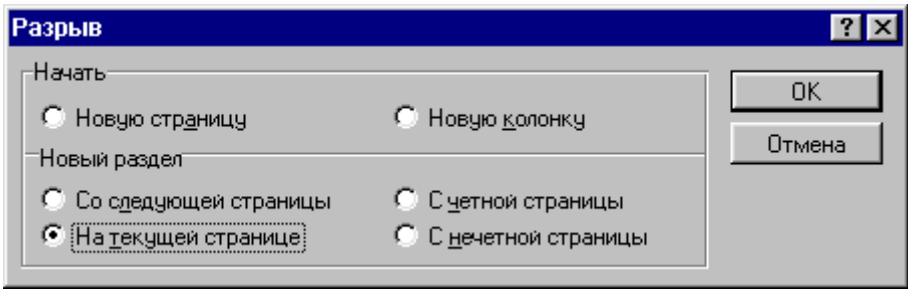



Рис. 9. Меню «Вставка» стандартной панели инструментов

В диалоговом окне **Разрыв** установим переключатель *На текущей странице*. В результате этих действий на той строке, в которой был установлен курсор, появится маркер конца раздела. Теперь документ состоит из двух разделов, каждый из которых можно форматировать независимо друг от друга.

## Списки

Для того чтобы создать нумерованный, маркированный (не нумерованный) или многоуровневый список, достаточно выделить все абзацы, которые нужно оформить в виде списка и воспользоваться пунктом **Список** из меню **Формат**. Word выведет на экран диалоговое окно **Список** (рис. 10).

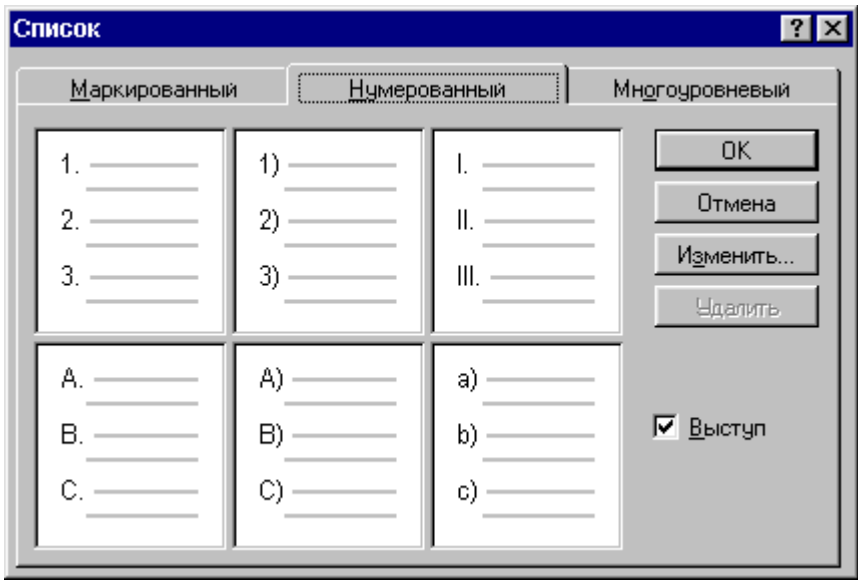



Рис. 10. Диалоговое окно «Список» меню «Формат»

При необходимости можно выбрать нестандартный символ маркировки. Word вставит символ маркировки перед каждым новым абзацем. Для увеличения и уменьшения вложенности списка можно воспользоваться кнопками «**Увеличить отсту**» и «**Уменьшить отступ**» на панели инструментов «Стандартная».

## Таблицы

Текстовый процессор Word позволяет вставлять таблицу в документ. Для этого служит меню **Таблица**. Также можно воспользоваться кнопкой **Вставка таблицы** на панели инструментов «Стандартная». На экране появится диалоговое окно **Вставка таблицы** (рис. 11), в котором можно установить количество строк и столбцов создаваемой таблицы. Для выбора одного из стандартных видов оформления таблицы можно воспользоваться кнопкой **Автоформат**, при нажатия на которую Word выведет на экран диалоговое окно **Автоформат таблицы**. Оформление таблицы осуществляется при помощи панели инструментов «Обрамление».

Установку ширины столбцов (строк) можно регулировать при помощи маркера границы столбца (строки) или при помощи пункта **Высота и ширина ячейки** меню **Таблица**, который предоставляет дополнительные возможности по сравнению с установкой ширины столбцов путем установки маркера границы столбца (строки). Во-первых, можно задать ширину столбцов с большей точностью, а во-вторых, эта вкладка предоставляет гораздо больше возможностей по управлению шириной столбцов.

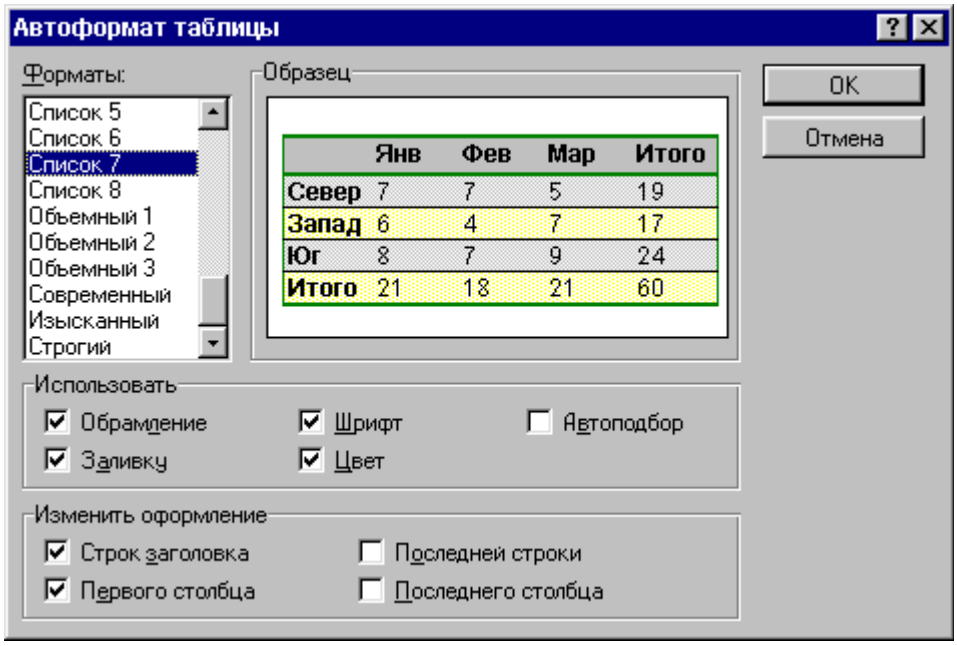



Рис. 11. Диалоговое окно «Вставка таблицы» меню «Таблица»

Для вставки нового столбца (строки) в любом месте таблицы вы можете воспользоваться пунктом «**Вставить столбец (строку)**» из меню «**Таблица**». Выделим столбец и выполним команду вставки столбца; новый столбец будет вставлен *слева* от исходного. Вставка строки *над* исходной осуществляется аналогично.

Для того чтобы удалить из таблицы целые столбцы (строки), выделите эти столбцы (строки) и выберите пункт **Удалить столбцы (строки)** из меню **Таблица** или пункт **Вырезать** из меню **Правка**.

В процессе редактирования таблицы может понадобиться объединение или разбивка ячеек. Для объединения ячеек необходимо их выделить и выполнить команду **Объединение ячеек** из меню **Таблица**. Для разбиения ячейки нужно выделить ее, выполнить команду **Разбить ячейку** из меню **Таблица**, и в появившемся диалоговом окне указать количество столбцов, на которые будет разбита ячейка.

**Пример 2.** Процесс создания таблицы следующего вида, представленного на рис. 12.


Рис. 12. Пример таблицы

Выполним пункт **Вставить** из меню **Таблица**. В появившемся диалоговом окне установим количество строк – 5 и количество столбцов – 3. Выделим первые две ячейки первой строки таблицы и выполним команду **Объединить ячейки**. Таким же образом объединим ячейки 2–3 строки 4 и ячейки 1–3 строки 5. После этого при помощи панели инструментов «Обрамление» оформим таблицу так, чтобы 1-я ячейка 3-й строки не имела верхней и нижней линии, внешний контур таблицы был нарисован двойной толстой линией, а внутренние линии были двойными тонкими. Заполним таблицу соответствующим образом. Выделим ячейки, которым хотим изменить цвет, и выполним пункт **Обрамление и заливка** из меню **Формат**. Word выведет на экран диалоговое окно **Обрамление и заливка**, в котором на вкладке *Заливка* можно выбрать требуемый цвет ячейки.

В документах Word можно использовать формулы, подсчитывающие сумму значений чисел в строке или столбце таблицы. Для этого установите точку вставки в ячейку, в которую вы хотите ввести формулу, и в меню **Таблица** выберите **Формула**. По умолчанию суммируются все значения, находящиеся выше точки вставки формулы для суммирования по столбцу или левее при суммировании по строке. Можно изменить диапазон ячеек для суммирования, указав явно диапазон их *имен*. Имя ячейки формируется из буквы, обозначающей столбец, и числа, задающего строку. Интервал суммирования задается путем ввода имен первой и последней ячеек, разделенных двоеточием, например «A2:B6». Нажатие на клавишу F9 обновит сумму после изменения значений в ячейках.

### Вставка графики

В документы Word может быть импортирована графика самых разных форматов. Среди них широко распространенные форматы – BMP и PCX, а также TIF, EPS, GIF, PIC и другие. Для работы с этими

форматами Word использует графические фильтры-программы, позволяющие интерпретировать и отображать на экране графику.

Word позволяет вам легко поместить графический рисунок в документ. Для этого нужно вызвать пункт **Рисунок** из меню **Вставка**, после чего в появившемся диалоговом окне выбрать рисунок из списка стандартных или указать свой.

Для изменения масштаба и положения рисунка необходимо поместить рисунок в кадр. Это можно сделать при помощи пункта меню **Кадр** из меню **Вставка**. Для работы с кадрами наиболее удобен режим просмотра страницы. В этом случае иллюстрация окажется вставленной в невидимую рамку, которую можно произвольно перемещать на экране. После помещения рисунка в кадр нужно выполнить пункт **Кадр** из меню **Формат**. Word выведет на экран диалоговое окно **Кадр (**рис. 13), в котором вы можете указать масштаб рисунка, расстояние между текстом и графикой, обрезать графическое изображение по краям или оставить вокруг него свободное пространство. В поле «Обтекание текста» можно установить режим обтекания рисунка текстом.

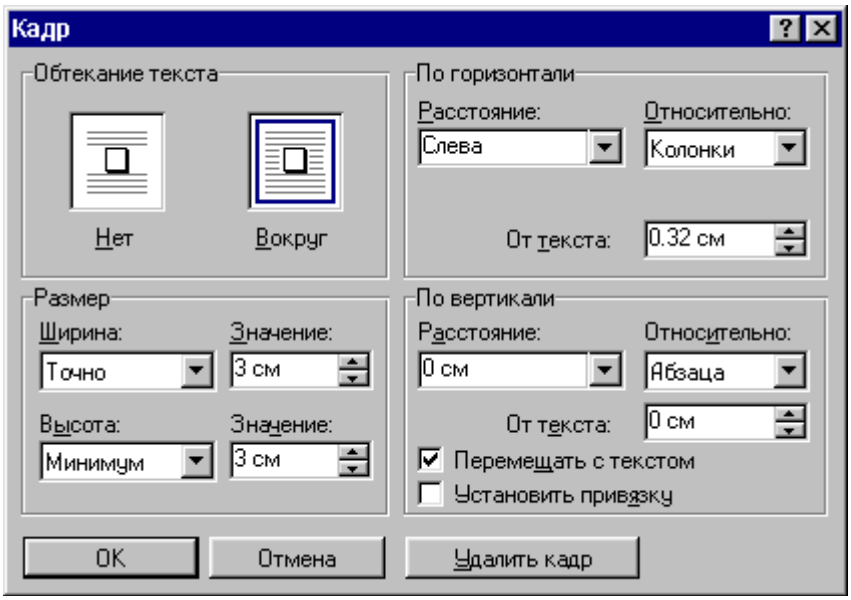



Рис. 13. Диалоговое окно «Кадр»

В состав редактора Word включены средства, которые позволяют создавать рисунки, состоящие из линий и геометрических фигур. Они могут быть успешно использованы для оформления приглашений, фирменных знаков, рекламных проспектов. Для создания таких иллюстраций служит панель инструментов «Рисование». С ее помощью можно легко и быстро создать простой рисунок.

## Текстовые эффекты

При помощи Microsoft WordArt могут быть вставлены в документ объекты, содержащие преобразованный текст. Можно изменять форму надписей, растягивать их, оттенять. Для запуска этого приложения нужно вызвать пункт **Объект** из меню **Вставка** и в списке диалогового окна выбрать строку **Microsoft WordArt**. Word выведет на экран окно программы WordArt.

У верхнего края окна расположена панель управления. Значения всех кнопок на панели управления WordArt, которые выполняются при их нажатии, приведены в табл. 1.

Таблица 1

Значения кнопок панели управления WordArt

| Название кнопки | Функции |
|---|---|
| Форма | Выбор формы изображения текстового фрагмента |
| Шрифт | Выбор вида шрифта |
| Размер шрифта | Выбор размера шрифта |
| Полужирный | Установка полужирного начертания |
| Курсив | Установка курсивного начертания |
| Равная высота | Установка одинаковой высоты прописных и строчных букв |
| Поворот | Поворот букв на 90 градусов |
| Растянуть | Растягивание текста по горизонтали и вертикали |
| Выравнивание | Центрирование текста |
| Интервал между символами | Выбор расстояния между отдельными символами |
| Вращение | Выбор параметров поворота текста |
| Узор | Выбор типа узора или цвета для букв текста |
| Тень | Наложение тени на текст |
| Граница | Выбор толщины линии, ограничивающей текст |

## Включение математических формул

Большинство математических и других научных статей включают в себя теоремы, доказательства и уравнения, использующие особую математическую символику. Математические формулы, как правило, содержат многочисленные специальные символы и конструкции, а также используют особые правила расположения составных частей, почти не применяющиеся при работе с обычным текстом. Microsoft Word поставляется вместе с приложением **Microsoft Equation**, которое позволяет создавать математические формулы на экране и выводить их на печать.

Для запуска Microsoft Equation нужно вызвать пункт **Объект** из меню **Вставка**, в появившемся диалоговом окне **Вставка объекта** вы-

брать вкладку «Создание», где в списке «Тип объекта» указать **Microsoft Equation**.

Панель инструментов «Формула» состоит из двух рядов кнопок (рис. 14).

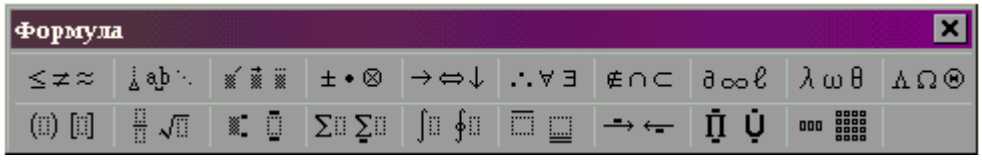



Рис. 14. Панель инструментов «Формула»

Назначения кнопок панели инструментов «Формула» в порядке следования построчно и слева направо представлены в табл. 2.

Таблица 2

Значения кнопок панели инструментов «Формула»

| Вид кнопки | Значение |
|---|---|
| ≤ ≠ ≈ | Символы отношений, такие как равно или приблизительно равно |
| ∴ ∀ ∃ | Области и эллипсы |
| ∉ ∩ ⊂ | Символы, подобные знакам дифференцирования или векторам |
| ± • ⊗ | Математические операции, такие как знаки умножения и деления |
| → ⇔ ↓ | Стрелки |
| ∴ ∀ ∃ | Логические символы, такие как кванторы |
| ∉ ∩ ⊂ | Символы теории множеств, такие как пересечения или объединения множеств |
| ∂ ∞ ℓ | Разнообразные символы, такие как бесконечность и градус |
| λ ω θ | Строчные символы греческого алфавита |
| Λ Ω ⊛ | Заглавные символы греческого алфавита |
| (▯) [▯] | Шаблоны для заключения вводимых объектов в круглые, квадратные или фигурные скобки |
| ▯/▯ √▯ | Шаблоны для дробей и корней |
| ▯ ▯ | Шаблоны для вставки верхних или нижних индексов |
| Σ▯ ∑▯ | Шаблоны для суммирования |
| ∫▯ ∮▯ | Шаблоны для интегралов |
| ▯ ▯ | Шаблоны для подчеркивания и надчеркивания |
| ▯→ ←▯ | Шаблоны для надписей под или над стрелками |
| ∏▯ ∐▯ | Шаблоны для представления выражений теории множеств |
| ▯▯ ▯▯ | Шаблоны для матриц |

## 1.6 Вопросы для самоконтроля

1. Поколение ЭВМ, в котором в качестве элементной базы процессора использовались транзисторы.
2. К какому типу составной части относится монитор?
3. Часть ЭВМ, где находится арифметико-логическое устройство (АЛУ)
4. Составная часть ЭВМ, контролирующая действия всех устройств компьютера и координирующая выполнение программ.
5. Возможность операционной системы ЭВМ, которая заключается в возможности подключения ЭВМ к сети, а также объединении вычислительных ресурсов нескольких машин и совместное их использование.
6. Операционная система, которая используется на мэйнфреймах и мини-ЭВМ и является основной ОС для рабочих станций.
7. Система, которая управляет аппаратными и программными средствами компьютера.
8. Перечислите операции с файлами.
9. Панель инструментов в текстовом процессоре Word, которая содержит кнопки вызова команд создания фигурного текста.
10. Форматы, в которых сохраняются текстовые документы Word .
11. Процесс изменение шрифта текста.
12. Перечислите задачи, которые ставят перед собой компьютерные вирусы.
13. Чем троянские программы отличается от вируса?
14. Последовательность выполнения этапов при работе ЭВМ.
15. Комплекс программ, предназначенных для запуска других программ на исполнение, организации диалога с пользователем, распределения оперативной памяти и т.д.

# 2. АРИФМЕТИЧЕСКИЕ ОСНОВЫ ПОСТРОЕНИЯ ЭВМ

## 2.1. Единицы измерения информации

Решая различные задачи, человек вынужден использовать информацию об окружающем нас мире. И чем более полно и подробно человеком изучены те или иные явления, тем подчас проще найти ответ на поставленный вопрос. Так, например, знание законов физики позволяет создавать сложные приборы, а для того, чтобы перевести текст на иностранный язык, нужно знать грамматические правила и помнить много слов.

Часто приходится слышать, что то или иное сообщение несет мало информации или, наоборот, содержит исчерпывающую информацию. При этом разные люди, получившие одно и то же сообщение (например, прочитав статью в газете), по-разному оценивают количество информации, содержащейся в нем. Это происходит оттого, что знания людей об этих событиях (явлениях) до получения сообщения были различными. Поэтому те, кто знал об этом мало, сочтут, что получили много информации, те же, кто знал больше, чем написано в статье, скажут, что информации не получили вовсе. Количество информации в сообщении, таким образом, зависит от того, насколько ново это сообщение для получателя.

Однако иногда возникает ситуация, когда людям сообщают много новых для них сведений (например, на лекции), а информации при этом они практически не получают (в этом нетрудно убедиться во время опроса или контрольной работы). Происходит это от того, что сама тема в данный момент слушателям не представляется интересной.

Итак, *количество информации* зависит от *новизны* сведений об интересном для получателя информации явлении. Иными словами, неопределенность (т.е. неполнота знания) по интересующему нас вопросу с получением информации уменьшается. Если в результате получения сообщения будет достигнута полная ясность в данном вопросе (т.е. неопределенность исчезнет), говорят, что была получена *исчерпывающая информация*. Это означает, что необходимости в получении дополнительной информации на эту тему нет. Напротив, если после получения сообщения неопределенность осталась прежней (сообщаемые сведения или уже были известны, или не относятся к делу), значит, информации получено не было (*нулевая информация*).

Если подбросить монету и проследить, какой стороной она упадет, то мы получим определенную информацию. Обе стороны монеты «равноправны», поэтому одинаково вероятно, что выпадет как одна, так и другая сторона. В таких случаях говорят, что событие несет информацию в *1 бит*. Если положить в мешок два шарика разного цвета, то, вытащив вслепую один шар, мы также получим информацию о цвете шара в 1 бит [4].

Единица измерения информации называется *бит* (bit) – сокращение от английских слов *binary digit,* что означает двоичная цифра.

В компьютерной технике бит соответствует физическому состоянию носителя информации: намагничено – не намагничено, есть отверстие – нет отверстия. При этом одно состояние принято обозначать цифрой 0, а другое – цифрой 1. Выбор одного из двух возможных вариантов позволяет также различать логические истину и ложь. Последовательностью битов можно закодировать текст, изображение, звук или какую-либо другую информацию. Такой метод представления информации называется *двоичным кодированием (binary encoding)*.

В информатике часто используется величина, называемая **байтом** (byte) и равная 8 битам. И если бит позволяет выбрать один вариант из двух возможных, то байт, соответственно, 1 из 256 ($2^8$). В большинстве современных ЭВМ при кодировании каждому символу соответствует своя последовательность из восьми нулей и единиц, т.е. байт. Соответствие байтов и символов задается с помощью таблицы, в которой для каждого кода указывается свой символ. Так, например, в широко распространенной кодировке Koi8-R буква «М» имеет код 11101101, буква «И» – код 11101001, а пробел – код 00100000.

Наряду с байтами для измерения количества информации используются более крупные единицы:

1 Кбайт (один килобайт) = $2^{10}$ байт = 1024 байта;

1 Мбайт (один мегабайт) = $2^{10}$ Кбайт = 1024 Кбайта;

1 Гбайт (один гигабайт) = $2^{10}$ Мбайт = 1024 Мбайта.

**Пример 3.** Книга содержит 100 страниц; на каждой странице – 35 строк, в каждой строке – 50 символов. Рассчитаем объем информации, содержащийся в книге.

Страница содержит 35 × 50 = 1750 байт информации. Объем всей информации в книге (в разных единицах):

1750 × 100 = 175000 байт.

175000 / 1024 = 170,8984 Кбайт.

170,8984 / 1024 = 0,166893 Мбайт.

## 2.2. Системы счисления

*Система счисления* – принятый способ записи чисел и сопоставления этим записям реальных значений. Все системы счисления можно разделить на два класса: *позиционные* и *непозиционные*. Для записи чисел в различных системах счисления используется некоторое количество отличных друг от друга знаков. Число таких знаков в позиционной системе счисления называется *основанием системы счисления*.

Некоторые системы счисления представлены в табл. 3.

Таблица 3

Примеры систем счисления

| Основание | Система счисления | Знаки |
|---|---|---|
| 2 | Двоичная | 0, 1 |
| 3 | Троичная | 0, 1, 2 |
| 8 | Восьмеричная | 0, 1, 2, 3, 4, 5, 6, 7 |
| 10 | Десятичная | 0, 1 … 9 |
| 16 | Шестнадцатеричная | 0, 1, …, A, B,C, D, E, F |

### Непозиционные системы счисления

Кроме позиционных систем счисления существуют такие, в которых значение знака не зависит от того места, которое он занимает в числе. Такие системы счисления называются *непозиционными* (например, римская). В этой системе счисления используется семь знаков:

I (1),V (5), X(10), L (50), C (100), D(500), M(1000).

**Пример 4.** Непозиционная системы счисления.

LII (52)  DLV (555)

Недостатки непозиционных систем счисления: отсутствие формальных правил записи чисел и, соответственно, арифметических действий над ними.

Эти системы счисления представляют лишь исторический интерес.

### Позиционные системы счисления

В *позиционных* системах счисления величина, обозначаемая цифрой в записи числа, зависит от ее позиции. Количество используемых цифр называется *основанием* системы счисления. Место каждой цифры в числе называется *позицией.* Первая известная нам система, основанная на позиционном принципе – шестидесятеричная вавилонская. Цифры в ней были двух видов, одним из которых обозначались единицы, другим – десятки. Следы вавилонской системы сохранились до наших дней в способах измерения и записи величин углов и промежутков времени.

В позиционной системе счисления число может быть представлено в виде произведений коэффициентов на степени основания системы счисления:

$$A_n A_{n-1} A_{n-2} \ldots A_1 A_0 , A_{-1} A_{-2} \ldots =$$

$$A_n * B^n + A_{n-1} * B^{n-1} + \ldots + A_1 * B^1 + A_0 * B^0 + A_{-1} * B^{-1} + A_{-2} * B^{-2} + \ldots$$

Таким образом, значение каждого знака в числе зависит от позиции, которую занимает знак в записи числа. Именно поэтому такие системы счисления называются позиционными.

**Пример 5.** Позиционные системы счисления.

$23{,}43_{(10)} = 2*10^1 + 3*10^0 + 4*10^{-1} + 3*10^{-2}$

$692_{(10)} = 6*10^2 + 9*10^1 + 2*10^0$

$1101_{(2)} = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

$341{,}5_{(8)} = 3*8^2 + 4*8^1 + 1*8^0 + 5*8^{-1}$

$A1F{,}4_{(16)} = A*16^2 + 1*16^1 + F*16^0 + 4*16^{-1}$

При работе с компьютерами приходится параллельно использовать несколько позиционных систем счисления (чаще всего двоичную, десятичную и шестнадцатеричную). Поэтому большое практическое значение имеют процедуры перевода чисел из одной системы счисления в другую.

Чтобы перевести целую часть числа из десятичной системы в систему с основанием *B*, необходимо разделить её на *B*. Остаток даст младший разряд числа. полученное при этом частное необходимо вновь разделить на *B* – остаток даст следующий разряд числа и т.д. Для перевода дробной части её необходимо умножить на *B*. Целая часть полученного произведения будет первым (после запятой, отделяющей целую часть от дробной) знаком. Дробную же часть произведения необходимо вновь умножить на *B*. Целая часть полученного числа будет следующим знаком и т.д. [6].

### 2.2.1. Двоичная система счисления

Люди предпочитают десятичную систему, вероятно, потому, что с древних времен считали по пальцам. Но, не всегда и не везде люди пользовались десятичной системой счисления. В Китае, например, долгое время применялась пятеричная система счисления. В ЭВМ используют двоичную систему потому, что она имеет ряд преимуществ перед другими:

1) для ее реализации используются технические *элементы с двумя возможными состояниями* (есть ток – нет тока, намагничен – ненамагничен);

2) представление информации посредством только двух состояний *надежно и помехоустойчиво*;

3) возможно *применение аппарата булевой алгебры* для выполнения логических преобразований информации;

4) двоичная арифметика проще десятичной (двоичные таблицы сложения и умножения предельно просты).

В двоичной системе счисления всего две цифры, называемые **двоичными** (*binary digits*). Сокращение этого наименования привело к появлению термина **бит**, ставшего названием разряда двоичного числа. Веса разрядов в двоичной системе изменяются по степеням двойки. Поскольку вес каждого разряда умножается либо на 0, либо на 1, то в результате значение числа определяется как сумма соответствующих значений степеней двойки. Если какой-либо разряд двоичного числа равен 1, то он называется *значащим разрядом*. Запись числа в двоичном виде намного длиннее записи в десятичной системе счисления.

Арифметические действия, выполняемые в двоичной системе, подчиняются тем же правилам, что и в десятичной системе. Только в двоичной системе перенос единиц в старший разряд возникает чаще, чем в десятичной. Правила сложения в двоичной системе приведены в табл. 4.

Таблица 4

Правила сложения в двоичной системе счисления

| **Слагаемое** | **Слагаемое** | **Сумма** |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 10 |

Особая значимость двоичной системы счисления в информатике определяется тем, что внутреннее представление любой информации в компьютере является двоичным, т.е. описываемым набором только из двух знаков (0 и 1).

При переводе чисел из десятичной системы счисления в двоичную целая и дробная части переводятся порознь. Для перевода целой части (или просто целого числа) числа необходимо разделить её на основание системы счисления и продолжать делить частные от деления до тех пор, пока частное не станет равным 0. Значения получившихся остатков, взятые в обратной последовательности, образуют искомое двоичное число.

**Пример 6.** Перевод числа из десятичной системы счисления в двоичную. Расчет представлен в табл. 5

Таблица 5

Вспомогательный расчёт при переводе числа

| **Частное** | **Остаток** |
|---|---|
| 25:2=12 | 1 |
| 12:2=6 | 0 |
| 6:2=3 | 0 |
| 3:2=1 | 1 |
| 1:2=0 | 1 |

Таким образом, $25_{(10)} = 11001_2$.

Для перевода дробной части надо умножить её на 2. Целая часть произведения будет первой цифрой числа в двоичной системе. Затем, отбрасывая от результата целую часть, вновь умножаем на 2 и т.д. Конечная десятичная дробь при этом может стать бесконечной (периодической) двоичной. Например:

0,73*2=1,46 (целая часть 1),

0,46*2=0,92 (целая часть 0),

0,92*2=1,84 (целая часть 1),

0,84*2=1,68 (целая часть 1) и т.д.

В итоге $0,73_{(10)} = 0,1011\ldots_{(2)}$.

Над числами, записанными в любой системе счисления, можно производить различные арифметические операции. Так, для сложения и умножения двоичных чисел необходимо использовать табл. 6.

При двоичном сложении 1+1 возникает перенос единицы в старший разряд – точь-в-точь в десятичной арифметике:

```
 +1001        ×1001
   11           11
 ------       ------
 1100          1001
             +1001  0
             -------
              11011
```

Таблица 6

Арифметические операции над двоичными числами

| **Сложение** | | | | **Умножение** | | |
|---|---|---|---|---|---|---|
| + | 0 | 1 | | * | 0 | 1 |
| 0 | 0 | 1 | | 0 | 0 | 0 |
| 1 | 1 | 10 | | 1 | 0 | 1 |

Необходимо усвоить арифметические операции в двоичной системе, общие правила перевода целых и вещественных чисел из десятич-

ной формы в другие системы счисления. Проследить методику перевода чисел из двоичной, восьмеричной и шестнадцатеричной СС в десятеричную и обратно.

Следует отметить, что большинство калькуляторов, реализованных на ЭВМ (в том числе и KCalc) позволяют осуществлять работу в системах счисления с основаниями 2, 8, 16 и, конечно, 10.

### 2.2.2. Восьмеричная и шестнадцатеричная системы счисления

При наладке аппаратных средств ЭВМ или создании новой программы возникает необходимость «заглянуть внутрь» памяти машины, чтобы оценить ее текущее состояние. Но там все заполнено длинными последовательностями нулей и единиц двоичных чисел. Эти последовательности очень неудобны для восприятия человеком, привыкшим к более короткой записи десятичных чисел. Кроме того, естественные возможности человеческого мышления не позволяют оценить быстро и точно величину числа, представленного, например, комбинацией из 16 нулей и единиц.

Для облегчения восприятия двоичного числа решили разбивать его на группы разрядов, например, по три или четыре разряда. Эта идея оказалась очень удачной, так как последовательность из трех бит имеет 8 комбинаций, а последовательность из 4 бит – 16. Числа 8 и 16 являются степенями двойки, поэтому легко находить соответствие с двоичными числами. Развивая эту идею, пришли к выводу, что группы разрядов можно закодировать, сократив при этом длину последовательности знаков. Для кодировки трех битов требуется восемь цифр, поэтому взяли цифры от 0 до 7 десятичной системы. Для кодировки же четырех битов необходимо шестнадцать знаков; для этого взяли 10 цифр десятичной системы и 6 букв латинского алфавита: A, B, C, D, E, F. Полученные системы, имеющие основания 8 и 16, назвали соответственно восьмеричной и шестнадцатеричной.

В **восьмеричной** (octal) системе счисления используются восемь различных цифр 0, 1, 2, 3, 4, 5, 6, 7. Основание системы – 8. При записи отрицательных чисел перед последовательностью цифр ставят знак минус. Сложение, вычитание, умножение и деление чисел, представленных в восьмеричной системе, выполняются весьма просто подобно тому, как это делают в общеизвестной десятичной системе счисления. В различных языках программирования запись восьмеричных чисел начинается с 0, например, запись 011 означает число 9.

В **шестнадцатеричной** (hexadecimal) системе счисления применяется десять различных цифр и шесть первых букв латинского алфавита.

При записи отрицательных чисел слева от последовательности цифр ставят знак минус. Для того чтобы при написании компьютерных программ отличить числа, записанные в шестнадцатеричной системе, от других, перед числом ставят 0x. То есть 0x11 и 11 – это разные числа. В других случаях можно указать основание системы счисления нижним индексом [6].

Шестнадцатеричная система счисления широко используется при задании различных оттенков цвета при кодировании графической информации (модель RGB). Так, в редакторе гипертекста Netscape Composer можно задавать цвета для фона или текста как в десятичной, так и шестнадцатеричной системах счисления.

### 2.2.3. Перевод чисел из одной системы счисления в другую

Наиболее часто встречающиеся системы счисления – это двоичная, шестнадцатеричная и десятичная. Как же связаны между собой представления числа в различных системах счисления? Рассмотрим различные способы перевода чисел из одной системы счисления в другую на конкретных примерах.

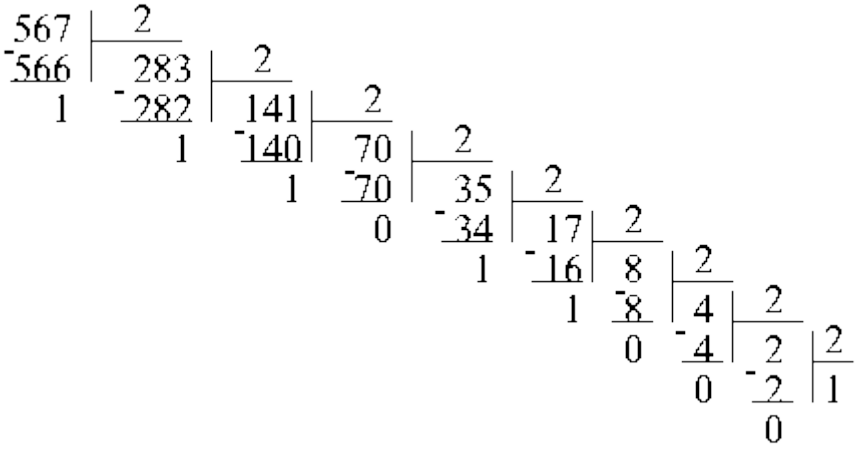
Пусть требуется перевести число 567 из десятичной в двоичную систему. Сначала определим максимальную степень двойки, такую, чтобы два в этой степени было меньше или равно исходному числу. В нашем случае это 9, т.к. $2^9 = 512$, а $2^{10} = 1024$, что больше начального числа. Таким образом, мы получим число разрядов результата. Оно равно 9 + 1 = 10. Поэтому результат будет иметь вид 1xxxxxxxxx, где вместо x могут стоять любые двоичные цифры. Найдем вторую цифру результата. Возведем двойку в степень 9 и вычтем из исходного числа: $567 – 2^9 = 55$. Остаток сравним с числом $2^8=256$. Так как 55 меньше 256, то девятый разряд будет нулем, т.е. результат примет вид 10xxxxxxxx. Рассмотрим восьмой разряд. Так как $2^7=128>55$, то и он будет нулевым.

Седьмой разряд также оказывается нулевым. Искомая двоичная запись числа принимает вид 1000xxxxxx. $2^5 = 32 < 55$, поэтому шестой разряд равен 1 (результат 10001xxxxx). Для остатка 55 – 32 = 23 справедливо неравенство $2^4 = 16 < 23$, что означает равенство единице пятого разряда. Действуя аналогично, получаем в результате число 1000110111. Мы разложили данное число по степеням двойки:

$$567 = 1*2^9+0*2^8+0*2^7+0*2^6+1*2^5+1*2^4+0*2^3+1*2^2 +1*2^1+1*2^0 .$$

При другом способе перевода чисел используется операция деления в столбик. Рассмотрим то же самое число 567. Разделив его на 2, получим частное 283 и остаток 1. Проведем ту же самую операцию с числом 283. Получим частное 141, остаток 1. Опять делим полученное частное

на 2, и так до тех пор, пока частное не станет меньше делителя. Теперь для того, чтобы получить число в двоичной системе счисления, достаточно записать последнее частное, т.е. 1, и приписать к нему в обратном порядке все полученные в процессе деления остатки.



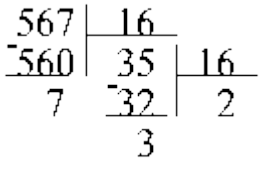

Результат, естественно, не изменился: 567 в двоичной системе счисления записывается как 1000110111.

Эти два способа применимы при переводе числа из десятичной системы в систему с любым основанием. Для закрепления навыков рассмотрим перевод числа 567 в систему счисления с основанием 16.

Сначала осуществим разложение данного числа по степеням основания. Искомое число будет состоять из трех цифр, т.к. $16^2 = 256 < 567 < 16^3 = 4096$. Определим цифру старшего разряда. $2*16^2 = 512 < 567 < 3*16^2 = 768$, следовательно искомое число имеет вид 2xx, где вместо x могут стоять любые шестнадцатеричные цифры. Остается распределить по следующим разрядам число 55 (567 – 512). $3*16 = 48 < 55 < 4 * 16 = 64$, значит во втором разряде находится цифра 3. Последняя цифра равна 7 (55 – 48). Искомое шестнадцатеричное число равно 237.

Второй способ состоит в осуществлении последовательного деления в столбик, с единственным отличием в том, что делить надо не на 2, а на 16, и процесс деления заканчивается, когда частное становится строго меньше 16.

Конечно, не надо забывать и о том, что для записи числа в шестнадцатеричной системе счисления, необходимо заменить 10 на A, 11 на B и так далее.




Операция перевода в десятичную систему выглядит гораздо проще, так как любое десятичное число можно представить в виде

$x = a_0*p^n + a_1*p^{n-1} + ... + a_{n-1}*p^1 + a_n*p^0$,

где $a_0$ ... $a_n$ – это цифры данного числа в системе счисления с основанием p.

**Пример 7.** Переведем число 4A3F в десятичную систему. По определению, $4A3F = 4*16^3 + A*16^2 + 3*16 + F$. Заменив A на 10, а F на 15, получим $4*16^3+10*16^2+3*16+15= 19007$.

Пожалуй, проще всего осуществляется перевод чисел из двоичной системы в системы с основанием, равным степеням двойки (8 и 16), и наоборот. Для того чтобы целое двоичное число записать в системе счисления с основанием $2^n$, нужно

1) данное двоичное число разбить справа налево на группы по n-цифр в каждой;

2) если в последней левой группе окажется меньше n разрядов, то дополнить ее нулями до нужного числа разрядов;

3) рассмотреть каждую группу, как n-разрядное двоичное число, и заменить ее соответствующей цифрой в системе счисления с основанием $2^n$. В табл. 7 и табл. 8 приведены двоично-шестнадцатеричный перевод числа и двоично-восьмеричный перевод числа соответственно.

Таблица 7

Двоично-шестнадцатеричная таблица

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **2-ная** | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |
| **16-ная** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| **2-ная** | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
| **16-ная** | 8 | 9 | A | B | C | D | E | F |

Таблица 8

Двоично-восьмеричная таблица

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **2-ная** | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| **8-ная** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

## 2.3. Двоичное кодирование информации

Среди всего разнообразия информации, обрабатываемой на компьютере, значительную часть составляют числовая, текстовая, графическая и аудиоинформация. Познакомимся с некоторыми способами кодирования этих типов информации в ЭВМ.

## Кодирование чисел

Существуют два основных формата представления чисел в памяти компьютера. Один из них используется для кодирования целых чисел, второй (так называемое представление числа в формате с плавающей точкой) используется для задания некоторого подмножества действительных чисел.

Множество *целых чисел*, представимых в памяти ЭВМ, ограничено. Диапазон значений зависит от размера области памяти, используемой для размещения чисел. *В k-разрядной ячейке может храниться $2^k$ различных значений целых чисел*.

Чтобы получить внутреннее представление целого положительного числа N, хранящегося в *k*-разрядном машинном слове, необходимо:

1) перевести число *N* в двоичную систему счисления;

2) полученный результат дополнить слева незначащими нулями до *k* разрядов.

**Пример 8.** Получить внутреннее представление целого числа 1607 в двухбайтовой ячейке.

Переведем число в двоичную систему: $1607_{10} = 11001000111_2$. Внутреннее представление этого числа в ячейке будет следующим: 0000 0110 0100 0111.

Для записи внутреннего представления целого отрицательного числа (-*N*) необходимо:

1) получить внутреннее представление положительного числа N;

2) обратный код этого числа заменой 0 на 1 и 1 на 0;

3) полученному числу прибавить 1.

**Пример 9.** Получим внутреннее представление целого отрицательного числа -1607. Воспользуемся результатом предыдущего примера и запишем внутреннее представление положительного числа 1607: 0000 0110 0100 0111. Инвертированием получим обратный код: 1111 1001 1011 1000. Добавим единицу: 1111 1001 1011 1001 – это и есть внутреннее двоичное представление числа –1607.

*Формат с плавающей точкой* использует представление вещественного числа *R* в виде произведения мантиссы *m* на основании системы счисления *n* в некоторой целой степени *p*, которую называют **порядком**: $R = m * n^p$.

Представление числа в форме с плавающей точкой неоднозначно. Например, справедливы следующие равенства: $12.345 = 0.0012345 \times 10^4 = 1234.5 \times 10^{-2} = 0.12345 \times 10^2$.

Чаще всего в ЭВМ используют *нормализованное представление числа в форме с плавающей точкой*. Мантисса в таком представлении

должна удовлетворять условию: $0.1_p <= m < 1_p$. Иначе говоря, мантисса меньше 1 и первая значащая цифра – не ноль ($p$ – основание системы счисления). [12]

В памяти компьютера мантисса представляется как целое число, содержащее только значащие цифры (0 целых и запятая не хранятся), так для числа 12.345 в ячейке памяти, отведенной для хранения мантиссы, будет сохранено число 12345. Для однозначного восстановления исходного числа остается сохранить только его порядок, в данном примере – это 2.

## Кодирование текста

Множество символов, используемых при записи текста, называется алфавитом. Количество символов в алфавите называется его мощностью.

Для представления *текстовой информации* в компьютере чаще всего используется алфавит мощностью 256 символов. Один символ из такого алфавита несет 8 бит информации, т.к. $2^8 = 256$. Но 8 бит составляют один байт, следовательно, двоичный код каждого символа занимает 1 байт памяти ЭВМ.

Все символы такого алфавита пронумерованы от 0 до 255, а каждому номеру соответствует 8-разрядный двоичный код от 00000000 до 11111111. Этот код является порядковым номером символа в двоичной системе счисления.

Для разных типов ЭВМ и операционных систем используются различные таблицы кодировки, отличающиеся порядком размещения символов алфавита в кодовой таблице. Международным стандартом на персональных компьютерах является уже упоминавшаяся таблица кодировки ASCII.

*Принцип последовательного кодирования алфавита* заключается в том, что в кодовой таблице ASCII латинские буквы (прописные и строчные) располагаются в алфавитном порядке. Расположение цифр также упорядочено по возрастанию значений. [7]

Стандартными в этой таблице являются только первые 128 символов, т.е. символы с номерами от нуля (двоичный код 00000000) до 127 (01111111). Сюда входят буквы латинского алфавита, цифры, знаки препинания, скобки и некоторые другие символы. Остальные 128 кодов, начиная со 128 (двоичный код 10000000) и кончая 255 (11111111), используются для кодировки букв национальных алфавитов, символов псевдографики и научных символов. О кодировании символов русского алфавита рассказывается в главе «Обработка документов».

## Кодирование графической информации

В видеопамяти находится двоичная информация об изображении, выводимом на экран. Почти все создаваемые, обрабатываемые или просматриваемые с помощью компьютера изображения можно разделить на две большие части – растровую и векторную графику.

*Растровые изображения* представляют собой однослойную сетку точек, называемых пикселами (pixel, от англ. picture element). *Код пиксела* содержит информацию о его цвете.

Для черно-белого изображения (без полутонов) пиксел может принимать только два значения: белый и черный (светится – не светится), а для его кодирования достаточно одного бита памяти: 1 – белый, 0 – черный.

Пиксел на цветном дисплее может иметь различную окраску, поэтому одного бита на пиксел недостаточно. Для кодирования 4-цветного изображения требуются два бита на пиксел, поскольку два бита могут принимать 4 различных состояния. Может использоваться, например, такой вариант кодировки цветов: 00 – черный, 10 – зеленый, 01 – красный, 11 – коричневый.

На RGB-мониторах все разнообразие цветов получается сочетанием базовых цветов – красного (Red), зеленого (Green), синего (Blue), из которых можно получить 8 основных комбинаций (табл. 9).

Таблица 9

Сочетание цветов в RGB-мониторах

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **R** | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| **G** | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| **B** | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| **цвет** | чёр-ный | синий | зелё-ный | гол-у-бой | крас-ный | розо-вый | корич-не-вый | белый |

Разумеется, если иметь возможность управлять интенсивностью (яркостью) свечения базовых цветов, то количество различных вариантов их сочетаний, порождающих разнообразные оттенки, увеличивается. Количество различных цветов – $K$ и количество битов для их кодировки – $N$ связаны между собой простой формулой: $2^N = K$.

В противоположность растровой графике *векторное изображение* многослойно. Каждый элемент векторного изображения – линия, прямоугольник, окружность или фрагмент текста – располагается в своем собственном слое, пикселы которого устанавливаются независимо от дру-

гих слоев. Каждый элемент векторного изображения является объектом, который описывается с помощью специального языка (математических уравнения линий, дуг, окружностей и т. д.). Сложные объекты (ломаные линии, различные геометрические фигуры) представляются в виде совокупности элементарных графических объектов.

Объекты векторного изображения, в отличие от растровой графики, могут изменять свои размеры без потери качества (при увеличении растрового изображения увеличивается зернистость).

### Кодирование звука

Из курса физики вам известно, что *звук* – это колебания воздуха. Если преобразовать звук в электрический сигнал (например, с помощью микрофона), мы увидим плавно изменяющееся с течением времени напряжение. Для компьютерной обработки такой – аналоговый – сигнал нужно каким-то образом преобразовать в последовательность двоичных чисел.

Поступим следующим образом. Будем измерять напряжение через равные промежутки времени и записывать полученные значения в память компьютера. Этот процесс называется *дискретизацией* или оцифровкой (рис. 14), а устройство, выполняющее его – *аналого-цифровым преобразователем* (АЦП).

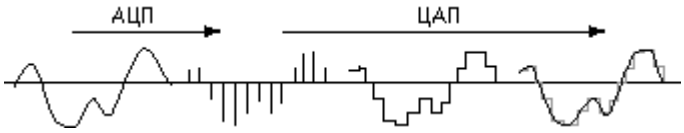



Рис. 14. Оцифровка звука

Для того чтобы воспроизвести закодированный таким образом звук, нужно выполнить обратное преобразование (для него служит *цифро-аналоговый преобразователь* – ЦАП), а затем сгладить получившийся ступенчатый сигнал.

Чем выше частота дискретизации (т. е. количество отсчетов за секунду) и чем больше разрядов отводится для каждого отсчета, тем точнее будет представлен звук. Но при этом увеличивается и размер звукового файла. Поэтому в зависимости от характера звука, требований, предъявляемых к его качеству и объему занимаемой памяти, выбирают некоторые компромиссные значения.

Описанный способ кодирования звуковой информации достаточно универсален, он позволяет представить любой звук и преобразовывать

его самыми разными способами. Но бывают случаи, когда выгодней действовать по-иному.

Человек издавна использует довольно компактный способ представления музыки – нотную запись. В ней специальными символами указывается, какой высоты звук, на каком инструменте и как сыграть. Фактически, ее можно считать алгоритмом для музыканта, записанным на особом формальном языке. В 1983 г. ведущие производители компьютеров и музыкальных синтезаторов разработали стандарт, определивший такую систему кодов. Он получил название MIDI.

Конечно, такая система кодирования позволяет записать далеко не всякий звук, она годится только для инструментальной музыки. Но есть у нее и неоспоримые преимущества: чрезвычайно компактная запись, естественность для музыканта (практически любой MIDI-редактор позволяет работать с музыкой в виде обычных нот), легкость замены инструментов, изменения темпа и тональности мелодии.

Заметим, что существуют и другие, чисто компьютерные, форматы записи музыки. Среди них следует отметить формат MP3, позволяющий с очень большим качеством и степенью сжатия кодировать музыку. При этом вместо 18–20 музыкальных композиций на стандартный компакт-диск (CDROM) помещается около 200. Одна песня занимает примерно 3,5 Mb, что позволяет пользователям сети Интернет легко обмениваться музыкальными композициями [12].

## 2.4 Вопросы для самоконтроля

1. Наименьшая единица измерения информации в компьютере.
2. Системы счисления, применяемые в ЭВМ.
3. Максимальная цифра, применяющаяся в восьмеричной системе счисления.
4. Сумма в двоичной системе счисления при сложении двух чисел, заданных также в двоичной системе счисления 01+11.
5. Представление числа $21_{10}$ в двоичной системе счисления.
6. Внутреннее двоичное представление числа 10 в двухбайтовой ячейке
7. Что основано на том, что в кодовой таблице ASCII латинские буквы (прописные и строчные) располагаются в алфавитном порядке.
8. Сколько цифр может храниться в k-разрядной ячейке может?
9. Объём в памяти ЭВМ, занимаемый одним двоичным кодом каждого символа.
10. Знаки, которые используются в восьмеричной и шестнадцатеричной системах счисления одновременно.

11. Коды, применяемые в ЭВМ для выполнения арифметических операций в двоичной системе счисления.
12. Основные форматы представления чисел в памяти компьютера.
13. Устройство, служащее для преобразования звукового сигнала в машинный код.
14. Стандартные символы в кодовой таблице ASCII.
15. Код красного цвета на RGB-мониторах.

# 3. ОСНОВЫ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ ЗАДАЧ

## 3.1. Технология программирования и основные этапы её развития

*Технология программирования* – совокупность методов и средств, используемых в процессе разработки программного обеспечения.

Как любая другая технология, технология программирования представляет собой набор технологических инструкций, включающих:

1) указание последовательности выполнения технологических операций;

2) перечисление условий, при которых выполняется та или иная операция;

3) описание самих операций, где для каждой операции определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки и т.п.

Кроме набора операций и их последовательности, технология также определяет способ описания проектируемой системы, точнее модели, используемой на конкретном этапе разработки.

Технологии программирования играло разную роль на разных этапах развития программирования. По мере повышения мощности компьютеров и развития средств и методологии программирования росла и сложность решаемых на компьютерах задач, что привело к повышенному вниманию к технологии программирования. Резкое удешевление стоимости компьютеров и, в особенности, стоимости хранения информации на компьютерных носителях привело к широкому внедрению компьютеров практически во все сферы человеческой деятельности, что существенно изменило направленность технологии программирования. Человеческий фактор стал играть в ней решающую роль. Сформировалось достаточно глубокое понятие качества программных средств (ПС), в котором акценты стали ставится не столько на его эффективности, сколько на удобстве работы с ним для пользователей (не говоря уже о его надежности). Широкое использование компьютерных сетей привело к интенсивному развитию распределенных вычислений, дистанционного доступа к информации и электронного способа обмена сообщениями между людьми. Компьютерная техника из средства решения отдельных задач все более превращается в средство информационного моделирования реального и мыслимого мира, способное просто отвечать людям на интересующие их вопросы. Начинается этап глубокой и полной информатизации (компьютеризации) человеческого общества. Все это ставит перед технологией программирования новые и достаточно трудные проблемы [1].

Сделаем краткую характеристику развития программирования по десятилетиям.

В 50-е годы мощность компьютеров (компьютеры первого поколения) была невелика, а программирование для них велось, в основном, в машинном коде. Решались, главным образом, научно-технические задачи (счёт по формулам), задание на программирование уже содержало, как

правило, достаточно точную постановку задачи. Использовалась интуитивная технология программирования: почти сразу приступали к составлению программы по заданию, при этом часто задание несколько раз изменялось (что сильно увеличивало время и без того итерационного процесса составления программы), минимальная документация оформлялась уже после того, как программа начинала работать. Тем не менее, именно в этот период родилась фундаментальная для технологии программирования концепция модульного программирования (для преодоления трудностей программирования в машинном коде) []. Появились первые языки программирования высокого уровня, из которых только ФОРТРАН пробился для использования в следующие десятилетия.

В 60-е годы можно было наблюдать бурное развитие и широкое использование языков программирования высокого уровня (АЛГОЛ 60, ФОРТРАН, КОБОЛ и др.), роль которых в технологии программирования явно преувеличивалась. Надежда на то, что эти языки решат все проблемы при разработки больших программ, не оправдалась. В результате повышения мощности компьютеров и накопления опыта программирования на языках высокого уровня быстро росла сложность решаемых на компьютерах задач, в результате чего обнаружилась ограниченность языков, проигнорировавших модульную организацию программ. И только ФОРТРАН, бережно сохранивший возможность модульного программирования, гордо прошествовал в следующие десятилетия (все его ругали, но его пользователи отказаться от его услуг не могли из-за грандиозного накопления фонда программных модулей, которые с успехом использовались в новых программах). Кроме того, было понято, что важно не только то, на каком языке мы программируем, но и то, как мы программируем. Это было уже началом серьезных размышлений над методологией и технологией программирования. Появление в компьютерах 2-го поколения прерываний привело к развитию мультипрограммирования и созданию больших программных систем. Это стало возможным с использованием коллективной разработки, которая поставила ряд серьезных технологических проблем [9].

В 70-е годы получили широкое распространение информационные системы и базы данных. Этому способствовало очень важное событие, происшедшее в середине 70-х годов: стоимость хранения одного бита информации на компьютерных носителях стала меньше, чем на традиционных. Интенсивно развивалась технология программирования []: обоснование и широкое внедрение нисходящей разработки и структурного программирования, развитие абстрактных типов данных и модульного программирования (в частности, возникновение идеи разделения спецификации и реализации модулей и использование модулей, скрывающих струк-

туры данных), исследование проблем обеспечения надежности и мобильности ПС, создание методики управления коллективной разработкой ПС, появление инструментальных программных средств (программных инструментов) поддержки технологии программирования.

80-е годы характеризуются широким внедрением персональных компьютеров во все сферы человеческой деятельности и тем самым созданием обширного и разнообразного контингента пользователей ПС. Это привело к бурному развитию пользовательских интерфейсов и созданию четкой концепции качества ПС. Появляются языки программирования (например, Ада), учитывающие требования технологии программирования. Развиваются методы и языки спецификации ПС. Выходит на передовые позиции объектный подход к разработке ПС. Создаются различные инструментальные среды разработки и сопровождения ПС. Развивается концепция компьютерных сетей.

90-е годы знаменательны широким охватом всего человеческого общества международной компьютерной сетью, персональные компьютеры стали подключаться к ней как терминалы. Это поставило ряд проблем регулирования доступа к компьютерно-сетевой информации (как технологического, так и юридического и этического характера). Остро встала проблема защиты компьютерной информации и передаваемых по сети сообщений. Стали бурно развиваться компьютерная технология (CASE-технология) разработки ПС и связанные с ней формальные методы спецификации программ. Начался решающий этап полной информатизации и компьютеризации общества.

Таким образом, можно выделить четыре этапа развития программирования, приведённые ниже.

### Основные этапы развития программирования как науки

*Первый этап* – «стихийное программирование». Этот этап охватывает период с момента появления первых вычислительных машин до середины 60-х годов XX века. В этот период практически отсутствовали сформулированные технологии, и программирование фактически было искусством. Первые программы имели простейшую структуру. Они состояли из собственно программы на машинном языке и обрабатываемых ею данных. Сложность программ в машинных кодах ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и местонахождение данных при программировании.

*Второй этап* – структурный подход к программированию (60–70-е годы XX века.) Структурный подход к программированию представляет

собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит декомпозиция сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. С появлением других принципов декомпозиции данный способ получил название процедурной декомпозиции.

*Третий этап* – объектный подход к программированию (с середины 80-х – до конца 90-х годов XX века). Объектно-ориентированное программирование определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа, а классы образуют иерархию с наследованием свойств. Взаимодействие программных объектов в такой системе осуществляется путем передачи сообщений.

*Четвертый этап* – компетентный подход и CASE-технологии (с середины 90-х годов XX века – до нашего времени). Компонентный подход предполагает построение программного обеспечения из отдельных компонентов – физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы. В отличие от обычных объектов объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде и использовать в любом языке программирования, поддерживающем соответствующую технологию.

## 3.2. Источники ошибок в программных средствах

Интеллектуальные возможности человека, используемые при разработке программных систем. Понятия о простых и сложных системах, о малых и больших системах. Неправильный перевод информации из одного представления в другое – основная причина ошибок при разработке программных средств. Модель перевода и источники ошибок.

### 3.2.1. Интеллектуальные возможности человека

Дейкстра [Ошибка: источник перёкрестной ссылки не найден2] выделяет три интеллектуальные возможности человека, используемые при разработке ПС:

1) способность к перебору,
2) способность к абстракции,
3) способность к математической индукции.

Способность человека к перебору связана с возможностью последовательного переключения внимания с одного предмета на другой с узнаванием искомого предмета. Эта способность весьма ограничена – в среднем человек может уверенно (не сбиваясь) перебирать в пределах 1000 предметов (элементов). Человек должен научиться действовать с учетом этой своей ограниченности. Средством преодоления этой ограниченности является его способность к абстракции, благодаря которой человек может объединять разные предметы или экземпляры в одно понятие, заменять множество элементов одним элементом (другого рода). Способность человека к математической индукции позволяет ему справляться с бесконечными последовательностями.

При разработке ПС человек имеет дело с системами. Под системой будем понимать совокупность взаимодействующих (находящихся в отношениях) друг с другом элементов. ПС можно рассматривать как пример системы. Логически связанный набор программ является другим примером системы. Любая отдельная программа также является системой. *Понять систему* – значит осмысленно перебрать все пути взаимодействия между ее элементами. В силу ограниченности человека к перебору будем различать простые и сложные системы [Ошибка: источник перёкрестной ссылки не найден]. Под простой системой будем понимать такую систему, в которой человек может уверенно перебрать все пути взаимодействия между ее элементами, а под сложной системой – такую систему, в которой он этого сделать не в состоянии. Между простыми и сложными системами нет чёткой границы, поэтому можно говорить и о промежуточном классе систем: к таким системам относятся программы, о которых программистский фольклор утверждает, что «в каждой отлаженной программе имеется хотя бы одна ошибка».

При разработке ПС мы не всегда можем уверенно знать обо всех связях между её элементами из-за возможных ошибок. Поэтому полезно уметь оценивать сложность системы по числу ее элементов: числом потенциальных путей взаимодействия между её элементами, т. е. $n!$ , где $n$ – число её элементов. Систему назовём *малой*, если $n < 7$ ($6! = 720 < 1000$), систему назовём *большой*, если $n > 7$ . При $n = 7$ имеем *промежуточный класс* систем. Малая система всегда проста, а большая может быть как простой, так и сложной. Задача технологии программирования – научиться делать большие системы простыми.

Полученная оценка простых систем по числу элементов широко используется на практике. Так, для руководителя коллектива весьма желательно, чтобы в нем не было больше шести взаимодействующих между собой подчиненных. Весьма важно также следовать правилу: «всё, что может быть сказано, должно быть сказано в шести пунктах или мень-

ше». Этому правилу мы будем стараться следовать в настоящем пособии: всякие перечисления взаимосвязанных утверждений (набор рекомендаций, список требований и т.п.) будут соответствующим образом группироваться и обобщаться. Полезно ему следовать и при разработке ПС.

### 3.2.2. Неправильный перевод как причина ошибок в программных средствах

При разработке и использовании ПС мы многократно имеем дело с преобразованием (переводом) информации из одной формы в другую (Ошибка: источник перёкрестной ссылки не найден5).

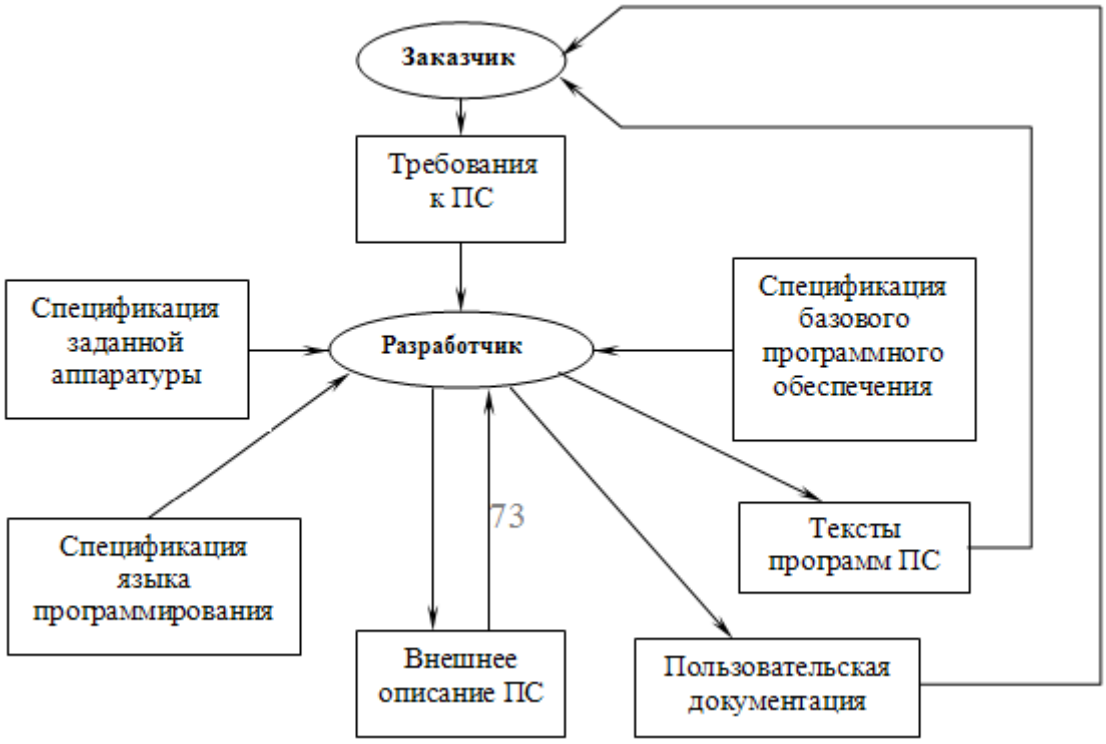



Рис. 15. Грубая схема разработки и применения ПС

Заказчик формулирует свои потребности в ПС в виде некоторых требований. Исходя из этих требований, разработчик создаёт внешнее описание ПС, используя при этом спецификацию (описание) заданной аппаратуры и, возможно, спецификацию базового программного обеспечения. На основании внешнего описания и спецификации языка программирования создаются тексты программ ПС на этом языке. По внешнему описанию ПС разрабатывается также и пользовательская документация. Текст каждой программы является исходной информацией при

любом её преобразовании, в частности, при исправлении в ней ошибки [8].

## Модель перевода

Чтобы понять природу ошибок при переводе рассмотрим модель [Ошибка: источник перёкрестной ссылки не найден], изображённую на рис. 16. На ней человек осуществляет перевод информации из представления A в представление B. При этом он совершает четыре основных шага перевода:

- он получает информацию, содержащуюся в представлении A, с помощью своего читающего механизма R;
- он запоминает полученную информацию в своей памяти M;
- он выбирает из своей памяти преобразуемую информацию и информацию, описывающую процесс преобразования, выполняет перевод и посылает результат своему пишущему механизму W;
- с помощью этого механизма он фиксирует представление B.

На каждом из этих шагов человек может совершить ошибку разной природы. На первом шаге способность человека «читать между строк» (способность, позволяющая ему понимать текст, содержащий неточности или даже ошибки) может стать причиной ошибки в ПС. Ошибка возникает в том случае, когда при чтении документа A человек, пытаясь восстановить недостающую информацию, видит то, что он ожидает, а не то, что имел в виду автор документа A. В этом случае лучше было бы обратиться к автору документа за разъяснениями. При запоминании информации человек осуществляет её осмысливание (здесь важен его уровень подготовки и знание предметной области, к которой относится документ A). И, если он поверхностно или неправильно поймёт, то информация будет запомнена в искажённом виде. На третьем этапе забывчивость человека может привести к тому, что он может выбрать из своей памяти не всю преобразуемую информацию или не все правила перевода, в результате чего перевод будет осуществлён неверно.

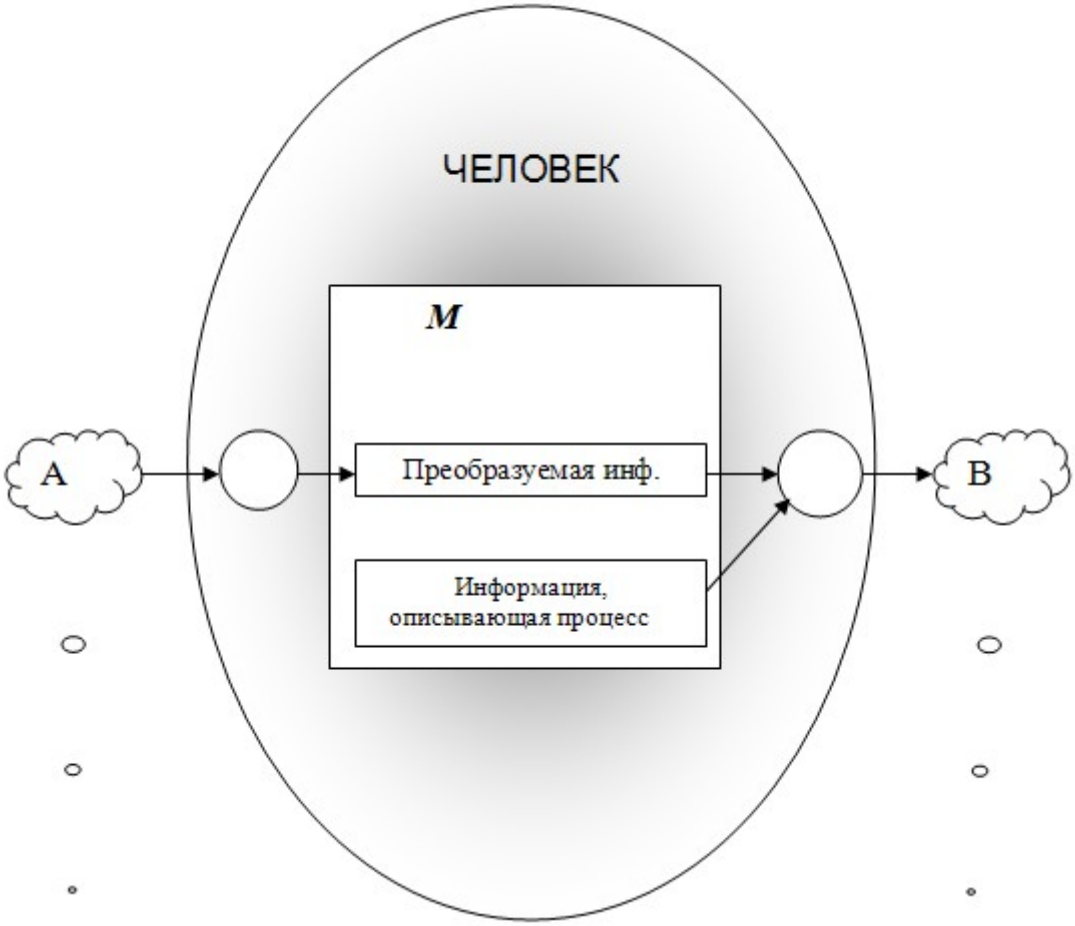



Рис. 16. Модель перевода

Это обычно происходит при большом объёме плохо организованной информации. И, наконец, на последнем этапе стремление человека поскорее зафиксировать информацию часто приводит к тому, что представление этой информации оказывается неточным, создавая ситуацию для последующей неоднозначной её интерпретации.

### 3.2.3. Основные пути борьбы с ошибками

Учитывая рассмотренные особенности действий человека при переводе можно указать следующие пути борьбы с ошибками:

1) сужение пространства перебора (упрощение создаваемых систем);

2) обеспечение требуемого уровня подготовки разработчика (это функции менеджеров коллектива разработчиков);

3) обеспечение однозначности интерпретации представления информации;

4) контроль правильности перевода (включая и контроль однозначности интерпретации) [8].

### 3.3. Понятие алгоритма, свойства алгоритмов

Алгоритм относится к *фундаментальным понятиям* информатики. На понятии алгоритма построено все основные принципы программирования – составления программ для вычислительных машин.

*Алгоритм* – это совокупность действий со строго определенными правилами выполнения. В информатике изучаются различного рода алгоритмы – диалоговые алгоритмы, алгоритмы обработки данных, вычислительные алгоритмы, алгоритмы управления роботами, станками и другими техническими устройствами [1].

Для *описания алгоритмов* используются блок-схемы или структурированная запись. Блок-схемы наглядны. Однако блок-схемы трудно рисовать, в них сложно вносить изменения и исправления из-за сложности перерисовки рамок и стрелок. Однако блок-схемы до сих пор требуются отечественными стандартами на документирование программ.

Достоинство записи алгоритмов и программ в структурированной форме заключается в *простоте их чтения и ввода* с экрана ЭВМ, а также в простоте внесения изменений и исправлений с использованием даже самых простейших редакторов тестов. По этим причинам зарубежом блок-схемы уже давно не используются ни для документирования, ни для обучения, а все современные языки построены на принципах структурного программирования.

Алгоритм, приведенный слева, записан на псевдокоде. *Псевдокод* – это язык записи структурированных алгоритмов в качестве документации к программам для ЭВМ. Особенность псевдокода заключается в том, что описания на нем выполняются на родном языке – русском, английском, украинском, казахском, немецком и т.п.

*Достоинства псевдокода* заключаются в том, что описания алгоритмов, записанные на родном языке, намного проще читать и понимать, чем запись программ на языке с иностранной лексикой. По этим причинам псевдокод используется как основное средство документирования программ во всех ведущих фирмах, занимающихся разработкой программ.

С точки зрения информатики алгоритмы, записанные в такой обобщенной записи, позволяют выразить общую *логику работы программ*, независимо от используемых языков программирования и типов ЭВМ. При этом алгоритмы, записанные в такой обобщенной форме, могут быть реализованы с помощью различных языков программирования для самых различных типов ЭВМ.

*Основные свойства* алгоритмов и программ для вычислительных машин следующие [12]:

1) однозначность;
2) результативность;
3) правильность;
4) массовость;
5) определённость;
6) дискретность.

Этими свойствами алгоритмы отличаются от различного рода расплывчатых и неоднозначных предписаний, инструкций и кулинарных рецептов, которые могут толковаться и исполняться многими способами.

*Однозначность* алгоритмов – это однозначность правил их выполнения. Следствием этого свойства алгоритмов является однозначность результатов их выполнения в одинаковых начальных условиях. Это не всегда верно для кулинарных рецептов, когда разные исполнители в одних и тех же условиях могут придавать различный вкус и пикантность одним и тем же блюдам.

*Результативность* – это завершение выполнения алгоритмов определенными результатами. Результативность – наиболее важное свойство алгоритмов и программ, предназначенных для решения прикладных задач. Алгоритмы и программы, не дающие результатов или ведущие к сбоям и отказам, никому не нужны.

*Массовость* – это возможность применения алгоритмов в различных конкретных исходных условиях. Массовые алгоритмы особенно важны для решения прикладных задач, когда алгоритмы и программы должны обеспечить решение целого класса задач, различающихся исходными данными.

*Правильность алгоритмов* определяется правильностью результатов, получаемых с их помощью. По этой причине правильность алгоритмов и программ является относительным понятием. Оценка правильности может проводиться только при наличии требований к конечным результатам.

*Алгоритм* считается *правильным***,** если он дает правильные результаты для любых допустимых начальных условиях. Правильность алгоритмов гарантирует правильность результатов их выполнения.

*Алгоритм* содержит *ошибки***,** если его выполнение может привести к отказам, сбоям или неправильным результатам, либо вовсе не дает никаких результатов. Эти ошибки называются алгоритмическими. Алгоритмы и программы, содержащие такие ошибки, могут нанести вред или ущерб тем, кто захочет ими воспользоваться.

Для оценки правильности алгоритмов и программ необходимо уметь оценивать результаты выполнения составляющих их действий и конечные результаты их выполнения в целом.

*Определенность* (детерминированность) – каждое действие алгоритма должно быть понятно его исполнителю (инструкция к бытовому прибору на японском языке для человека, не владеющего японским языком не является алгоритмом, т.к. не обладает свойством детерминированности).

*Дискретность* – процесс должен быть описан с помощью неделимых операций, выполняемых на каждом шаге (т.е. шаги нельзя разделить на более мелкие шаги).

*Простейшие виды* машинных операций – операции присваивания. С помощью присваиваний в алгоритмах описываются вычисления в программах для ЭВМ.

### Компиляторы и интерпретаторы

С помощью языка программирования создается текст, описывающий ранее составленный алгоритм. Чтобы получить работающую программу, надо этот текст перевести в последовательность команд процессора, что выполняется при помощи специальных программ, которые называются трансляторами. Трансляторы бывают двух видов: *компиляторы* и *интерпретаторы*.

*Компилятор* транслирует текст исходного модуля в машинный код, который называется объектным модулем за один непрерывный процесс. При этом сначала он просматривает исходный текст программы в поисках синтаксических ошибок.

*Интерпретатор* выполняет исходный модуль программы в режиме оператор за оператором, по ходу работы, переводя каждый оператор на машинный язык.

## 3.4. Языки программирования

Разные типы процессоров имеют разный набор команд. Если язык программирования ориентирован на конкретный тип процессора и учитывает его особенности, то он называется языком программирования низкого уровня.

*Языком самого низкого уровня* является язык *ассемблера*, который просто представляет каждую команду машинного кода в виде специальных символьных обозначений, которые называются *мнемониками*. С помощью языков низкого уровня создаются очень эффективные и компактные программы, т.к. разработчик получает доступ ко всем возможностям процессора.

Так как наборы инструкций для разных моделей процессоров тоже разные, то каждой модели процессора соответствует свой язык ассем-

блера, и написанная на нем программа может быть использована только в этой среде. Подобные языки применяют для написания небольших системных приложений, драйверов устройств и т.п.

*Языки программирования высокого уровня* не учитывают особенности конкретных компьютерных архитектур, поэтому создаваемые программы на уровне исходных текстов легко переносятся на другие платформы, если для них созданы соответствующие трансляторы. Разработка программ на языках высокого уровня гораздо проще, чем на машинных языках.

Языками высокого уровня являются:

1. *Фортран* – первый компилируемый язык, созданный в 50-е годы 20 века. В нем были реализован ряд важнейших понятий программирования. Для этого языка было создано огромное количество библиотек, начиная от статистических комплексов и заканчивая управлением спутниками, поэтому он продолжает использоваться во многих организациях.

2. *Кобол* – компилируемый язык для экономических расчетов и решения бизнес-задач, разработанный в начале 60-х годов. В Коболе были реализованы очень мощные средства работы с большими объемами данных, хранящихся на внешних носителях.

3. *Паскаль* – создан в конце 70-х годов швейцарским математиком Никлаусом Виртом специально для обучения программированию. Он позволяет выработать алгоритмическое мышление, строить короткую, хорошо читаемую программу, демонстрировать основные приемы алгоритмизации, он также хорошо подходит для реализации крупных проектов.

4. *Бейсик* – создавался в 60-х годах также для обучения программированию. Для него имеются и компиляторы и интерпретаторы, является одним из самых популярных языков программирования.

5. *Си* – был создан в 70-е годы, первоначально не рассматривался как массовый язык программирования. Он планировался для замены ассемблера, чтобы иметь возможность создавать такие же эффективные и короткие программы, но не зависеть от конкретного процессора. Он во многом похож на Паскаль и имеет дополнительные возможности для работы с памятью. На нем написано много прикладных и системных программ, а также операционная система Unix.

6. *Си++* – объектно-ориентированное расширение языка Си, созданное Бьярном Страуструпом в 1980г.

7. *Java* – язык, который был создан компанией Sun в начале 90-х годов на основе Си++. Он призван упростить разработку приложений на Си++ путем исключения из него низкоуровневых возможностей. Главная особенность языка – это то, что он компилируется не в машинный код, а в платформно-независимый байт-код (каждая команда занимает

один байт). Этот код может выполняться с помощью интерпретатора – виртуальной Java-машины (JVM).

Представленные языки программирования поддерживают такие типы программирования, как:

1) структурное программирование;
2) объектно-ориентированное программирование;
3) обобщённое программирование.

Ниже данные типы программирования рассматриваются более подробно.

### 3.5. Структурное программирование

При программировании модуля следует иметь в виду, что программа должна быть понятной не только компьютеру, но и человеку: и разработчик модуля, и лица, проверяющие модуль, и текстовики, готовящие тесты для отладки модуля, и сопроводители ПС, осуществляющие требуемые изменения модуля, вынуждены будут многократно разбирать логику работы модуля. В современных языках программирования достаточно средств, чтобы запутать эту логику сколь угодно сильно, тем самым, сделать модуль труднопонимаемым для человека и, как следствие этого, сделать его ненадежным или трудно сопровождаемым. Поэтому необходимо принимать меры для выбора подходящих языковых средств и следовать определенной дисциплине программирования. Впервые на это обратил внимание Дейкстра [12] и предложил строить программу как композицию из нескольких типов управляющих конструкций (структур), которые позволяют сильно повысить понимаемость логики работы программы. Программирование с использованием только таких конструкций назвали структурным.

Основными конструкциями структурного программирования являются:

1) следование,
2) разветвление,
3) повторение.

Компонентами этих конструкций являются обобщенные операторы (узлы обработки) и условие (предикат). В качестве обобщенного оператора может быть либо простой оператор используемого языка программирования (операторы присваивания, ввода, вывода, обращения к процедуре), либо фрагмент программы, являющийся композицией основных управляющих конструкций структурного программирования. Существенно, что каждая из этих конструкций имеет по управлению только один вход и один выход. Тем самым, и обобщенный оператор имеет только один вход и один выход.

Весьма важно также, что эти конструкции являются уже математическими объектами (что, посуществу, и объясняет причину успеха структурного программирования). Доказано, что для каждой неструктурированной программы можно построить функционально эквивалентную (т.е. решающую ту же задачу) структурированную программу. Для структурированных программ можно математически доказывать некоторые свойства, что позволяет обнаруживать в программе некоторые ошибки. Этому вопросу будет посвящена отдельная лекция.

Структурное программирование иногда называют еще «программированием без GO TO». Однако дело здесь не в операторе GO TO, а в его беспорядочном использовании. Очень часто при воплощении структурного программирования на некоторых языках программирования (например, на ФОРТРАНе) оператор перехода (GO TO) используется для реализации структурных конструкций, не снижая основных достоинств структурного программирования. Запутывают программу как раз «неструктурные» операторы перехода, особенно переход на оператор, расположенный в тексте модуля выше (раньше) выполняемого оператора перехода. Тем не менее, попытка избежать оператора перехода в некоторых простых случаях может привести к слишком громоздким структурированным программам, что не улучшает их ясность и содержит опасность появления в тексте модуля дополнительных ошибок. Поэтому можно рекомендовать избегать употребления оператора перехода всюду, где это возможно, но не ценой ясности программы.

К полезным случаям использования оператора перехода можно отнести выход из цикла или процедуры по особому условию, «досрочно» прекращающего работу данного цикла или данной процедуры, т.е. завершающего работу некоторой структурной единицы (обобщенного оператора) и тем самым лишь локально нарушающего структурированность программы. Большие трудности (и усложнение структуры) вызывает структурная реализация реакции на возникающие исключительные (часто ошибочные) ситуации, так как при этом требуется не только осуществить досрочный выход из структурной единицы, но и произвести необходимую обработку (исключение) этой ситуации (например, выдачу подходящей диагностической информации). Обработчик исключительной ситуации может находиться на любом уровне структуры программы, а обращение к нему может производиться с разных нижних уровней. Вполне приемлемой с технологической точки зрения является следующая «неструктурная» реализация реакции на исключительные ситуации [1]. Обработчики исключительных ситуаций помещаются в конце той или иной структурной единицы и каждый такой обработчик программируется таким образом, что после окончания своей работы

производит выход из той структурной единицы, в конце которой он помещен. Обращение к такому обработчику производится оператором перехода из данной структурной единицы (включая любую вложенную в нее структурную единицу).

## 3.6. Объектно-ориентированное программирование

*Объектно-ориентированное*, или *объектное*, программирование (ООП) – парадигма программирования, в которой основными концепциями являются понятия объектов и классов. В случае языков с прототипированием вместо классов используются объекты-прототипы [2].

### 3.6.1. История

ООП возникло в результате развития идеологии процедурного программирования, где данные и подпрограммы (процедуры, функции) их обработки формально не связаны. Для дальнейшего развития объектно-ориентированного программирования часто большое значение имеют понятия события (так называемое событийно-ориентированное программирование) и компонента (компонентное программирование, КОП).

Формирование КОП от ООП произошло, как случилось формирование модульного от процедурного программирования: процедуры сформировались в модули – независимые части кода до уровня сборки программы, так объекты сформировались в компоненты – независимые части кода до уровня выполнения программы. Взаимодействие объектов происходит посредством сообщений. Результатом дальнейшего развития ООП, по-видимому, будет агентно-ориентированое программирование, где *агенты* – независимые части кода на уровне выполнения. Взаимодействие агентов происходит посредством изменения *среды*, в которой они находятся.

Языковые конструкции, конструктивно не относящиеся непосредственно к объектам, но сопутствующие им для их безопасной (исключительные ситуации, проверки) и эффективной работы, инкапсулируются от них в аспекты (в аспектно-ориентированном программировании). Субъектно-ориентированное программирование расширяет понятие объект посредством обеспечения более унифицированного и независимого взаимодействия объектов. Может являться переходной стадией между ООП и агентным программирование в части самостоятельного их взаимодействия.

Первым языком программирования, в котором были предложены принципы объектной ориентированности, была Симула. В момент свое-

го появления (в 1967 году), этот язык программирования предложил поистине революционные идеи: объекты, классы, виртуальные методы и др., однако это всё не было воспринято современниками как нечто грандиозное. Тем не менее, большинство концепций были развиты Аланом Кэйем и Дэном Ингаллсом в языке *Smalltalk*. Именно он стал первым широко распространённым объектно-ориентированным языком программирования.

В настоящее время количество прикладных языков программирования (список языков), реализующих объектно-ориентированную парадигму, является наибольшим по отношению к другим парадигмам. В области системного программирования до сих пор применяется парадигма процедурного программирования, и общепринятым языком программирования является язык C. Хотя при взаимодействии системного и прикладного уровней операционных систем заметное влияние стали оказывать языки объектно-ориентированного программирования. Например, одной из наиболее распространенных библиотек мультиплатформенного программирования является объектно-ориентированная библиотека Qt, написанная на языке C++ [2].

### 3.6.2. Основные понятия

*Абстрагирование* – это способ выделить набор значимых характеристик объекта, исключая из рассмотрения незначимые. Соответственно, абстракция – это набор всех таких характеристик.

*Инкапсуляция* – это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали реализации от пользователя.

*Класс* является описываемой на языке терминологии (пространства имён) исходного кода моделью ещё не существующей сущности (объекта). Фактически он описывает устройство объекта, являясь своего рода чертежом. Говорят, что объект – это *экземпляр* класса. При этом в некоторых исполняющих системах класс также может представляться некоторым объектом при выполнении программы посредством динамической идентификации типа данных. Обычно классы разрабатывают таким образом, чтобы их объекты соответствовали объектам предметной области.

*Наследование* – это свойство системы, позволяющее описать новый класс на основе уже существующего, с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс – потомком, наследником или производным классом.

*Объект*. Сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса или копирования прототипа (например, после запуска результатов компиляции и связывания исходного кода на выполнение).

*Полиморфизм* – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

*Прототип* – это объект-образец, по образу и подобию которого создаются другие объекты. Объекты-копии могут сохранять связь с родительским объектом, автоматически наследуя изменения в прототипе; эта особенность определяется в рамках конкретного языка.

### 3.6.3. Основные концепции ООП

В центре ООП находится понятие объекта. *Объект* – это сущность, которой можно посылать сообщения, и которая может на них реагировать, используя свои данные. Данные объекта скрыты от остальной программы. Сокрытие данных называется инкапсуляцией.

Наличие инкапсуляции достаточно для объектности языка программирования, но ещё не означает его объектной ориентированности – для этого требуется наличие наследования.

Но даже наличие инкапсуляции и наследования не делает язык программирования в полной мере объектным с точки зрения ООП. Основные преимущества ООП проявляются только в том случае, когда в языке программирования реализован полиморфизм; т.е. возможность объектов с одинаковой спецификацией иметь различную реализацию.

Язык Self, соблюдая многие исходные положения объектно-ориентированного программирования, ввёл альтернативное классам понятие *прототипа*, положив начало прототипному программированию, считающемуся подвидом объектного.

ООП имеет уже более чем сорокалетнюю историю, но, несмотря на это, до сих пор не существует чёткого общепринятого определения данной технологии. Основные принципы, заложенные в первые объектные языки и системы, подверглись существенному изменению (или искажению) и дополнению при многочисленных реализациях последующего времени. Кроме того, примерно с середины 1980-х годов термин «объектно-ориентированный» стал модным, в результате с ним произошло то же самое, что несколько раньше с термином «структурный» (ставшим модным после распространения технологии структурного программирования) – его стали искусственно «прикреплять» к любым новым разработкам, чтобы обеспечить им привлекательность. Бьёрн

Страуструп в 1988 году писал, что обоснование «объектной ориентированности» чего-либо, в большинстве случаев, сводится к ложному силлогизму: «*X* – это хорошо. Объектная ориентированность – это хорошо. Следовательно, *X* является объектно-ориентированным».

Всё является объектом. Вычисления осуществляются путём взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некоторое действие. Объекты взаимодействуют, посылая и получая сообщения. Сообщение – это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия. Каждый объект имеет независимую память, которая состоит из других объектов. Каждый объект является представителем (экземпляром) класса, который выражает общие свойства объектов. В классе задаётся поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия. Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанное с экземплярами определённого класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве. Таким образом, программа представляет собой набор объектов, имеющих состояние и поведение. Объекты взаимодействуют посредством сообщений. Естественным образом выстраивается иерархия объектов: программа в целом – это объект, для выполнения своих функций она обращается к входящим в неё объектам, которые, в свою очередь, выполняют запрошенное путём обращения к другим объектам программы. Естественно, чтобы избежать бесконечной рекурсии в обращениях, на каком-то этапе объект трансформирует обращённое к нему сообщение в сообщения к стандартным системным объектам, предоставляемым языком и средой программирования.

Устойчивость и управляемость системы обеспечивается за счёт чёткого разделения ответственности объектов (за каждое действие отвечает определённый объект), однозначного определения интерфейсов межобъектного взаимодействия и полной изолированности внутренней структуры объекта от внешней среды (инкапсуляции)

Появление в ООП отдельного понятия *класса* закономерно вытекает из желания иметь множество объектов со сходным поведением. *Класс в ООП* – это в чистом виде абстрактный тип данных, создаваемый программистом. С этой точки зрения объекты являются значениями данного абстрактного типа, а определение класса задаёт внутреннюю структуру значений и набор операций, которые над этими значениями могут быть выполнены. Желательность иерархии классов (а значит, на-

следования) вытекает из требований к повторному использованию кода – если несколько классов имеют сходное поведение, нет смысла дублировать их описание, лучше выделить общую часть в общий родительский класс, а в описании самих этих классов оставить только различающиеся элементы.

Необходимость совместного использования объектов разных классов, способных обрабатывать однотипные сообщения, требует поддержки *полиморфизма* – возможности записывать разные объекты в переменные одного и того же типа. В таких условиях объект, отправляя сообщение, может не знать в точности, к какому классу относится адресат, и одни и те же сообщения, отправленные переменным одного типа, содержащим объекты разных классов, вызовут различную реакцию.

Отдельного пояснения требует понятие обмена *сообщениями*. Первоначально (например, в том же Smalltalk) взаимодействие объектов представлялось как «настоящий» обмен сообщениями, т.е. пересылка от одного объекта другому специального объекта-сообщения. Такая модель является чрезвычайно общей. Она прекрасно подходит, например, для описания параллельных вычислений с помощью *активных объектов*, каждый из которых имеет собственный поток исполнения и работает одновременно с прочими. Такие объекты могут вести себя как отдельные, абсолютно автономные вычислительные единицы. Посылка сообщений естественным образом решает вопрос обработки сообщений объектами, присвоенными полиморфным переменным – независимо от того, как объявляется переменная, сообщение обрабатывает код класса, к которому относится присвоенный переменной объект.

Однако общность механизма обмена сообщениями имеет и другую сторону – «полноценная» передача сообщений требует дополнительных накладных расходов, что не всегда приемлемо. Поэтому в большинстве ныне существующих объектно-ориентированных языков программирования используется концепция «*отправка сообщения как вызов метода*» – объекты имеют доступные извне методы, вызовами которых и обеспечивается взаимодействие объектов. Данный подход реализован в огромном количестве языков программирования, в том числе C++, Object Pascal, Java, Oberon-2. В настоящий момент именно он является наиболее распространённым в объектно-ориентированных языках.

Концепция *виртуальных методов*, поддерживаемая этими и другими современными языками, появилась как средство обеспечения выполнения нужных методов при использовании полиморфных переменных, т.е., по сути, как попытка расширить возможности вызова методов для реализации части функциональности, обеспечиваемой механизмом обработки сообщений [3].

### 3.6.4. Особенности реализации

Как уже говорилось выше, в современных объектно-ориентированных языках программирования каждый объект является значением, относящимся к определённому классу. Класс представляет собой объявленный программистом составной тип данных, имеющий в составе.

**Поля данных**

Параметры объекта (конечно, не все, а только необходимые в программе), задающие его состояние (свойства объекта предметной области). Иногда поля данных объекта называют свойствами объекта, из-за чего возможна путаница. Физически поля представляют собой значения (переменные, константы), объявленные как принадлежащие классу.

**Методы**

Процедуры и функции, связанные с классом. Они определяют действия, которые можно выполнять над объектом такого типа, и которые сам объект может выполнять.

Классы могут наследоваться друг от друга. Класс-потомок получает все поля и методы класса-родителя, но может дополнять их собственными либо переопределять уже имеющиеся. Большинство языков программирования поддерживает только единичное наследование (класс может иметь только один класс-родитель), лишь в некоторых допускается множественное наследование – порождение класса от двух или более классов-родителей. Множественное наследование создаёт целый ряд проблем, как логических, так и чисто реализационных, поэтому в полном объёме его поддержка не распространена. Вместо этого в 1990-е годы появилось и стало активно вводиться в объектно-ориентированные языки понятие интерфейса. Интерфейс – это класс без полей и без реализации, включающий только заголовки методов. Если некий класс наследует (или, как говорят, реализует) интерфейс, он должен реализовать все входящие в него методы. Использование интерфейсов предоставляет относительно дешёвую альтернативу множественному наследованию.

Взаимодействие объектов в абсолютном большинстве случаев обеспечивается вызовом ими методов друг друга.

*Инкапсуляция обеспечивается* следующими средствами:

1. *Контроль доступа.* Поскольку методы класса могут быть как чисто внутренними, обеспечивающими логику функционирования объекта, так и внешними, с помощью которых взаимодействуют объекты, необходимо обеспечить скрытость первых при доступности извне вторых. Для этого в языки вводятся специальные синтаксические конструк-

ции, явно задающие область видимости каждого члена класса. Традиционно это модификаторы public, protected и private, обозначающие, соответственно, открытые члены класса, члены класса, доступные только из классов-потомков и скрытые, доступные только внутри класса. Конкретная номенклатура модификаторов и их точный смысл различаются в разных языках.

2. *Методы доступа* . Поля класса, в общем случае, не должны быть доступны извне, поскольку такой доступ позволил бы произвольным образом менять внутреннее состояние объектов. Поэтому поля обычно объявляются скрытыми (либо язык в принципе не позволяет обращаться к полям класса извне), а для доступа к находящимся в полях данным используются специальные методы, называемые методами доступа. Такие методы либо возвращают значение того или иного поля, либо производят запись в это поле нового значения. При записи метод доступа может проконтролировать допустимость записываемого значения и, при необходимости, произвести другие манипуляции с данными объекта, чтобы они остались корректными (внутренне согласованными). Методы доступа называют ещё аксессорами (от англ. *access* – доступ), а по отдельности – геттерами (англ. *get* – чтение) и сеттерами (англ. *set* – запись).

3. *Свойства объекта*. Псевдополя, доступные для чтения и/или записи. Свойства внешне выглядят как поля и используются аналогично доступным полям (с некоторыми исключениями), однако фактически при обращении к ним происходит вызов методов доступа. Таким образом, свойства можно рассматривать как «умные» поля данных, сопровождающие доступ к внутренним данным объекта какими-либо дополнительными действиями (например, когда изменение координаты объекта сопровождается его перерисовкой на новом месте). Свойства, по сути – не более чем *синтаксический сахар*, поскольку никаких новых возможностей они не добавляют, а лишь скрывают вызов методов доступа. Конкретная языковая реализация свойств может быть разной. Например, в C# объявление свойства непосредственно содержит код методов доступа, который вызывается только при работе со свойствами, т.е. не требует отдельных методов доступа, доступных для непосредственного вызова. В Delphi объявление свойства содержит лишь имена методов доступа, которые должны вызываться при обращении к полю. Сами методы доступа представляют собой обычные методы с некоторыми дополнительными требованиями к сигнатуре.

Полиморфизм реализуется путём введения в язык правил, согласно которым переменной типа «класс» может быть присвоен объект любого класса-потомка её класса.

### 3.6.5. Подходы к проектированию программ в целом

ООП ориентировано на разработку крупных программных комплексов, разрабатываемых командой программистов (возможно, достаточно большой). Проектирование системы в целом, создание отдельных компонент и их объединение в конечный продукт при этом часто выполняется разными людьми, и нет ни одного специалиста, который знал бы о проекте всё.

Объектно-ориентированное проектирование состоит в описании структуры и поведения проектируемой системы, т.е. фактически в ответе на два основных вопроса:

- Из каких частей состоит система.
- В чём состоит ответственность каждой из частей.

Выделение частей производится таким образом, чтобы каждая имела минимальный по объёму и точно определённый набор выполняемых функций (обязанностей), и при этом взаимодействовала с другими частями как можно меньше.

Дальнейшее уточнение приводит к выделению более мелких фрагментов описания. По мере детализации описания и определения ответственности выявляются данные, которые необходимо хранить, наличие близких по поведению агентов, которые становятся кандидатами на реализацию в виде классов с общими предками. После выделения компонентов и определения интерфейсов между ними реализация каждого компонента может проводиться практически независимо от остальных (разумеется, при соблюдении соответствующей технологической дисциплины).

Большое значение имеет правильное построение иерархии классов. Одна из известных проблем больших систем, построенных по ООП-технологии – так называемая *проблема хрупкости базового класса*. Она состоит в том, что на поздних этапах разработки, когда иерархия классов построена и на её основе разработано большое количество кода, оказывается трудно или даже невозможно внести какие-либо изменения в код базовых классов иерархии (от которых порождены все или многие работающие в системе классы). Даже если вносимые изменения не затронут интерфейс базового класса, изменение его поведения может непредсказуемым образом отразиться на классах-потомках. В случае крупной системы разработчик базового класса не просто не в состоянии предугадать последствия изменений, он даже не знает о том, как именно базовый класс используется и от каких особенностей его поведения зависит корректность работы классов-потомков.

### 3.6.6. Родственные методологии

Компонентное программирование – следующий этап развития ООП; прототип- и класс-ориентированное программирование – разные подходы к созданию программы, которые могут комбинироваться, имеющие свои преимущества и недостатки.

**Компонентное программирование**

Компонентно-ориентированное программирование – это своеобразная «надстройка» над ООП, набор правил и ограничений, направленных на построение крупных развивающихся программных систем с большим временем жизни. Программная система в этой методологии представляет собой набор компонентов с хорошо определёнными интерфейсами. Изменения в существующую систему вносятся путём создания новых компонентов в дополнение или в качестве замены ранее существующих. При создании новых компонентов на основе ранее созданных запрещено использование наследования реализации – новый компонент может наследовать лишь интерфейсы базового. Таким образом компонентное программирование обходит проблему хрупкости базового класса [8].

**Прототипное программирование**

Прототипное программирование, сохранив часть черт ООП, отказалось от базовых понятий – класса и наследования.

Вместо механизма описания классов и порождения экземпляров язык предоставляет механизм создания объекта (путём задания набора полей и методов, которые объект должен иметь) и механизм клонирования объектов.

Каждый вновь созданный объект является «экземпляром без класса». Каждый объект может стать *прототипом* – быть использован для создания нового объекта с помощью операции *клонирования*. После клонирования новый объект может быть изменён, в частности, дополнен новыми полями и методами.

Клонированный объект либо становится полной копией прототипа, хранящей все значения его полей и дублирующей его методы, либо сохраняет ссылку на прототип, не включая в себя клонированных полей и методов до тех пор, пока они не будут изменены. В последнем случае среда исполнения обеспечивает механизм *делегирования* – если при обращении к объекту он сам не содержит нужного метода или поля данных, вызов передаётся прототипу, от него, при необходимости – дальше по цепочке.

### 3.6.7. Производительность объектных программ

Гради Буч указывает на следующие причины, приводящие к снижению производительности программ из-за использования объектно-ориентированных средств [2]:

*Динамическое связывание методов*.

Обеспечение полиморфного поведения объектов приводит к необходимости связывать методы, вызываемые программой (т.е. определять, какой конкретно метод будет вызываться) не на этапе компиляции, а в процессе исполнения программы, на что тратится дополнительное время. При этом реально динамическое связывание требуется не более чем для 20 % вызовов, но некоторые ООП-языки используют его постоянно.

*Значительная глубина абстракции*.

ООП-разработка часто приводит к созданию «многослойных» приложений, где выполнение объектом требуемого действия сводится к множеству обращений к объектам более низкого уровня. В таком приложении происходит очень много вызовов методов и возвратов из методов, что, естественно, сказывается на производительности.

*Наследование «размывает» код*.

Код, относящийся к «оконечным» классам иерархии наследования (которые обычно и используются программой непосредственно) – находится не только в самих этих классах, но и в их классах-предках. Относящиеся к одному классу методы фактически описываются в разных классах. Это приводит к двум неприятным моментам:

1. Снижается скорость трансляции, так как компоновщику приходится подгружать описания всех классов иерархии.

2. Снижается производительность программы в системе со страничной памятью – поскольку методы одного класса физически находятся в разных местах кода, далеко друг от друга, при работе фрагментов программы, активно обращающихся к унаследованным методам, система вынуждена производить частые переключения страниц.

*Инкапсуляция снижает скорость доступа к данным*.

Запрет на прямой доступ к полям класса извне приводит к необходимости создания и использования методов доступа. И написание, и компиляция, и исполнение методов доступа сопряжено с дополнительными расходами.

*Динамическое создание и уничтожение объектов*.

Динамически создаваемые объекты, как правило, размещаются в куче, что менее эффективно, чем размещение их на стеке и, тем более, статическое выделение памяти под них на этапе компиляции.

Несмотря на отмеченные недостатки, Буч утверждает, что выгоды от использования ООП более весомы. Кроме того, повышение производительности за счёт лучшей организации ООП-кода, по его словам, в некоторых случаях компенсирует дополнительные накладные расходы на организацию функционирования программы. Можно также заметить, что многие эффекты снижения производительности могут сглаживаться или даже полностью устраняться за счёт качественной оптимизации кода компилятором. Например, упомянутое выше снижение скорости доступа к полям класса из-за использования методов доступа устраняется, если компилятор вместо вызова метода доступа использует инлайн-подстановку (современные компиляторы делают это вполне уверенно).

### 3.6.8. Критика ООП

Несмотря на отдельные критические замечания в адрес ООП, в настоящее время именно эта парадигма используется в подавляющем большинстве промышленных проектов. Однако, нельзя считать, что ООП является наилучшей из методик программирования во всех случаях [6].

Обычно сравнивают объектное и процедурное программирование:

- Процедурное программирование лучше подходит для случаев, когда важны быстродействие и используемые программой ресурсы, но требует большего времени для разработки.
- Объектное – когда важна управляемость проекта и его модифицируемость, а также скорость разработки.

Критические высказывания в адрес ООП:

- Исследование Thomas E. Potok, Mladen Vouk и Andy Rindos [13] показало отсутствие значимой разницы в продуктивности разработки программного обеспечения между ООП и процедурным подходом.
- Кристофер Дэйт указывает на невозможность сравнения ООП и других технологий во многом из-за отсутствия строгого и общепризнанного определения ООП (C. J. Date, Introduction to Database Systems, 6th-ed., Page 650).
- Александр Степанов, в одном из своих интервью, указывал на то, что ООП «методологически неправильно» и что «… ООП практически такая же мистификация как и искусственный интеллект…» [2].
- Фредерик Брукс (Frederick P. Brooks, Jr.) в своей статье «No Silver Bullet. Essence and Accidents of Software Engineering» (Computer Magazine; April 1987) указывает на то, что наиболее сложной частью создания программного обеспечения является « … спецификация, дизайн и тестирование концептуальных конструкций, а отнюдь не работа по выражению этих концептуальных конструкций…». ООП (наряду с таки-

ми технологиями как искусственный интеллект, верификация программ, автоматическое программирование, графическое программирование, экспертные системы и др.), по его мнению, не является «серебряной пулей», которая могла бы на порядок величины (т.е. примерно в 10 раз, как говорится в статье) снизить сложность разработки программных систем. Согласно Бруксу, «…ООП позволяет сократить только привнесённую сложность в выражение дизайна. Дизайн остаётся сложным по своей природе…» [12].

- Эдсгер Дейкстра указывал: «…то о чём общество в большинстве случаев просит – это змеиное масло. Естественно, «змеиное масло» имеет очень впечатляющие имена, иначе будет очень трудно что-то продать: «Структурный анализ и Дизайн», «Программная инженерия», «Модели зрелости», «Управляющие информационные системы» (Management Information Systems), «Интегрированные среды поддержки проектов», «Объектная ориентированность», «Реинжиниринг бизнес-процессов«…» – *EWD 1175: The strengths of the academic enterprise.*
- Никлаус Вирт считает, что ООП – не более чем тривиальная надстройка над структурным программированием, и преувеличение её значимости, выражающееся, в том числе, во включении в языки программирования всё новых модных «объектно-ориентированных» средств, вредит качеству разрабатываемого программного обеспечения.
- Патрик Киллелиа в своей книге «Тюнинг веб-сервера» писал: «…ООП предоставляет вам множество способов замедлить работу ваших программ …» .
- Известная обзорная статья проблем современного ООП-программирования перечисляет некоторые типичные проблемы ООП – *Почему объектно-ориентированное программирование провалилось*.

Если попытаться классифицировать критические высказывания в адрес ООП, можно выделить несколько аспектов критики данного подхода к программированию.

### 3.6.9. Объектно-ориентированные языки

Многие современные языки специально созданы для облегчения объектно-ориентированного программирования. Однако следует отметить, что можно применять техники ООП и для необъектно-ориентированного языка и наоборот, применение объектно-ориентированного языка вовсе не означает, что код автоматически становится объектно-ориентированным.

Современный объектно-ориентированный язык предлагает, как правило, следующий обязательный набор синтаксических средств:

• Объявление классов с полями (данными – членами класса) и методами (функциями – членами класса).

• Механизм расширения класса (наследования) – порождение нового класса от существующего с автоматическим включением всех особенностей реализации класса-предка в состав класса-потомка. Большинство ООП-языков поддерживают только единичное наследование.

• Полиморфные переменные и параметры функций (методов), позволяющие присваивать одной и той же переменной экземпляры различных классов.

• Полиморфное поведение экземпляров классов за счёт использования виртуальных методов. В некоторых ООП-языках все методы классов являются виртуальными.

Минимальным традиционным объектно-ориентированным языком можно считать язык Оберон, который не содержит никаких других объектных средств, кроме вышеперечисленных (в исходном Обероне даже нет отдельного ключевого слова для объявления класса, а также отсутствуют явно описываемые методы, их заменяют поля процедурного типа). Но большинство языков добавляют к указанному минимальному набору те или иные дополнительные средства. В их числе:

1. Конструкторы, деструкторы, финализаторы.
2. Свойства (аксессоры).
3. Индексаторы.
4. Интерфейсы (например, в Java используются также как альтернатива множественному наследованию – любой класс может реализовать сколько угодно интерфейсов).
5. Переопределение операторов для классов.
6. Средства защиты внутренней структуры классов от несанкционированного использования извне. Обычно это модификаторы доступа к полям и методам, типа public, private, обычно также protected, иногда некоторые другие.

Часть языков (иногда называемых «чисто объектными») целиком построена вокруг объектных средств – в них любые данные (возможно, за небольшим числом исключений в виде встроенных скалярных типов данных) являются объектами, любой код – методом какого-либо класса, и невозможно написать программу, в которой не использовались бы объекты. Примеры подобных языков – Smalltalk, Java, C#, Ruby, AS3. Другие языки (иногда используется термин «гибридные») включают ООП-подсистему в исходно процедурный язык. В них существует возможность программировать, не обращаясь к объектным средствам. Классические примеры – C++, Delphi и Perl.

## 3.7. Обобщённое программирование

*Обобщённое программирование* – парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание. В том или ином виде поддерживается разными языками программирования. Возможности обобщённого программирования впервые появились в 70-х годах в языках CLU и Ada, а затем во многих объектно-ориентированных языках, таких как C++, Java, Object Pascal и языках для платформы .NET [12].

### 3.7.1. Общий механизм

Средства обобщённого программирования реализуются в языках программирования в виде тех или иных синтаксических средств, дающих возможность описывать данные (типы данных) и алгоритмы (процедуры, функции, методы), параметризуемые типами данных. У функции или типа данных явно описываются формальные параметры-типы. Это описание является обобщённым и в исходном виде непосредственно использовано быть не может.

В тех местах программы, где обобщённый тип или функция используется, программист должен явно указать фактический параметр-тип, конкретизирующий описание. Например, обобщённая процедура перестановки местами двух значений может иметь параметр-тип, определяющий тип значений, которые она меняет местами. Когда программисту нужно поменять местами два целых значения, он вызывает процедуру с параметром-типом «целое число» и двумя параметрами – целыми числами, когда две строки – с параметром-типом «строка» и двумя параметрами – строками. В случае с данными программист может, например, описать обобщённый тип «список» с параметром-типом, определяющим тип хранимых в списке значений. Тогда при описании реальных списков программист должен указать обобщённый тип и параметр-тип, получая, таким образом, любой желаемый список с помощью одного и того же описания.

Компилятор, встречая обращение к обобщённому типу или функции, выполняет необходимые процедуры статического контроля типов, оценивает возможность заданной конкретизации и при положительной оценке генерирует код, подставляя фактический параметр-тип на место формального параметра-типа в обобщённом описании. Естественно, что для успешного использования обобщённых описаний фактические типы-параметры должны удовлетворять определённым условиям. Если

обобщённая функция сравнивает значения типа-параметра, любой конкретный тип, использованный в ней, должен поддерживать операции сравнения, если присваивает значения типа-параметра переменным – конкретный тип должен обеспечивать корректное присваивание.

### 3.7.2. Способы реализации

Известно два основных способа реализации поддержки обобщённого программирования в компиляторе.

- Порождение нового кода для каждой конкретизации. В этом варианте компилятор рассматривает обобщённое описание как текстовый шаблон для создания вариантов конкретизаций. Когда компилятору требуется новая конкретизация обобщённого типа или процедуры, он создаёт новый экземпляр типа или процедуры, чисто механически добавляя туда тип-параметр. То есть, имея обобщённую функцию перестановки элементов, компилятор, встретив её вызов для целого типа, создаст функцию перестановки целых чисел и подставит в код её вызов, а затем, встретив вызов для строкового типа – создаст функцию перестановки строк, никак не связанную с первой. Этот метод обеспечивает максимальное быстродействие, поскольку варианты конкретизаций становятся разными фрагментами программы, каждый из них может быть оптимизирован для своего типа-параметра, к тому же в код не включаются никакие лишние элементы, связанные с проверкой или преобразованием типов на этапе исполнения программы. Недостатком его является то, что при активном использовании обобщённых типов и функций с различными типами-параметрами размер откомпилированной программы может очень сильно возрастать, поскольку даже для тех фрагментов описания, которые для разных типов не различаются, компилятор всё равно генерирует отдельный код. Этот недостаток можно затушевать путём частичной генерации общего кода (часть обобщённого описания, которая не зависит от типов-параметров, оформляется специальным образом и по ней компилятор генерирует единый для всех конкретизаций код). Зато данный механизм даёт естественную возможность создания специальных (обычно – сильно вручную оптимизированных) конкретизаций обобщённых типов и функций для некоторых типов-параметров.
- Порождение кода, который во время исполнения выполняет преобразование фактических параметров-типов к одному типу, с которым фактически и работает. В этом случае на этапе компиляции программы компилятор лишь проверяет соответствие типов и включает в код команды преобразования конкретного типа-параметра к общему типу. Код, определяющий функционирование обобщённого типа или функции, имеется в откомпилированной программе в единственном эк-

зкземпляре, а преобразования и проверки типов выполняются динамически, во время работы программы. В этом варианте порождается, как правило, более компактный код, но программа оказывается в среднем медленнее, чем в первом варианте, из-за необходимости выполнения дополнительных операций и меньших возможностей оптимизации. Кроме того, в компилированный код для типов-параметров далеко не всегда включается динамическая информация о типах (в первом варианте она есть, если вообще поддерживается, поскольку конкретизации для каждого типа-параметра различны), что определяет некоторые ограничения на применение обобщённых типов и функций. Подобные ограничения есть, например, в Java [12].

### 3.7.3. Обобщённое программирование в языке C++

В языке C++ обобщённое программирование основывается на понятии «шаблон», обозначаемом ключевым словом **template**. Широко применяется в стандартной библиотеке C++ (см. STL), а также в сторонних библиотеках boost, Loki. Большой вклад в появление развитых средств обобщённого программирования в C++ внёс Александр Степанов. [2]

Java предоставляет средства обобщённого программирования, синтаксически основанные на C++, начиная с версии J2SE 5.0. В этом языке имеются generics или «контейнеры типа T» – подмножество обобщённого программирования.

На платформе .NET средства обобщённого программирования появились в версии 2.0.

Поддержка средств обобщённого программирования появилась в Object Pascal в среде Delphi в октябре 2008 года. Основы поддержки обобщенных классов сначала появилась в Delphi 2007 .NET в 2006 году, но она затрагивала только .NET платформу. Более полная поддержка обобщенного программирования была добавлена в Delphi 2009. Обобщенные классы также поддерживаются в Object Pascal в системе PascalABC.NET

## 3.8 Вопросы для самоконтроля

1. Обозначение диска.
2. Название совокупности данных на диске, имеющей имя.
3. Обозначение винчестера.
4. Этап развития программирования как науки, при котором появился структурный подход к решению поставленной задачи.
5. Полный набор инструкций, описывающих последовательность действий некоторого исполнителя для достижения результата.

6. При каких условиях язык программирования является языком низкого уровня.
7. Суть структурного программирования.
8. Свойство, позволяющее описать новый класс на основе уже существующего, с частично или полностью заимствующейся функциональностью.
9. В каких годах впервые появились возможности обобщённого программирования?
10. Чем характеризуется обобщённое программирование.
11. Основные функции транслятора.
12. Основные конструкции структурного программирования.
13. Понятия, относящиеся к объектно-ориентированному программированию.
14. На каком этапе развития программирования появился термин «стихийное программирование».
15. Какую функцию в блок-схеме несет блок с названием «терминатор».

# 4. ОСНОВЫ ЯЗЫКА C++

В наше время, кажется, нет такой отрасли знаний, которая бы так стремительно развивалась, как програмимирование и вычислительная техника. Никакая еще наука не развивалась такими семимильными шагами и такими темпами. Возникает новая техника: компьютеры, процес-

соры, дисководы. Появляются новые возможности и новые информационные технологии.

Программирование сейчас везде и всюду. Оно обслуживает предприятия, оффисы, конторы, учебные заведения – все, где есть управленческий труд и потоки информации. Нелегок труд программиста. Трудны языки программирования. Особенно поражает их многообразие. И сам процесс программирования становится таким объемным и сложным, что старые методы уже никого не удовлетворяют, и на смену им приходят новые методы и новые языки программирования, подобные языку C++ и системе Visual C++ 6.0, способные убыстрить во много раз разработку и сопровождение программ. Сегодня мы смотрим назад из XXI-ого века в XX-й век и восхищаемся новейшими Windows-технологиями, визуальным подходом и объектно ориенированным программированием. За короткий срок они покорили и завоевали весь мир.

Немаловажную роль здесь играет язык программирования C++. Но зачем он был нужен, как и почему возник и был востребован? На эти и на другие вопросы мы и постараемся вместе с Вами найти правильные ответы на этом уроке, посвященном C.

C++ – расширение языка C – был разработан сотрудником научно-исследовательского центра AT&T Bell Laboratories (Нью-Джерси, США) Бьерном Строустропом в 1979 году. C++ содержит в себе все, что есть в C. Но, кроме того, он поддерживает объектно-ориентированное программирование (Object Oriented Programming, OOP). Изначально C++ был создан для того, чтобы облегчить разработку больших программ. Объектно-ориентированное программирование это новый подход к созданию программ. [11]

В 60-е годы XX века особо остро возникла потребность создавать большие и сложные программы. Однако, она натолкнулась на ряд трудностей. Люди, связанные с разработкой программ, начали понимать, что создание сложных программ – это гораздо более сложная задача, чем они себе представляли. Проведенные в этот период исследования привели к появлению и интенсивному развитию структурного программирования. Этот подход отличался большей дисциплинированностью, ясностью и простотой тестирования и отладки программ, легкостью их модификации.

Создание в 1971 году Никлаусом Виртом (швейцарским математиком) языка Паскаль было одним из замечательных результатов проводившихся исследований в ученой унивеситетской среде. Созданный первоначально исключительно для изучения структурного программирования в академической среде, он стал наиболее предпочитаемым языком во многих унивестиетах мира. Однако, отсутствие в нем необхо-

димых свойств для решения коммерческих задач сдерживало его применение в коммерции, в промышленности и управлении.

В течение 70-х и в начале 80-х годов при огромной заинтересованности и поддержке Министерства Обороны США был создан язык программирования Ада. Министерством Обороны США использовались сотни отдельных языков. Но все время хотелось иметь один язык, который бы удовлетворял всем интересам этого ведомства. Таким языком был выбран Паскаль. Но в итоге разработки язык Ада оказался совсем не похожим на Паскаль. Наиболее важное свойство Ады – многозадачность. Оно позволяет программистам разрабатывать алгоритмы параллельного выполнения действий.

Другие языки, как например С и С++, одновременно могли выполнять одно действие.

## 4.1. Типичная среда С++ программирования

Современные системы программирования на С++ состоят из нескольких составных частей. Это такие части, как сама среда программирования, язык, стандартная библиотека С-функций и различные библиотеки С-классов.

Как правило, чтобы выполнить программу на С++, необходимо пройти через 6 этапов: редактирование, препроцессорную (т.е. предварительную) обработку, компиляцию, компоновку, загрузку и выполнение. Мы с Вами остановим свое внимание на системе С++ программирования Borland C++ v. 5.0 или 5.2.

**Первый этап** представляет создание и редактирование файла с исходныи текстом программы. Он может выполняться с помощью простейшего редактора текстов программ. Программист набирает в этом редакторе свою С++ программу. При необходимости он снова обращается к ней и вносит с помощью этого редактора изменения в исходный текст программы. Далее программа запоминается на диске. Имена файлов С/С++ программ оканчиваются на «с» или «cpp». Однако, пакет программ Borland C++ v 5.0 (5.2) имеет встроенный редактор, которым также можно пользоваться.

**На втором этапе** компилятор начинает препроцессорную обработку текста программы прежде чем ее компилировать. Компилятор. Что он делает? Он переводит программу в машинный код, т.е. это объектный код программы.

Следует знать, что в системе С++ программирования перед началом этапа самой трансляции всегда выполняется программа предварительной обработки. Что она делает? Она отыскивает так называемые «дирек-

тивы трансляции» или «директивы препроцессора», которые указывают, какие нужно выполнить преобразования перед трансляцией исходного текста программы. Обычно это включение других текстовых файлов в файл, который подлежит компиляции. Препроцессорная обработка инициируется компилятором перед тем, как программа будет преобразована в машинный код. Это позволяет забирать нужные программы-функции в текст компилируемой программы до начала процесса компоновки.

**Третий этап** это компиляция. Как правило, программы на языке С++ содержат ссылки на различные функции, которые определены вне самой программы. Например, в стандартных библиотеках или в личных библиотеках программистов. Объектный код, созданный компилятором, содержит «дыры» на месте этих отсутствующих частей.

**Четвертый этап – компоновка.** Компоновщик связывает объектный код с кодами отсутствующих функций и создает, таким образом, исполняемый загрузочный модуль (без пропущенных «дыр»).

**Пятый этап – загрузка**. Перед выполнением программа должна быть размещена в памяти. Это делается с помощью загрузчика, который забирает загрузочный модуль программы с диска и перемещает его в память.

**Шестой этап – это выполнение**. Программа редко заработает с первой попытки. Каждый из названных этапов может заканчиваться ошибкой или неудачей из-за ошибки.

Тогда программист должен вернуться к редактированию исходного текста программы. Он должен внести необходимые изменения в текст программы, предварительно его хорошо проанализировав. Затем снова пройти через все этапы работы с исходным текстом программы до получения работающего без ошибок загрузочного модуля.

## 4.2. Структура программы на С++

Программа на языке С имеет следующую структуру [11]:

```
#директивы препроцессора
. . . . . .
#директивы препроцессора
функция a( )
 операторы
функция b( )
 операторы
void main ( )       //функция, с которой начинается выполнение программы
 операторы
```

```
описания
присваивания
функция
пустой оператор
    составной
    выбора
    циклов
    перехода
```

*Директивы препроцессора* – управляют преобразованием текста программы до ее компиляции. Исходная программа, подготовленная на С++ в виде текстового файла, проходит 3 этапа обработки (рис. 17):

1) препроцессорное преобразование текста;

2) компиляция;

3) компоновка (редактирование связей или сборка).

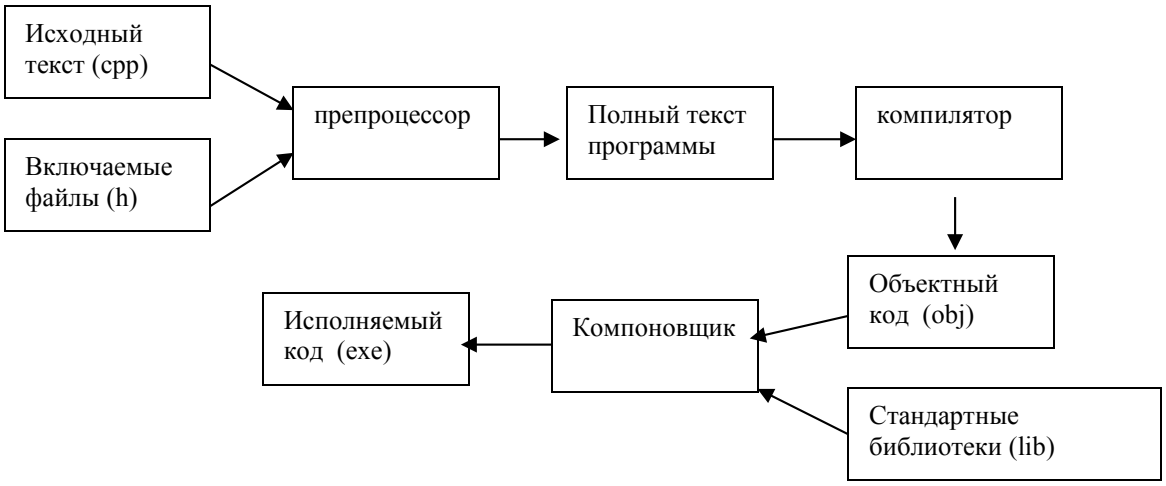



Рис. 17. Этапы создания исполняемого кода

После этих трех этапов формируется исполняемый код программы. *Задача препроцессора* – преобразование текста программы до ее компиляции. Правила препроцессорной обработки определяет программист с помощью директив препроцессора.

Директива начинается с #.

#define – указывает правила замены в тексте. Например, #define ZERO 0.0 означает, что каждое использование в программе имени ZERO будет заменяться на 0.0.

Директива #include < имя заголовочного файла> – предназначена для включения в текст программы текста из каталога «Заголовочных файлов», поставляемых вместе со стандартными библиотеками. Каждая библиотечная функция С++ имеет соответствующее описание в одном из заголовочных файлов. Список заголовочных файлов определен стан-

дартом языка. Употребление директивы include не подключает соответствующую стандартную библиотеку, а только позволяют вставить в текст программы описания из указанного заголовочного файла. Подключение кодов библиотеки осуществляется на этапе компоновки, т.е. после компиляции. Хотя в заголовочных файлах содержатся все описания стандартных функций, в код программы включаются только те функции, которые используются в программе.

После выполнения препроцессорной обработки в тексте программы не остается ни одной препроцессорной директивы.

Программа представляет собой набор описаний и определений, и состоит из *набора функций*. Среди этих функций всегда должна быть функция с именем *main*. Без нее программа не может быть выполнена. Перед именем функции помещаются сведения о типе возвращаемого функцией значения (тип результата). Если функция ничего не возвращает, то указывается тип void: void main ( ). Каждая функция, в том числе и main должна иметь набор параметров, он может быть пустым, тогда в скобках указывается (void).

За заголовком функции размещается тело функции. *Тело функции* – это последовательность определений, описаний и исполняемых операторов, заключенных в фигурные скобки. Каждое определение, описание или оператор заканчивается точкой с запятой.

*Определения* – вводят объекты (объект – это именованная область памяти, частный случай объекта - переменная), необходимые для представления в программе обрабатываемых данных.

**Пример. 10**

```
int y=10; //именованная константа
float x;   //переменная
```

*Описания* – уведомляют компилятор о свойствах и именах объектов и функций, описанных в других частях программы.

*Операторы* – определяют действия программы на каждом шаге ее исполнения.

**Пример 11.** Простейшая программа на языке С++

```
#include <stdio.h>  //препроцессорная директива
void main()       //функция
{             //начало
printf("Hello! ");    //печать
}             //конец
```

## 4.3. Базовые средства языка С++

### 4.3.1. Состав языка С++

В тексте на любом естественном языке можно выделить четыре основных элемента: символы, слова, словосочетания и предложения. Алгоритмический язык также содержит такие элементы, только слова называют лексемами (элементарными конструкциями), словосочетания – выражениями, предложения – операторами. Лексемы образуются из символов, выражения из лексем и символов, операторы из символов выражений и лексем (рис. 18).

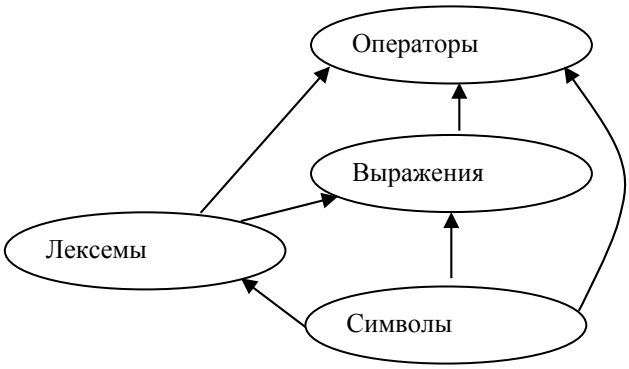



Рис. 18. Состав алгоритмического языка

Таким образом, элементами алгоритмического языка являются:

1. *Алфавит языка С++,* который включает

– прописные и строчные латинские буквы и знак подчеркивания;

– арабские цифры от 0 до 9;

– специальные знаки “{},| []()+–/%*.\’:;&?<>=!#^

– пробельные символы (пробел, символ табуляции, символы перехода на новую строку).

2. Из символов формируются лексемы языка:

*Идентификаторы* – имена объектов С-программ. В идентификаторе могут быть использованы латинские буквы, цифры и знак подчеркивания. Прописные и строчные буквы различаются, например, PROG1, prog1 и Prog1 – три различных идентификатора. Первым символом должна быть буква или знак подчеркивания (но не цифра). Пробелы в идентификаторах не допускаются.

*Ключевые* (зарезервированные) слова – это слова, которые имеют специальное значение для компилятора. Их нельзя использовать в качестве идентификаторов.

*Знаки операций* – это один или несколько символов, определяющих действие над операндами. Операции делятся на унарные, бинарные и тернарную по количеству участвующих в этой операции операндов.

*Константы* – это неизменяемые величины. Существуют целые, вещественные, символьные и строковые константы. Компилятор выделяет константу в качестве лексемы (элементарной конструкции) и относит ее к одному из типов по ее внешнему виду.

*Разделители* – скобки, точка, запятая пробельные символы.

#### 4.3.1.1. Константы в C++

*Константа* – это лексема, представляющая изображение фиксированного числового, строкового или символьного значения.

Константы делятся на пять групп:

1) целые;
2) вещественные (с плавающей точкой);
3) перечислимые;
4) символьные;
5) строковые.

Компилятор выделяет лексему и относит ее к той или другой группе, а затем внутри группы к определенному типу по ее форме записи в тексте программы и по числовому значению.

*Целые константы* могут быть десятичными, восьмеричными и шестнадцатеричными. Десятичная константа определяется как последовательность десятичных цифр, начинающаяся не с 0, если это число не 0 (примеры: 8, 0, 192345). Восьмеричная константа – это константа, которая всегда начинается с 0. За 0 следуют восьмеричные цифры (примеры: 016 – десятичное значение 14, 01). Шестнадцатеричные константы – последовательность шестнадцатеричных цифр, которым предшествуют символы 0x или 0X (примеры: 0xA, 0X00F).

В зависимости от значения целой константы компилятор по-разному представит ее в памяти компьютера (т.е. компилятор припишет константе соответствующий тип данных).

*Вещественные константы* имеют другую форму внутреннего представления в памяти компьютера. Компилятор распознает такие константы по их виду. Вещественные константы могут иметь две формы представления: с фиксированной точкой и с плавающей точкой.

1. Вид константы с *фиксированной* точкой:
   [**цифры].[цифры]** (примеры: 5.7, .0001, 41.).
2. Вид константы с *плавающей* точкой:
   **[цифры][.][цифры]E|e[+|-][цифры]** (примеры: 0.5e5, .11e-5, 5E3).

В записи вещественных констант может опускаться либо целая, либо дробная части, либо десятичная точка, либо признак экспоненты с показателем степени.

*Перечислимые константы* вводятся с помощью ключевого слова *enum*. Это обычные целые константы, которым приписаны уникальные и удобные для использования обозначения.

**Пример. 12.** Перечислимые константы.

```
enum { one=1, two=2, three=3,four=4};
enum {zero, one, two, three} /* – если в определении перечислимых констант опустить знаки = и числовые значения, то значения будут приписываться по умолчанию. При этом самый левый идентификатор получит значение 0, а каждый последующий будет увеличиваться на 1.*/
enum { ten=10, three=3, four, five, six};
enum {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday} ;
```

*Символьные константы* – это один или два символа, заключенные в апострофы. Символьные константы, состоящие из одного символа, имеют тип char и занимают в памяти один байт, символьные константы, состоящие из двух символов, имеют тип int и занимают два байта. Последовательности, начинающиеся со знака \ , называются *управляющими*, они используются:

1) для представления символов, не имеющих графического отображения, например:

*a* – звуковой сигнал,

*b* – возврат на один шаг,

*n* – перевод строки,

*t* – горизонтальная табуляция.

2) для представления символов: \ , ’ , ? , ” ( \\, \’ ,\? ,\” ).

3) для представления символов с помощью шестнадцатеричных или восьмеричных кодов (\073, \0xF5).

*Строковая константа* – это последовательность символов, заключенная в кавычки. Внутри строк также могут использоваться управляющие символы.

**Пример. 13.** Строковые константы.

```
“\nНовая строка”,
“\n\”Алгоритмические языки программирования высокого уровня \””
```

### 4.3.2. Типы данных в C++

Данные отображают в программе окружающий мир. Цель программы состоит в обработке данных. Данные различных типов хранятся и обрабатываются по-разному. Тип данных определяет:

1) внутреннее представление данных в памяти компьютера;

2) множество значений, которые могут принимать величины этого типа;

3) операции и функции, которые можно применять к данным этого типа.

В зависимости от требований задания программист выбирает тип для объектов программы. Типы C++ можно разделить на простые и составные. К простым типам относят типы, которые характеризуются одним значением. В C++ определено 6 простых типов данных:

*int* (целый)
*char* (символьный)
*wchar_t* (расширенный символьный)
*bool* (логический)

} целочисленные

*float* (вещественный)
*double* (вещественный с двойной точностью)

} с плавающей точкой (число=мантисса $x$ $10^{к}$)

Существует 4 спецификатора типа, уточняющих внутреннее представление и диапазон стандартных типов:

1) *short* (короткий)
2) *long* (длинный)
3) *signed* (знаковый)
4) *unsigned* (беззнаковый)

#### Тип int

Значениями этого типа являются целые числа.

Размер типа *int* не определяется стандартом, а зависит от компьютера и компилятора. Для 16-разрядного процессора под него отводится 2 байта, для 32-разрядного – 4 байта.

Если перед *int* стоит спецификатор short, то под число отводится 2 байта, а если спецификатор long, то 4 байта. От количества отводимой под объект памяти зависит множество допустимых значений, которые может принимать объект:

*short int* – занимает 2 байта, следовательно, имеет *диапазон*:

–32768 ... +32767;

*long int* – занимает 4 байта, следовательно, имеет *диапазон*:
–2 147 483 648 ... +2 147 483 647

Тип *int* совпадает с типом *short int* на 16-разрядных ПК и с типом *long int* на 32-разрядных ПК.

Модификаторы *signed* и *unsigned* также влияют на множество допустимых значений, которые может принимать объект:

*unsigned short int* – занимает 2 байта, следовательно, имеет диапазон:
0 … 65536;

*unsigned long int* – занимает 4 байта, следовательно, имеет диапазон:
0 ... +4 294 967 295.

### Тип char

Значениями этого типа являются элементы конечного упорядоченного множества символов. Каждому символу ставится в соответствие число, которое называется кодом символа. Под величину символьного типа отводится 1 байт. Тип char может использоваться со спецификаторами signed и unsigned. В данных типа signed char можно хранить значения в диапазоне от –128 до 127. При использовании типа unsigned char значения могут находиться в диапазоне от 0 до 255. Для кодировки используется код ASCII(American Standard Code foe International Interchange). Символы с кодами от 0 до 31 относятся к служебным и имеют самостоятельное значение только в операторах ввода-вывода.

Величины типа char также применяются для хранения чисел из указанных диапазонов.

### Тип wchar_t

Предназначен для работы с набором символов, для кодировки которых недостаточно 1 байта, например Unicode. Размер этого типа, как правило, соответствует типу short. Строковые константы такого типа записываются с префиксом L: `L"String #1"`.

### Тип bool

Тип bool называется логическим. Его величины могут принимать значения true и false. Внутренняя форма представления false – 0, любое другое значение интерпретируется как true.

### Типы с плавающей точкой

Внутреннее представление вещественного числа состоит из 2 частей: мантиссы и порядка. В IBM-совместимых ПК величины типа float занимают 4 байта, из которых один разряд отводится под знак мантиссы, 8 разрядов под порядок и 24 – под мантиссу.

Величины типы double занимают 8 байтов, под порядок и мантиссу отводятся 11 и 52 разряда соответственно. Длина мантиссы определяет точность числа, а длина порядка его диапазон.

Если перед именем типа double стоит спецификатор long, то под величину отводится байтов.

**Тип void**

К основным типам также относится тип void Множество значений этого типа – пусто.

### 4.3.3. Переменные

Переменная в C++ – именованная область памяти, в которой хранятся данные определенного типа. У переменной есть имя и значение. Имя служит для обращения к области памяти, в которой хранится значение. Перед использованием любая переменная должна быть описана.

**Пример. 14.** Описание переменных

```
int a; float x;
```

Общий вид оператора описания:

**[класс памяти][const]тип имя [инициализатор];**

*Класс памяти* может принимать значения: *auto*, *extern*, *static*, *register*. Класс памяти определяет время жизни и область видимости переменной. Если класс памяти не указан явно, то компилятор определяет его исходя из контекста объявления. Время жизни может быть постоянным – в течение выполнения программы или временным – в течение блока. Область видимости – часть текста программы, из которой допустим обычный доступ к переменной. Обычно область видимости совпадает с областью действия. Кроме того случая, когда во внутреннем блоке существует переменная с таким же именем.

*Const* – показывает, что эту переменную нельзя изменять (именованная константа).

При описании можно присвоить переменной начальное значение (инициализация).

**Классы памяти**:

*auto* –автоматическая локальная переменная. Спецификатор *auto* может быть задан только при определении объектов блока, например, в теле функции. Этим переменным память выделяется при входе в блок и освобождается при выходе из него. Вне блока такие переменные не существуют.

*extern* – глобальная переменная, она находится в другом месте программы (в другом файле или долее по тексту). Используется для создания переменных, которые доступны во всех файлах программы.

*static* – статическая переменная, она существует только в пределах того файла, где определена переменная.

*register* – аналогичны *auto*, но память под них выделяется в регистрах процессора. Если такой возможности нет, то переменные обрабатываются как auto.

**Пример 15.**

```
int a; //глобальная переменная
void main(){
int b;//локальная переменная
extern int x;//переменная x определена в другом
месте
static int c;//локальная статическая переменная
a=1;//присваивание глобальной переменной
int a;//локальная переменная a
a=2;//присваивание локальной переменной
::a=3;//присваивание глобальной переменной
}
int x=4;//определение и инициализация x
```

В примере переменная *a* определена вне всех блоков. Областью действия переменной *a* является вся программа, кроме тех строк, где используется локальная переменная a. Переменные *b* и *c* – локальные, область их видимости – блок. Время жизни различно: память под *b* выделяется при входе в блок (т.к. по умолчанию класс памяти *auto*), освобождается при выходе из него. Переменная c (*static*) существует, пока работает программа.

Если при определении начальное значение переменным не задается явным образом, то компилятор обнуляет глобальные и статические переменные. Автоматические переменные не инициализируются.

Имя переменной должно быть уникальным в своей области действия.

Описание переменной может быть выполнено или как объявление, или как определение. Объявление содержит информацию о классе памяти и типе переменной, определение вместе с этой информацией дает указание выделить память. В примере 15 строка *extern int x*; является объявлением, а остальные – определениями.

### 4.3.4. Знаки операций в С++

Знаки операций обеспечивают формирование *выражений*. Выражения состоят:

1) из операндов,

2) знаков операций,

3) скобок.

Каждый *операнд* является, в свою очередь, *выражением* или частным случаем выражения – *константой* или *переменной*.

Унарные операции приведены в табл. 9.

Таблица 9

Унарные операции

| Операция | Описание |
|---|---|
| & | Получение адреса операнда |
| * | Обращение по адресу (разыменование) |
| - | унарный минус, меняет знак арифметического операнда |
| ~ | поразрядное инвертирование внутреннего двоичного кода целочисленного операнда (побитовое отрицание) |
| ! | логическое отрицание (НЕ). В качестве логических значений используется 0 – ложь и не 0 – истина, отрицанием 0 будет 1, отрицанием любого ненулевого числа будет 0. |
| ++ | Увеличение на единицу:<br>*префиксная* операция – увеличивает операнд до его использования,<br>*постфиксная* операция увеличивает операнд после его использования |
| -- | уменьшение на единицу:<br>*префиксная* операция – уменьшает операнд до его использования,<br>*постфиксная* операция уменьшает операнд после его использования |
| sizeof | вычисление размера (в байтах) для объекта того типа, который имеет операнд<br>имеет две формы:<br>sizeof выражение;<br>sizeof (тип) |

**Пример 16.** Унарные операции.

```
int m=1,n=2;
int a=(m++)+n; // a=4,m=2,n=2
int b=m+(++n);//a=3,m=1,n=3
sizeof(float)//4
sizeof(1.0)//8, т.к. вещественные константы по умолчанию имеют тип double
```

Бинарные операции представлены в табл. 10.

Таблица 10

Бинарные операции

| Операция | Описание |
|---|---|
| *Аддитивные* | |
| + | бинарный плюс (сложение арифметических операндов) |
| - | бинарный минус (вычитание арифметических операндов) |
| *Мультипликативные* | |
| * | умножение операндов арифметического типа |
| / | деление операндов арифметического типа (если операнды целочисленные, то выполняется целочисленное деление) |
| % | получение остатка от деления целочисленных операндов |
| *Операции сдвига* (определены только для целочисленных операндов). Формат выражения с операцией сдвига:<br>**операнд_левый операция_сдвига операнд_правый** | |
| << | сдвиг влево битового представления значения левого целочисленного операнда на количество разрядов, равное значению правого операнда, освободившиеся разряды обнуляются |
| >> | сдвиг вправо битового представления значения правого целочисленного операнда на количество разрядов, равное значению правого операнда, освободившиеся разряды обнуляются, если операнд беззнакового типа и заполняются знаковым разрядом, если – знакового |
| *Поразрядные операции* | |
| & | поразрядная конъюнкция (И) битовых представлений значений целочисленных операндов (бит =1, если соответствующие биты обоих операндов=1) |
| \| | поразрядная дизъюнкция (ИЛИ) битовых представлений значений целочисленных операндов (бит =1, если соответствующий бит одного из операндов=1) |
| ^ | поразрядное исключающее ИЛИ битовых представлений значений целочисленных операндов(бит =1, если соответствующий бит только одного из операндов=1) |

Окончание табл. 10

| Операция | Описание |
|---|---|
| *Операции сравнения*: результатом являются true (не 0) или false (0) | |
| < | меньше, чем |
| > | больше, чем |
| <= | меньше или равно |
| >= | больше или равно |
| == | Равно |
| != | не равно |
| *Логические бинарные операции* | |
| && | конъюнкция (И) целочисленных операндов или отношений, цело- |

| | |
|---|---|
| | численный результат ложь(0) или истина(не 0) |
| \|\| | дизъюнкция (ИЛИ) целочисленных операндов или отношений, целочисленный результат ложь(0) или истина(не 0) |

*Операции присваивания* имеют следующие виды:
=, +=, -=, += и т.д.
Формат операции простого присваивания:
**операнд1=операнд2**

*Леводопустимое значение* (L-значение) – выражение, которое адресует некоторый участок памяти, т.е. в него можно занести значение. Это название произошло от операции присваивания, т.к. именно левая часть операции присваивания определяет, в какую область памяти будет занесен результат операции. Переменная – это частный случай леводопустимого выражения.

*Условная операция.* В отличие от унарных и бинарных операций в ней используется три операнда.

**Выражение1 ? Выражение2 : Выражение3;**

Первым вычисляется значение выражения1. Если оно истинно, то вычисляется значение выражения2, которое становится результатом. Если при вычислении выражения1 получится 0, то в качестве результата берется значение выражения3.

**Пример 17.** Условная операция

```
x<0 ? -x : x ; //вычисляется абсолютное значение x.
```

*Операция явного приведения* (*преобразования*) *типа.*
Существует две формы:
1) каноническая, общий вид: **(имя_типа) операнд**;
2) функциональная, общий вид: **имя_типа (операнд)**.

**Пример 18.** Операции явного преобразования типа

```
(int)a //каноническая форма
int(a) //функциональная форма
```

### 4.3.5. Выражения

Из констант, переменных, разделителей и знаков операций можно конструировать *выражения*. Каждое выражение представляет собой правило вычисления нового значения. Если выражение формирует целое или вещественное число, то оно называется арифметическим. Пара арифметических выражений, объединенная операцией сравнения, называется отношением. Если отношение имеет ненулевое значение, то оно

– истинно, иначе – ложно. Приоритеты операций в выражениях представлены в табл. 11.

Таблица 11

Приоритеты операций в выражениях

| Ранг | Операции |
|---|---|
| 1 | ( ) [ ] -> . |
| 2 | ! ~ - ++ -- & * (тип) sizeof тип( ) |
| 3 | * / % (мультипликативные бинарные) |
| | + - (аддитивные бинарные) |
| 5 | << >> (поразрядного сдвига) |
| 6 | < > <= >= (отношения) |
| 7 | == != (отношения) |
| 8 | & (поразрядная конъюнкция «И») |
| 9 | ^ (поразрядное исключающее «ИЛИ») |
| 10 | \| (поразрядная дизъюнкция «ИЛИ») |
| 11 | && (конъюнкция «И») |
| 12 | \|\| (дизъюнкция «ИЛИ») |
| 13 | ?: (условная операция) |
| 14 | = *= /= %= -= &= ^= \|= <<= >>= (операция присваивания) |
| 15 | , (операция запятая) |

### 4.3.6. Ввод и вывод данных

В языке С++ нет встроенных средств ввода и вывода – он осуществляется с помощью функций, типов и объектов, которые находятся в стандартных библиотеках. Существует два основных способа: функции, унаследованные из С и объекты С++.

Для ввода/вывода данных в *стиле С* используются функции, которые описываются в библиотечном файле *stdio.h*.

1) *printf* (форматная строка, список аргументов);

*Форматная строка* – строка символов, заключенных в кавычки, которая показывает, как должны быть напечатаны аргументы.

**Пример 19**

```
printf ("Значение числа Пи равно  %f\n", pi);
```

Форматная строка может содержать

1) символы печатаемые текстуально;

2) спецификации преобразования;

3) управляющие символы.

Каждому аргументу соответствует своя *спецификация преобразования*:

*%d*, *%i* – десятичное целое число;

*%f* – число с плавающей точкой;
*%e, %E* – число с плавающей точкой в экспоненциальной форме;
*%u* – десятичное число в беззнаковой форме;
*%c* – символ;
*%s* – строка.
В форматную строку также могут входить *управляющие символы*:
*\n* – управляющий символ новая строка;
*\t* – табуляция;
*\a* – звуковой сигнал и др.

Также в форматной строке могут использоваться модификаторы формата, которые управляют шириной поля, отводимого для размещения выводимого значения. *Модификаторы* – это числа, которые указывают минимальное количество позиций для вывода значения и количество позиций ля вывода дробной части числа:

**%[j]m[.p]C**,

где *j* – задает выравнивание по левому краю; *m* – минимальная ширина поля; *p* – количество цифр после запятой для чисел с плавающей точкой и минимальное количество выводимых цифр для целых чисел (если цифр в числе меньше, чем значение p, то выводятся начальные нули); *C* – спецификация формата вывода.

**Пример 20**

```
printf(«\nСпецификации формата:\n%10.5d - целое,\n%10.5f – с плавающей точкой \
\n%10.5e – в экспоненциальной форме \n%10s – строка»,10,10.0,10.0,»10»);
```

Будет выведено:
Спецификации формата:
00010 – целое
10.00000 – с плавающей точкой
1.00000e+001 – в экспоненциальной форме
10 – строка.

2) *scanf* (форматная строка, список аргументов);
В качестве аргументов используются адреса переменных.

**Пример 21**

```
scanf(" %d%f ", &x,&y);
```

При использовании *библиотеки* классов C++, используется библиотечный файл *iostream.h*, в котором определены стандартные потоки

ввода данных от клавиатуры *cin* и вывода данных на экран дисплея *cout*, а также соответствующие операции:

1) << – операция записи данных в поток;
2) >> – операция чтения данных из потока.

**Пример 22**

```
#include <iostream.h>;
. . . . . .
cout << "\nВведите количество элементов: ";
cin >> n;
```

## 4.4. Основные операторы языка С++

### 4.4.1. Базовые конструкции структурного программирования

В теории программирования доказано, что программу для решения задачи любой сложности можно составить только из трех структур: линейной, разветвляющейся и циклической. Эти структуры называются базовыми конструкциями структурного программирования.

*Линейной* называется конструкция, представляющая собой последовательное соединение двух или более операторов.

*Ветвление* – задает выполнение одного из двух операторов, в зависимости от выполнения какого либо условия.

*Цикл* – задает многократное выполнение оператора.

Целью использования базовых конструкций является получение программы простой структуры. Такую программу легко читать, отлаживать и при необходимости вносить в нее изменения. Операторы управления работой программы называют управляющими конструкциями программы. К ним относят:

1) составные операторы;
2) операторы выбора;
3) операторы циклов;
4) операторы перехода.

### 4.4.2. Оператор «выражение»

Любое выражение, заканчивающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении этого выражения. Частным случаем выражения является пустой оператор «;».

**Пример. 22.** Операторы «выражение».

```
i++;
a+=2;
x=a+b;
```

### 4.4.3. Составные операторы

К составным операторам относят собственно *составные операторы* и *блоки*. В обоих случаях это последовательность операторов, заключенная в фигурные скобки. Блок отличается от составного оператора наличием определений в теле блока.

**Пример 23.** Составные операторы.

```
{
n++;                // это составной оператор
summa+=n;
}

{
int n=0;
n++;                // это блок
summa+=n;
}
```

### 4.4.4. Операторы выбора

Операторы выбора – это условный оператор и переключатель.

1. *Условный оператор* имеет полную и сокращенную форму.

Сокращённая форма:

**if (выражение-условие) оператор;**

В качестве выражения-условия могут использоваться арифметическое выражение, отношение и логическое выражение. Если значение выражения-условия отлично от нуля (т.е. истинно), то выполняется оператор.

**Пример 24.** Сокращённая форма условного оператора

```
if (x<y&&x<z)min=x;
```

Полная форма:

**if ( выражение-условие ) оператор1;**
**else оператор2;**

Если значение выражения-условия отлично от нуля, то выполняется оператор1, при нулевом значении выражения-условия выполняется оператор2.

**Пример 25.** Полная форма условного оператора.

```
if (d>=0)
{
x1=(-b-sqrt(d))/(2*a);
x2=(-b+sqrt(d))/(2*a);
cout<< "\nx1="<<x1<<"x2="<<x2;
}
else cout<<"\nРешения нет";
```

2.*Переключатель* определяет множественный выбор.

**switch (выражение)**
**{**
**case константа1 : оператор1 ;**
**case константа2 : оператор2 ;**
**. . . . . .**
**[default: операторы;]**
**}**

При выполнении оператора *switch*, вычисляется выражение, записанное после *switch*, оно должно быть целочисленным. Полученное значение последовательно сравнивается с константами, которые записаны следом за *case*. При первом же совпадении выполняются операторы, помеченные данной меткой. Если выполненные операторы не содержат оператора перехода, то далее выполняются операторы всех следующих вариантов, пока не появится оператор перехода или не закончится переключатель. Если значение выражения, записанного после *switch*, не совпало ни с одной константой, то выполняются операторы, которые следуют за меткой *default*. Метка *default* может отсутствовать.

**Пример 26.** Переключатель.

```
#include <iostream.h>
void main()
{
 int i;
 cout<<«\nEnter the number»;
 cin>>i;
 switch(i)
 {
 case 1:cout<<«\nthe number is one»;
 case 2:cout<<«\n2*2=«<<i*i;
```

```
 case 3: cout<<«\n3*3=«<<i*i;break;
 case 4: cout<<«\n»<<i<<« is very beautiful!»;
 default:cout<<«\nThe end of work»;
 }
}
```

Результаты работы программы:

1. При вводе 1 будет выведено:

The number is one
2*2=1
3*3=1

2. При вводе 2 будет выведено:

2*2=4
3*3=4

3. При вводе 3 будет выведено:

3*3=9

4. При вводе 4 будет выведено:

4 is very beautiful!

5. При вводе всех остальных чисел будет выведено:

The end of work

### 4.4.5. Операторы циклов

Различают:

1) итерационные циклы;

2) арифметические циклы.

Группа действий, повторяющихся в цикле, называется его *телом*. Однократное выполнение цикла называется его *шагом*.

В *итерационных* циклах известно условие выполнения цикла.

1. Цикл с предусловием:

**while (выражение-условие)**
**оператор;**

В качестве <выражения-условия> чаще всего используется отношение или логическое выражение. Если оно истинно, т.е. не равно 0, то тело цикла выполняется до тех пор, пока выражение-условие не станет ложным.

**Пример 27.** Цикл с предусловием.

```
while (a!=0)
{
cin>>a;
s+=a;
}
```

2. Цикл с постусловием:

**do**

**оператор**

**while (выражение-условие);**

Тело цикла выполняется до тех пор, пока выражение-условие истинно.

**Пример 28.** Цикл с постусловием.

```
do
{
cin>>a;
s+=a;
}
while(a!=0);
```

3. Цикл с параметром:

**for ( выражение_1;выражение-условие;выражение_3)**

**оператор;**

В данных циклах выражение_1 и выражение_3 могут состоять из нескольких выражений, разделенных запятыми. Выражение_1 – задает начальные условия для цикла (инициализация). Выражение-условие> определяет условие выполнения цикла, если оно не равно 0, цикл выполняется, а затем вычисляется значение выражения_3. Выражение_3 – задает изменение параметра цикла или других переменных (коррекция). Цикл продолжается до тех пор, пока выражение-условие не станет равно 0. Любое выражение может отсутствовать, но разделяющие их «;» должны быть обязательно.

**Пример 28.** Использования цикла с параметром.

1) Уменьшение параметра:

```
for ( n=10; n>0; n--)
{ оператор};
```

2) Изменение шага корректировки:

```
for ( n=2; n>60; n+=13)
{ оператор };
```

3) Возможность проверять условие отличное от условия, которое налагается на число итераций:

```
for ( num=1;num*num*num<216; num++)
{ оператор };
```

4) Коррекция может осуществляться не только с помощью сложения или вычитания:

```
for ( d=100.0; d<150.0;d*=1.1)
```

```
{ <тело цикла>};
for (x=1;y<=75;y=5*(x++)+10)
{ оператор };
```

5) Можно использовать несколько инициализирующих или корректирующих выражений:

```
for (x=1, y=0; x<10;x++;y+=x);
```

### 4.4.6. Операторы перехода

Операторы перехода выполняют безусловную передачу управления.

1) *break* – оператор прерывания цикла

**{**
**< операторы>**
**if (<выражение_условие>) break;**
**<операторы>**
**}**

То есть оператор *break* целесообразно использовать, когда условие продолжения итераций надо проверять в середине цикла.

**Пример 29**

```
// ищет сумму чисел вводимых с клавиатуры до тех пор, пока не будет введено 100 чисел или 0
for(s=0, i=1; i<100;i++)
{
cin>>x;
if( x==0) break; // если ввели 0, то суммирование заканчивается
s+=x;
}
```

2) *continue* – переход к следующей итерации цикла. Он используется, когда тело цикла содержит ветвления.

**Пример 30**

```
//ищет количество и сумму положительных чисел
for( k=0,s=0,x=1;x!=0;)
{
cin>>x;
if (x<=0) continue;
k++;s+=x;
}
```

3) Оператор *goto*

Оператор *goto* имеет формат: **goto метка;**

В теле той же функции должна присутствовать конструкция:

**метка: оператор**;

*Метка* – это обычный идентификатор, областью видимости которого является функция. Оператор goto передает управления оператору, стоящему после метки. Использование оператора *goto* оправдано, если необходимо выполнить переход из нескольких вложенных циклов или переключателей вниз по тексту программы или перейти в одно место функции после выполнения различных действий.

Применение *goto* нарушает принципы структурного и модульного программирования, по которым все блоки, из которых состоит программа, должны иметь только один вход и только один выход.

Нельзя передавать управление внутрь операторов *if*, *switch* и циклов. Нельзя переходить внутрь блоков, содержащих инициализацию, на операторы, которые стоят после инициализации.

**Пример 31**

```
int k;
goto m;
. . .
{
int a=3,b=4;
k=a+b;
m: int c=k+1;
. . .
}
```

В этом примере при переходе на метку *m* не будет выполняться инициализация переменных *a* , *b* и *k*.

4) Оператор *return* – оператор возврата из функции. Он всегда завершает выполнение функции и передает управление в точку ее вызова. Вид оператора:

**return [выражение];**

## 4.5. Примеры решения задач с использованием основных операторов С++

«Начинающие программисты, особенно студенты, часто пишут программы так: получив задание, тут же садятся за компьютер и начинают кодировать те фрагменты алгоритма, которые им удается придумать сразу. Переменным дают первые попавшиеся имена типа *x* и *y*. Когда компьютер зависает, делается перерыв, после которого все написанное стирается, и все повторяется заново. Периодически высказываются сомнения в правильности работы компилятора, компьютера и опе-

рационной системы. Когда программа доходит до стадии выполнения, в нее вводятся произвольные значения, после чего экран становится объектом пристального удивленного изучения. «Работает» такая программа обычно только в бережных руках хозяина на одном наборе данных,
а внесение в нее изменений может привести автора к потере веры в себя и ненависти к процессу программирования.

Ваша задача состоит в том, чтобы научиться подходить к программированию профессионально. В конце концов, профессионал отличается тем, что может достаточно точно оценить, сколько времени у него займет написание программы, которая будет работать в полном соответствии с поставленной задачей. Кроме «ума, вкуса и терпения», для этого требуется опыт, а также знание основных принципов, выработанных программистами в течение более, чем полувека развития этой дисциплины. Даже к написанию самых простых программ нужно подходить последовательно, соблюдая определенную дисциплину.» [3].

Решение задач по программированию предполагает ряд *этапов*:

1. Разработка математической модели. На этом этапе определяются исходные данные и результаты решения задачи, а также математические формулы, с помощью которых можно перейти от исходных данных к конечному результату.

2. Разработка алгоритма. Определяются действия, выполняя которые можно будет от исходных данных придти к требуемому результату.

3. Запись программы на некотором языке программирования. На этом этапе каждому шагу алгоритма ставится в соответствие конструкция выбранного алгоритмического языка.

4. Выполнение программы (исходный модуль → компилятор → объектный модуль → компоновщик → исполняемый модуль)

5. Тестирование и отладка программы. При выполнении программы могут возникнуть ошибки 3 типов:

а) синтаксические – исправляются на этапе компиляции;

б) ошибки исполнения программы (деление на 0, логарифм от отрицательного числа и т.п.) – исправляются при выполнении программы;

в) семантические (логические) ошибки – появляются из-за неправильно понятой задачи, неправильно составленного алгоритма.

Чтобы устранить эти ошибки программа должна быть выполнена на некотором наборе тестов. Цель процесса тестирования – определение наличия ошибки, нахождение места ошибки, ее причины и соответствующие изменения программы – исправление. Тест – это набор исходных данных, для которых заранее известен результат. Тест, выявивший ошибку, считается успешным. Отладка программы заканчивается, когда

достаточное количество тестов выполнилось неуспешно, т.е. программа на них выдала правильные результаты.

Для определения достаточного количества тестов существует два подхода. При первом подходе программа рассматривается как «черный ящик», в который передают исходные данные и получают результаты. Устройство самого ящика неизвестно. При этом подходе, чтобы осуществить полное тестирование, надо проверить программу на всех входных данных, что практически невозможно. Поэтому вводят специальные критерии, которые должны показать, какое конечное множество тестов является достаточным для программы. При первом подходе чаще всего используются следующие *критерии*:

1) тестирование классов входных данных, т.е. набор тестов должен содержать по одному представителю каждого класса данных:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| X= | 0 | 1 | 0 | 1 | –1 | 1 | –1 |
| Y= | 0 | 1 | 1 | 0 | 1 | –1 | –1 |

2) тестирование классов выходных данных, набор тестов должен содержать данные достаточные для получения по одному представителю из каждого класса выходных данных.

При втором подходе программа рассматривается как «белый ящик», для которого полностью известно устройство. Полное тестирование при этом подходе заканчивается после проверки всех путей, ведущих от начала программы к ее концу. Однако и при таком подходе полное тестирование программы невозможно, т.к. путей в программе с циклами бесконечное множество. При таком подходе используются следующие критерии:

1) тестирование команд. Набор тестов должен обеспечивать прохождение каждой команды не менее одного раза.

2) тестирование ветвей. Набор тестов в совокупности должен обеспечивать прохождение каждой ветви не менее одного раза. Это самый распространенный критерий в практике программирования.

### 4.5.1. Программирование ветвлений

**Задача № 1.** Определить, попадет ли точка с координатами ($x$, $y$) в заштрихованную область (рис. 19).

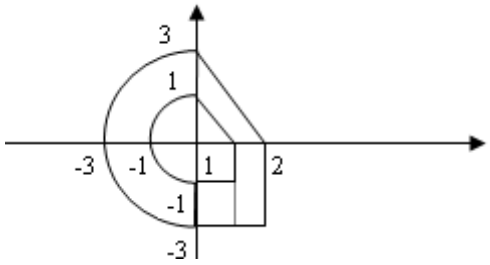


Рис. 19. Границы области для задачи №1

Исходные данные: $x$, $y$

Результат: да или нет

Математическая модель:

Ok=I || II || III || VI, где I, II, III, IV – условия попадания точки в заштрихованную область для каждого квадранта.

Квадрант I: Область формируется прямыми $0X$ и $0Y$, прямой, проходящей через точки (0,1) и (1,0) и прямой, проходящей через точки (0,3) и (2,0).

Необходимо определить уравнения прямых $y = a \cdot x + b$. Решаем две системы уравнений:

1) $\begin{cases} 1 = a \cdot 0 + b; \\ 0 = a \cdot 1 + b. \end{cases}$

2) $\begin{cases} 2 = a \cdot 0 + b; \\ 0 = a \cdot 3 + b. \end{cases}$

Из этих систем получаем следующие уравнения прямых:

$y = -1 \cdot x + 1$;

$y = -\frac{2}{3} \cdot x + 1$.

Тогда условие попадания точки в I квадрант будет выглядеть следующим образом:

**y>=-x+1&&y<=-2/3x+2&&y>=0&&x>=0**.

Квадранты II и III: Область формируется прямыми 0X и 0У и двумя окружностями, описываемыми формулами $x^2 + y^2 = 1$, $x^2 + y^2 = 9$.

Тогда условие попадания точки во II и III квадранты будет выглядеть следующим образом:

**x²+y²>=1&&x²+y²<=9&&x<=0**.

Квадрант IV:

Область формируется двумя прямоугольниками. Точка может попадать либо в первый прямоугольник, либо во второй.

Условие попадания точки в IV квадрант будет выглядеть следующим образом:

**(x>=0&&x<=1&&y<=-1&&y>=-3)||(x>=1&&x<=3&&y<=0&&y>=-3)**.

Программа:

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

void main()
{
 float x,y;
 cout<<«\nEnter x,y»;
 cin>>x>>y;
bool Ok=(y>=-x+1&&y<=2/3*x+2&&x>=0&&y>=0)||
     (pow(x,2)+pow(y,2)>=1&&pow(x,2)+pow(y,2)<=9&
 &x<=0)||
  (x>=0&&x<=1&&y<=-1&&y>=-3)||
(x>=1&&x<=2&&y<=0&&y>=-3);
 cout<<«\n»<<Ok;
}
```

Тесты приведены в табл. 12.

Таблица 12

Тесты к задаче №1

| Квадрант | Исходные данные (X,Y) | Результат (Ok) |
|---|---|---|
| I | 0.2, 0.2 | 0 |
| I | 0.7, 0.5 | 1 |
| II | –0.5, 0.5 | 0 |
| II | –2,0 | 1 |
| III | –0.5, –0,5 | 0 |
| III | –2, –1 | 1 |
| IV | 0,5, –0.5 | 0 |
| IV | 1.5, –1 | 1 |
| Центр системы координат | 0,0 | 0 |

### 4.5.2. Программирование арифметических циклов

Для арифметического цикла заранее известно сколько раз выполняется тело цикла.

**Задача № 2.** Дана последовательность целых чисел из *n* элементов. Найти среднее арифметическое этой последовательности.

Программа:

```
#include <iostream.h>
#include <math.h>
void main()
{
```

```
 int a,n,i,k=0;
 double s=0;
 cout<<«\nEnter n»;
 cin>>n;
      for(i=1;i<=n;i++)
      {
           cout<<«\nEnter a»;
           cin>>a;
           s+=a;k++;
      }
      s=s/k;
      cout<<«\nSr. arifm=«<<s<<«\n»;
}
```

Тесты приведены в табл. 13.

Таблица 13

Тесты к задаче № 2

| Параметр | Значение параметра |
|---|---|
| Количество цифр, *n* | 5 |
| Значение цифр, *a* | 1, 2, 3, 4, 5, 3 |
| Среднее арифметическое, *s* | 3 |

**Задача № 3.** Найти сумму чисел последовательности:
S = 1 + 2 + 3 + 4 + … + N
Программа:

```
#include <iostream.h>
#include <math.h>
void main()
{
 int n,i,s=0;
 cout<<«\nEnter n»;
 cin>>n;
if(n<=0) {cout<<”\nN<=0”;return;}
      for(i=1;i<=n;i++)s+=i;
      cout<<«\nS=«<<s<<«\n»;
}
```

Тесты приведены в табл. 14.

Таблица 14

Тесты к задаче № 3

| Значение параметр *n* | Значение параметра *s* |
|---|---|
| $n = -1$ | $N <= 0$ |

| | |
|---|---|
| $n = 0$ | $N <= 0$ |
| $n = 5$ | $S = 15$ |

**Задача №4.** Найти сумму последовательности вида:
S = 15 – 17 + 19 – 21 + . . ., всего *n* слагаемых.
Программа:

```
#include <iostream.h>
#include <math.h>
void main()
{
 int n,i,s=0,a=15;
 cout<<«\nEnter n»;
 cin>>n;
if(n<=0) {cout<<”\nN<=0”;return;}
for(i=1;i<=n;i++)
{
if(i%2==1)s+=a;
else s-=a;
a+=2;
}
cout<<«\nS=«<<s<<«\n»;
}
```

Тесты приведены в табл. 15.

Таблица 15

Тесты к задаче № 4

| **Значение параметр *n*** | **Значение параметра** s |
|---|---|
| $n = -1$ | $N <= 0$ |
| $N = 0$ | $N <= 0$ |
| $N = 3$ | $S = 17$ |

### 4.5.3. Итерационные циклы

Для итерационного цикла известно условие выполнения цикла.

**Задача № 5.** Дана последовательность целых чисел, за которой следует 0. Найти минимальный элемент этой последовательности.

Программа:

```
#include <iostream.h>
#include <math.h>
void main()
{
int a,min;
```

```
cout<<«\nEnter a»;
cin>>a;
min=a;
while(a!=0)//for(;a!=0;)
{
cout<<«\nEnter a»;
cin>>a;
if (a!=0&&a<min)min=a;
}
cout<<«\nmin=«<<min<<«\n»;
}
```

Тесты приведены в табл. 16.

Таблица 16

Тесты к задаче № 5

| **Последовательность *a*** | **min число** |
|---|---|
| 2 55 –3 –10 0 | –10 |
| 12 55 4 27 0 | 4 |
| –6 –43 –15 –10 0 | –10 |

**Задача № 6**. Найти сумму чисел Фибоначчи, меньших заданного числа *Q*.

Программа:

```
#include<iostream.h>
void main()
{
 int a=1,b=1,s=2,Q,c;
 cout<<«\nEnter Q»;
 cin>>Q;
 if(Q<=0)cout<<«Error in Q»;
 else
 if(Q==1)cout<<«\nS=1»;
 else
 {
c=a+b;
while(c<Q) //for(;c!=0;)
 {
      s+=c;
      a=b;
      b=c;
      c=a+b;
```

```
 }
 cout<<«\nS=«<<s<<«\n»;
}
}
```

Тесты приведены в табл. 17.

Таблица 17

Тесты к задаче № 6

| **Значение числа *Q*** | **Сумма *S*** |
|---|---|
| –1 | Error in Q |
| 0 | Error in Q |
| 1 | 1 |
| 2 | 2 |
| 10 | 20 |

### 4.5.4. Вложенные циклы

**Задача № 7**. Напечатать *N* простых чисел.

Программа:

```
#include<iostream.h>
void main()
{
 int a=1,n,d;
 cout<<«\nEnter N»;
 cin>>n;
 for(int i=0;i<n;)//внешний цикл
 {
 a++;d=1;
 do //внутренний цикл
 {
     d++;
 }
 while(a%d!=0);//конец внутреннего цикла
 if(a==d){
     cout<<a<<« «;
     i++;}

 }//конец внешнего цикла
}
```

## 4.6. Составные типы данных в С++

### 4.6.1. Массивы

В языке С/С++, кроме базовых типов, разрешено вводить и использовать производные типы, полученные на основе базовых. Стандарт языка определяет *три способа получения производных типов*:

1) массив элементов заданного типа;
2) указатель на объект заданного типа;
3) функция, возвращающая значение заданного типа.

*Массив* – это упорядоченная последовательность переменных одного типа. Каждому элементу массива отводится одна ячейка памяти. Элементы одного массива занимают последовательно расположенные ячейки памяти. Все элементы имеют одно имя – имя массива и отличаются индексами – порядковыми номерами в массиве. Количество элементов в массиве называется его размером. Чтобы отвести в памяти нужное количество ячеек для размещения массива, надо заранее знать его размер. Резервирование памяти для массива выполняется на этапе компиляции программы.

**Определение массива в С/С++**

```
int a[100];//массив из 100 элементов целого типа
```

Операция *sizeof*(*a*) даст результат 400, т.е.100 элементов по 4 байта.

Элементы массива всегда нумеруются с 0:

0 1 2 ….. 99

Чтобы обратиться к элементу массива, надо указать имя массива и номер элемента в массиве (индекс):

a[0] – индекс задается как константа,

a[55] – индекс задается как константа,

a[I] – индекс задается как переменная,

a[2*I] – индекс задается как выражение.

Элементы массива можно задавать при его определении:

```
int a[12]={1,2,3,4,5,6,7,8,9,10} ;
```

Операция sizeof(a) даст результат 40, т.е.10 элементов по 4 байта.

```
int a[12]={1,2,3,4,5};
```

Операция *sizeof*(*a*) даст результат 40, т.е. 10 элементов по 4 байта. Если количество начальных значений меньше, чем объявленная длина массива, то начальные элементы массива получат только первые элементы:

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

Операция *sizeof*(*a*) даст результат 20, т.е. 5 элементов по 4 байта. Длин массива вычисляется компилятором по количеству значений, перечисленных при инициализации.

**Обработка одномерных массивов**

При работе с массивами очень часто требуется одинаково обработать все элементы или часть элементов массива. Для этого организуется перебор массива.

*Перебор* элементов массива *характеризуется*:

1) направлением перебора;
2) количеством одновременно обрабатываемых элементов;
3) характером изменения индексов.

*По направлению* перебора массивы обрабатывают:

1) слева направо (от начала массива к его концу);
2) справа налево (от конца массива к началу);
3) от обоих концов к середине.

*Индексы* могут меняться:

1) линейно (с постоянным шагом);
2) нелинейно (с переменным шагом).

**Перебор массива по одному элементу**

Элементы можно перебирать:

1. Слева направо с шагом 1, используя цикл с параметром:

```
For (int I=0;I<n;I++) {обработка a[I];}
```

2. Слева направо с шагом отличным от 1, используя цикл с параметром:

```
for (int I=0;I<n;I+=step) {обработка a[I];}
```

3. Справа налево с шагом 1, используя цикл с параметром:

```
For (int I=n-1;I>=0;I--) {обработка a[I];}
```

4. Справа налево с шагом отличным от 1, используя цикл с параметром:

```
for (int I=n-1;I>=0;I-=step) {обработка a[I];}
```

**Формирование псевдодинамических массивов**

При описании массива в программе надо обязательно указывать количество элементов массива для того, чтобы компилятор выделил под этот массив нужное количество памяти. Это не всегда бывает удобно, т.к. число элементов в массиве может меняться в зависимости от решаемой задачи. Динамические массивы реализуются с помощью указателей (см. далее).

Псевдодинамические массивы реализуются следующим образом:

1) при определении массива выделяется достаточно большое количество памяти:

```
const int MAX_SIZE=100;//именованная константа
int mas[MAX_SIZE];
```

2) пользователь вводит реальное количество элементов массива меньшее N:

```
int n;
cout<<"\nEnter the size of
array<"<<MAX_SIZE<<":";
cin>>n;
```

3) дальнейшая работа с массивом ограничивается заданной пользователем размерностью *n* (рис. 19).

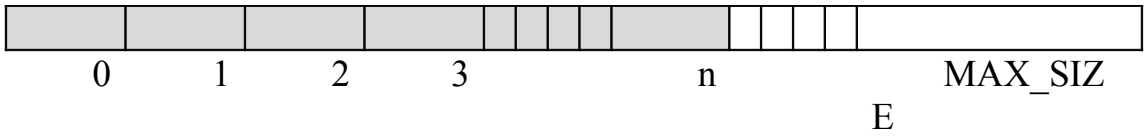



Рис. 19. Представление псевдодинамического массива

Таким образом, используется только часть массива.

**Использование датчика случайных чисел для формирования массива**

*Датчик случайных чисел* (ДСЧ) – это программа, которая формирует псевдослучайное число. Простейший ДСЧ работает следующим образом:

1)Берется большое число *K* и произвольное $x_0 \in [0,1]$.

2)Формируются числа $x_1$=дробная_часть($x_0$**K*);
$x_2$=дробная_часть($x_1$**K*); и т.д.

В результате получается последовательность чисел $x_0$, $x_1$, $x_2$,. . . беспорядочно разбросанных по отрезку от 0 до 1. Их можно считать случайными, а точнее псевдослучайными. Реальные ДСЧ реализуют более сложную функцию *f*(*x*).

В C++ имеется специальная функция

*int rand*() – возвращает псевдослучайное число из диапазона 0 ... RAND_MAX=32767, описание функции находится в файле <*stdlib.h*>.

**Пример 32.** Формирования и печати массива с помощью ДСЧ:

```
#include<iostream.h>
#include<stdlib.h>
void main()
```

```
{
int a[100];
int n;
cout<<”\nEnter the size of array:”;cin>>n;
for(int I=0;I<n;I++)
{a[I]=rand()%100-50;
cout<<a[I]<<” “;
}
}
```

В этой программе используется перебор массива по одному элементу слева направо с шагом 1.

**Пример 33.** Найти максимальный элемент массива.

```
#include<iostream.h>
#include<stdlib.h>
void main()
{
int a[100];
int n;
cout<<”\nEnter the size of array:”;cin>>n;
for(int I=0;I<n;I++)
{a[I]=rand()%100-50;
cout<<a[I]<<” “;
}
int max=a[0];
for(I=1;I<n;I++)
if (a[I]>max)max=a[I];
cout<<”\nMax=”<<max”;
}
```

В этой программе также используется перебор массива по одному элементу слева направо с шагом 1.

**Пример 34.** Найти сумму элементов массива с четными индексами.

```
#include<iostream.h>
#include<stdlib.h>
void main()
{
int a[100];
int n;
cout<<"\nEnter the size of array:";cin>>n;
for(int I=0;I<n;I++)
{a[I]=rand()%100-50;
cout<<a[I]<<" ";
}
int Sum=0;
for(I=0;I<n;I+=2)
Sum+=a[I];//элементы с индексами 0, 2, 4…
cout<<"\nSum="<<Sum";
}
```

Ввод массива

```
//Второй способ
for(I=0;I<n;I++)
if(I%2==0)Sum+=a[I];
//элементы с индексами 0, 2, 4…
cout<<"\nSum="<<Sum";
```

**Перебор массива по два элемента**

1) Элементы массива можно обрабатывать по два элемента, двигаясь с обеих сторон массива к его середине:

int I=0, J=N-1;
while( I<J)
{обработка a[I] и a[J];I++;J--;}

2) Элементы массива можно обрабатывать по два элемента, двигаясь от начала к концу с шагом 1(т.е. обрабатываются пары элементов a[1]и a[2], a[2]и a[3] и т.д.):

for (I=1;I<N;I++)
{обработка a[I] и a[I+1]}

3) Элементы массива можно обрабатывать по два элемента, двигаясь от начала к концу с шагом 2 (т.е. обрабатываются пары элементов a[1]и a[2], a[3]и a[4] и т.д.)

int I=1;
while (I<N-1 )
{обработка a[I] и a[I+1];
I+=2;}

**Классы задач по обработке массивов**

1. К задачам 1 класса относятся задачи, в которых выполняется однотипная обработка всех или указанных элементов массива.

2. К задачам 2 класса относятся задачи, в которых изменяется порядок следования элементов массива.

3. К задачам 3 класса относятся задачи, в которых выполняется обработка нескольких массивов или подмассивов одного массива. Массивы могут обрабатываться по одной схеме – синхронная обработка или по разным схемам – асинхронная обработка массивов.

4. К задачам 4 класса относятся задачи, в которых требуется отыскать первый элемент массива, совпадающий с заданным значением – поисковые задачи в массиве.

### Задачи 1-го класса

Решение таких задач сводится к установлению того, как обрабатывается каждый элемент массива или указанные элементы, затем подбирается подходящая схема перебора, в которую вставляются операторы обработки элементов массива.

**Пример 35.** Нахождение максимального элемента массива или среднего арифметического массива.

```
#include<iostream.h>
#include<stdlib.h>
void main()
{
int a[100];
int n;
cout<<"\nEnter the size of array:";cin>>n;
for(int I=0;I<n;I++)
{a[I]=rand()%100-50;
cout<<a[I]<<" ";
}
int Sum=0;
for(I=0;I<n;I++)
Sum+=a[I];
Cout<<"Среднее арифметическое="<<Sum/n";
}
```

### Задачи 2-го класса

Обмен элементов внутри массива выполняется с использованием вспомогательной переменной:

int R=a[I];a[I]=a[J]; a[J]:=R; // обмен a[I] и a[J] элементов массива.

**Пример 36.** Перевернуть массив.

```
//формирование массива
for(int i=0,j=n-1;i<j;i++,j--)
{int r=a[i];
a[i]=a[j];
a[j]=r;}
//вывод массива
```

**Пример 37.** Поменять местами пары элементов в массиве: 1 и 2, 3 и 4, 5 и 6 и т.д.

```
for(int i=0;i<n-1;i+=2)
{int r=a[i];
a[i]=a[i+1];
a[i+1]=r;}
```

**Пример 38.** Циклически сдвинуть массив на *k* элементов влево (вправо).

```
int k,i,t,r;
cout<<«\nK=?»;cin>>k;

for(t=0;t<k;t++)
{
      r=a[0];
      for(int i=0;i<n-1;i++)
          a[i]=a[i+1];
      a[n-1]=r;
}
```

### Задачи 3-го класса

При синхронной обработке массивов индексы при переборе массивов меняются одинаково.

**Пример 39.** Заданы два массива из *n* целых элементов. Получить массив *c*, где c[I]=a[I]+b[I].

```
For(int I=0;I<n;I++)c[I]=a[I]+b[I];
```

При асинхронной обработке массивов индекс каждого массива меняется по своей схеме.

**Пример 40.** В массиве целых чисел все отрицательные элементы перенести в начало массива.

```
int b[12];//вспомогательный массив
int i,j=0;
for(i=0;i<n;i++)
 if(a[i]<0){b[j]=a[i];j++;}//переписываем из a в
b все отрицательные элементы
```

```
for(i=0;i<n;i++)
 if(a[i]>=0){b[j]=a[i];j++;}// переписываем из a
в b все положительные элементы
for(i=0;i<n;i++) cout<<b[I]<<” “;
```

**Пример 41.** Удалить из массива все четные числа

```
int b[12];
 int i,j=0;
 for(i=0;i<n;i++)
     if(a[i]%2!=0){b[j]=a[i];j++;}

     for(i=0;i<j;i++) cout<<b[i]<<« «;
         cout<<«\n»;
```

### Задачи 4-го класса

В поисковых задачах требуется найти элемент, удовлетворяющий заданному условию. Для этого требуется организовать перебор массива и проверку условия. Но при этом существует две возможности выхода из цикла:

1) нужный элемент найден;

2) элемент не найден, но просмотр массива закончен.

**Пример 42.** Найти первое вхождение элемента *K* в массив целых чисел.

```
 int k;
 cout<<«\nK=?»;cin>>k;
 int ok=0;//признак найден элемент или нет
 int i,nom;
 for(i=0;i<n;i++)
     if(a[i]==k){ok=1;nom=i;break;}
 if(ok==1)
         cout<<«\nnom=«<<nom;
 else
     cout<<«\nthere is no such element!»;
```

### Сортировка массивов

*Сортировка* – это процесс перегруппировки заданного множества объектов в некотором установленном порядке.

Сортировки массивов подразделяются по быстродействию. Существуют простые методы сортировок, которые требуют n*n сравнений, где *n* – количество элементов массива и быстрые сортировки, которые требуют $n \cdot \ln(n)$ сравнений. Простые методы удобны для объяснения принципов сортировок, т.к. имеют простые и короткие алгоритмы.

Усложненные методы требуют меньшего числа операций, но сами операции более сложные, поэтому для небольших массивов простые методы более эффективны.

Простые методы подразделяются на три основные категории:

1) сортировка методом простого включения;

2) сортировка методом простого выделения;

3) сортировка методом простого обмена;

### Сортировка методом простого включения (вставки)

Элементы массива делятся на уже готовую последовательность и исходную. При каждом шаге, начиная с $I = 2$, из исходной последовательности извлекается $I$-й элемент и вставляется на нужное место готовой последовательности, затем $I$ увеличивается на 1 и т.д.

В процессе поиска нужного места осуществляются пересылки элементов больше выбранного на одну позицию вправо, т.е. выбранный элемент сравнивают с очередным элементом отсортированной части, начиная с $J = I - 1$. Если выбранный элемент больше $a[I]$, то его включают в отсортированную часть, в противном случае $a[J]$ сдвигают на одну позицию, а выбранный элемент сравнивают со следующим элементом отсортированной последовательности. Процесс поиска подходящего места заканчивается при двух различных условиях:

1) если найден элемент $a[J] > a[I]$;

2) достигнут левый конец готовой последовательности.

**Пример 43.** Сортировка методом вставки.

```
 int i,j,x;
 for(i=1;i<n;i++)
 {
x=a[i];//запомнили элемент, который будем встав-
лять
 j=i-1;
 while(x<a[j]&&j>=0)//поиск подходящего места
 {
a[j+1]=a[j];//сдвиг вправо
      j--;
}
 a[j+1]=x;//вставка элемента
 }
```

### Сортировка методом простого выбора

Выбирается минимальный элемент массива и меняется местами с первым элементом массива. Затем процесс повторяется с оставшимися элементами и т.д.

**Пример 44.** Сортировка методом выбора.

```
int i,min,n_min,j;
for(i=0;i<n-1;i++)
{
     min=a[i];n_min=i;//поиск минимального
     for(j=i+1;j<n;j++)
          if(a[j]<min){min=a[j];n_min=j;}
          a[n_min]=a[i];//обмен
          a[i]=min;
}
```

### Сортировка методом простого обмена

Сравниваются и меняются местами пары элементов, начиная с последнего. В результате самый маленький элемент массива оказывается самым левым элементом массива. Процесс повторяется с оставшимися элементами массива.

**Пример 45.** Сортировка методом простого обмена

```
for(int i=1;i<n;i++)
for(int j=n-1;j>=i;j--)
if(a[j]<a[j-1])
{int r=a[j];a[j]=a[j-1];a[j-1]=r;}
}
```

### Поиск в отсортированном массиве

В отсортированном массиве используется дихотомический (бинарный) поиск. При *последовательном поиске* требуется в среднем $n/2$ сравнений, где $n$ – количество элементов в массиве. При *дихотомическом поиске* требуется не более $m$ сравнений, если $n$-$m$-я степень 2, если $n$ не является степенью 2, то $n < k = 2^m$.

Массив делится пополам $S:=(L+R)/2+1$ и определяется, в какой части массива находится нужный элемент $X$. Т.к. массив упорядочен, то если $a[S] < X$, то искомый элемент находится в правой части массива, иначе – находится в левой части. Выбранную часть массива снова надо разделить пополам и т.д., до тех пор, пока границы отрезка $L$ и $R$ не станут равны [11].

**Пример 46**

```
int b;
```

```
cout<<«\nB=?»;cin>>b;
int l=0,r=n-1,s;
do
{
    s=(l+r)/2;//средний элемент
    if(a[s]<b)l=s+1;//перенести левую границу
       else r=s;//перенести правую границу
}while(l!=r);
    if(a[l]==b)return l;
    else return -1;
```

### 4.6.2. Указатели

**Понятие указателя**

*Указатели* – специальные объекты в программах на C++, предназначенные для хранения адресов памяти.

Когда компилятор обрабатывает оператор определения переменной, например, *int i*=10; то в памяти выделяется участок памяти в соответствии с типом переменной (*int*=> 4байта) и записывает в этот участок указанное значение. Все обращения к этой переменной компилятор заменит на адрес области памяти, в которой хранится эта переменная.

Программист может определить собственные переменные для хранения адресов областей памяти. Такие переменные называются указателями. Указатель не является самостоятельным типом, он всегда связан с каким-то другим типом.

Указатели делятся на две *категории*:

1) указатели на объекты,

2) указатели на функции.

Рассмотрим *указатели на объекты*, которые хранят адрес области памяти, содержащей данные определенного типа.

В простейшем случае объявление указателя имеет вид:

**тип *имя;**

Тип может быть любым, кроме ссылки.

**Пример 47**

```
int *i;
double *f, *ff;
char *c;
```

Размер указателя зависит от модели памяти. Можно определить указатель на указатель: `int**a;`

Указатель может быть константой или переменной, а также указывать на константу или переменную.

**Пример 48**

1. `int i;  //целая переменная`
`const int ci=1;  //целая константа`
`int *pi;  //указатель на целую переменную`
`const int *pci;//указатель на целую константу`

Указатель можно сразу проинициализировать:

`int *pi=&i;  //указатель на целую переменную`
`const int *pci=&ci;//указатель на целую константу`

2. `int*const cpi=&i; //указатель-константа на целую переменную`
`const int* const cpc=&ci; //указатель-константа на целую константу`

Если модификатор const относится к указателю (т.е. находится между именем указателя и *), то он запрещает изменение указателя, а если он находится слева от типа (т.е. слева от *), то он запрещает изменение значения, на которое указывает указатель.

Для *инициализации указателя* существуют следующие способы (рис. 20):

1. Присваивание адреса существующего объекта:
   1) с помощью операции получения адреса
      `int a=5;`
      `int *p=&a; или int p(&a);`
   2) с помощью проинициализированного указателя
      `int *r=p;`

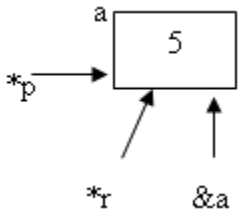



Рис. 20. Инициализация указателя

   3) адрес присваивается в явном виде
      `char*cp=(char*)0x B800 0000;`

где 0x B800 0000 – шестнадцатеричная константа, (char*) – операция приведения типа.

   4) присваивание пустого значения:
      `int*N=NULL;`

```
int *R=0;
```

**Динамические переменные**

Все переменные, объявленные в программе, размещаются в одной непрерывной области памяти, которую называют *сегментом данных* (64 Кб). Такие переменные не меняют своего размера в ходе выполнения программы и называются статическими. Размера сегмента данных может быть недостаточно для размещения больших массивов информации. Выходом из этой ситуации является использование динамической памяти. *Динамическая память* – это память, выделяемая программе для ее работы за вычетом сегмента данных, стека, в котором размещаются локальные переменные подпрограмм и собственно тела программы.

Для работы с динамической памятью используют указатели. С их помощью осуществляется доступ к участкам динамической памяти, которые называются динамическими переменными. Динамические переменные создаются с помощью специальных функций и операций. Они существуют либо до конца работы программ, либо до тех пор, пока не будут уничтожены с помощью специальных функций или операций.

Для создания динамических переменных используют операцию new, определенную в C++:

**указатель = new имя_типа[инициализатор];**

где *инициализатор* – выражение в круглых скобках.

Операция *new* позволяет выделить и сделать доступным участок динамической памяти, который соответствует заданному типу данных. Если задан инициализатор, то в этот участок будет занесено значение, указанное в инициализаторе:

```
int*x=new int(5);
```

Для удаления динамических переменных используется операция *delete*, определенная в C++:

```
delete указатель;
```

где указатель содержит адрес участка памяти, ранее выделенный с помощью операции *new*:

```
delete x;
```

В языке С определены библиотечные функции для работы с динамической памятью, они находятся в библиотеке <*stdlib.h*>:

1) **void*malloc** (**unsigned s**) – возвращает указатель на начало области динамической памяти длиной s байт, при неудачном завершении возвращает NULL;

2) **void*calloc** (**unsigned n**, **unsigned m**) – возвращает указатель на начало области динамической для размещения n элементов длиной *m* байт каждый, при неудачном завершении возвращает NULL;

3) **void*realloc (void *p, unsigned s)** – изменяет размер блока ранее выделенной динамической до размера s байт, *p* – адрес начала изменяемого блока, при неудачном завершении возвращает NULL;

4) **void *free (void *p)** – освобождает ранее выделенный участок динамической памяти, *p* – адрес начала участка.

**Пример 49**

```
int  *u=(int*)malloc(sizeof(int));  // в функцию
```

передается количество требуемой памяти в байтах, т.к. функция возвращает значение типа void*, то его необходимо преобразовать к типу указателя (int*).

```
free(u); //освобождение выделенной памяти
```

**Операции с указателями**

С указателями можно выполнять следующие операции:

1) разыменование (*);
2) присваивание;
3) арифметические операции (сложение с константой, вычитание, инкремент ++, декремент --);
4) сравнение;
5) приведение типов.

Операция *разыменования* предназначена для получения значения переменной или константы, адрес которой хранится в указателе. Если указатель указывает на переменную, то это значение можно изменять, также используя операцию разыменования.

**Пример 50**

```
int a; //переменная типа int
int*pa=new int; //указатель и выделение памяти под динамическую переменную
*pa=10;//присвоили значение динамической переменной, на которую указывает указатель
a=*pa; //присвоили значение переменной a
```

*Присваивать значение* указателям-константам **запрещено**.

*Приведение типов.* На одну и ту же область памяти могут ссылаться указатели разного типа. Если применить к ним операцию разыменования, то получатся разные результаты.

**Пример 51**

```
int a=123;
int*pi=&a;
char*pc=(char*)&a;
float *pf=(float*)&a;
```

```
printf(«\n%x\t%i»,pi,*pi);
printf(«\n%x\t%c»,pc,*pc);
printf(«\n%x\t%f»,pf,*pf);
```

При выполнении программы, представленной в примере 51, получатся следующие результаты:

66fd9c 123
66fd9c {
66fd9c 0.000000

Т.е. адрес у трех указателей один и тот же, но при разыменовании получаются разные значения в зависимости от типа указателя.

В примере при инициализации указателя была использована операция приведения типов. При использовании в выражении указателей разных типов, явное преобразование требуется для всех типов, кроме void*. Указатель может неявно преобразовываться в значения типа bool, при этом ненулевой указатель преобразуется в true, а нулевой в false.

*Арифметические операции* применимы только к указателям одного типа.

1. Инкремент увеличивает значение указателя на величину sizeof(тип).

**Пример 52**

```
char *pc;
int *pi;
float *pf;
...
pc++;//значение увеличится на 1
pi++;//значение увеличится на 4
pf++;//значение увеличится на 4
```

2. Декремент уменьшает значение указателя на величину sizeof(тип).

3. Разность двух указателей – это разность их значений, деленная на размер типа в байтах.

**Пример 53**

```
int a=123,b=456,c=789;
int*pi1=&a;
int *pi2=&b;
int*pi3=&c;
printf(«\n%x»,pi1-pi2);
printf(«\n%x»,pi1-pi3);
```

Результат

1
2

Суммирование двух указателей не допускается. Можно суммировать указатель и константу.

**Пример 54**

```
pi3=pi3+2;
pi2=pi2+1;
printf(«\n%x\t%d»,pi1,*pi1);
printf(«\n%x\t%d»,pi2,*pi2);
printf(«\n%x\t%d»,pi3,*pi3);
```

Результат выполнения программы:

66fd9c  123
66fd9c  123
66fd9c  123

При записи выражений с указателями требуется обращать внимание на приоритеты операций.

### 4.6.3. Ссылки

**Понятие ссылки**

*Ссылка* – это синоним имени объекта, указанного при инициализации ссылки. Формат объявления ссылки

**тип & имя =имя_объекта;**

**Пример 55.**

```
int  x;// определение переменной
int&  sx=x;// определение ссылки на переменную x
const char& CR='\n';//определение ссылки на
                        //константу
```

**Правила работы со ссылками**

1. Переменная ссылка должна явно инициализироваться при ее описании, если она не является параметром функции, не описана как extern или не ссылается на поле класса.

2. После инициализации ссылке не может быть присвоено другое значение.

3. Не существует указателей на ссылки, массивов ссылок и ссылок на ссылки.

4. Операция над ссылкой приводит к изменению величины, на которую она ссылается.

Ссылка не занимает дополнительного пространства в памяти, она является просто другим именем объекта.

**Пример 56**

```
#include <iostream.h>
void main()
{
int I=123;
int &si=I;
cout<<"\ni="<<I<<" si="<<si;
I=456;
cout<<"\ni="<<I<<" si="<<si;
I=0; cout<<"\ni="<<I<<" si="<<si;
}
```

Результат работы программы:
I=123 si=123
I=456 si=456
I=0 si=0

### 4.6.4. Указатели и массивы

**Одномерные массивы и указатели**

При определении массива ему выделяется память. После этого имя массива воспринимается как константный указатель того типа, к которому относятся элементы массива. Исключением является использование операции **sizeof (имя_массива)** и операции **&имя_массива**.

**Пример 57**

```
int a[100];
int k=sizeof(a);/* результатом будет 4*100=400 (байтов)*/
int n=sizeof(a)/sizeof(a[0]);/*количество элементов массива */
```

Результатом операции & является адрес нулевого элемента массива: имя_массива==&имя_массива=&имя_массива [0]

Имя массива является указателем-константой, значением которой служит адрес первого элемента массива, следовательно, к нему применимы все правила адресной арифметики, связанной с указателями. Запись **имя_массива[индекс]** – это выражение с двумя операндами: имя массива и индекс. *Имя_массива* – это указатель константа, а *индекс*

определяет смещение от начала массива. Используя указатели, обращение по индексу можно записать следующим образом:

***(имя_массива+индекс)**.

**Пример 58**

```
for (int i=0;i<n;i++)//печать массива
cout<<*(a+i)<< )<" "; /* к имени адресу массива добавляется константа i  и полученное значение разыменовывается */
```

Так как имя массива является константным указателем, то его невозможно изменить, следовательно, запись *(a++) будет ошибочной, а *(a+1) – нет.

Указатели можно использовать и при определении массивов:

```
int  a[100]={1,2,3,4,5,6,7,8,9,10};
int * na=a;//поставили указатель на уже определенный массив
int b=new int[100];//выделили в динамической памяти место под массив из 100 элементов
```

**Многомерные массивы и указатели**

Многомерный массив это массив, элементами которого служат массивы. Например, массив с описанием int a[4][5] – это массив из 4 указателей типа int*, которые содержат адреса одномерных массивов из 5 целых элементов (рис. 21).

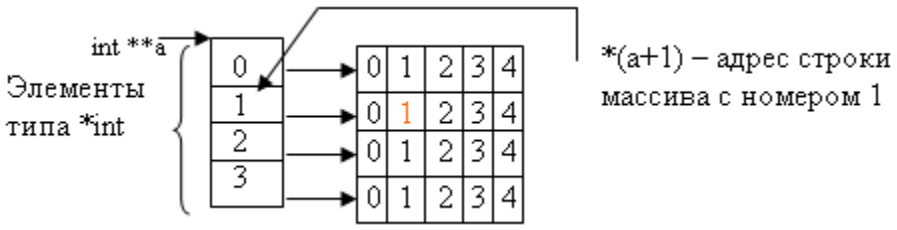



Рис. 21. Доступ к элементам одномерных массивов

Инициализация многомерных массивов выполняется аналогично одномерным массивам.

**Пример 59**

```
int a[3][4] = {{11,22,33,44},{55,66,77,88},
{99,110,120,130}};
//проинициализированы все элементы массива
int b[3][4] = {{1},{2},{3}};//проинициализированы первые элементы
// каждой строки
int c[3][2]={1,2,3,4,5,6};//проинициализированы
```

```
//все элементы массива
```

Доступ к элементам многомерных массивов возможен и с помощью индексированных переменных и с помощью указателей:

`a[1][1]` – доступ с помощью индексированных переменных,

`*(*(a+1)+1)` – доступ к этому же элементу с помощью указателей (рис. 21).

### 4.6.5. Динамические массивы

Операция *new* при использовании с массивами имеет следующий формат:

**new тип_массива**

Такая операция выделяет для размещения массива участок динамической памяти соответствующего размера, но не позволяет инициализировать элементы массива. Операция new возвращает указатель, значением которого служит адрес первого элемента массива. При выделении динамической памяти размеры массива должны быть полностью определены.

**Пример 60.** Выделение динамической памяти:

1. `int *a=new int[100];` /*выделение динамической памяти размером 100*sizeof(int) байтов*/

`double *b=new double[12];` /* выделение динамической памяти размером 10*sizeof(double) байтов */

2. `long(*la)[4];` /*указатель на массив из 4 элементов типа long*/

`la=new[2][4];` /*выделение динамической памяти размером 2*4*sizeof(long) байтов*/

3. `int**matr=(int**)new int[5][12];` /*еще один способ выделения памяти под двумерный массив*/

4. `int **matr;`

`matr=new int*[4];` /*выделяем память под массив указателей int* их n элементов*/

`for(int I=0;I<4;I++)matr[I]=new int[6];`/*выделяем память под строки массива*/

Указатель на динамический массив затем используется при освобождении памяти с помощью операции delete.

**Пример 61.** Освобождение динамической памяти.

```
delete[] a; //освобождает память, выделенную под массив, если a адресует его начало
```

```
    delete[]b;
    delete[] la;
    for(I=0;I<4;I++)delete    []    matr[I];//удаляем
строки
    delete [] matr;//удаляем массив указателей
```

**Пример 62.** Удалить из матрицы строку с номером *K*.

```
    #include <iostream.h>
    #include <stdlib.h>
    #include <conio.h>
    void main()
    {
     int n,m;//размерность матрицы
     int i,j;
     cout<<«\nEnter n»;
     cin>>n;//строки
     cout<<«\nEnter m»;
     cin>>m;//столбцы
     //выделение памяти
     int **matr=new int* [n];/* массив указателей на
строки*/
     for(i=0;i<n;i++)
         matr[i]=new  int  [m];/*память  под  элементы
матрицы*/
     //заполнение матрицы
     for(i=0;i<n;i++)
     for(j=0;j<m;j++)
         matr[i][j]=rand()%10;//заполнение матрицы
     //печать сформированной матрицы
     for(i=0;i<n;i++)
     {
         for(j=0;j<m;j++)
             cout<<matr[i][j]<<« «;
         cout<<«\n»;
     }
     //удаление строки с номером к
     int k;
     cout<<«\nEnter k»;
     cin>>k;
     int**temp=new   int*[n-1];/*формирование   новой
матрицы*/
     for(i=0;i<n;i++)
```

```
        temp[i]=new int[m];
    //заполнение новой матрицы
    int t;
    for(i=0,t=0;i<n;i++)
        if(i!=k)
        {
            for(j=0;j<m;j++)
                temp[t][j]=matr[i][j];
            t++;
        }
                //удаление старой матрицы
    for(i=0;i<n;i++)
        delete matr[i];
    delete[]matr;
   n--;
    //печать новой матрицы
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
            cout<<temp[i][j]<<« «;
        cout<<«\n»;
    }
   getch();
   }
```

### 4.7. Символьная информация и строки

Для символьных данных в С++ введен тип *char*. Для представления символьной информации используются символы, символьные переменные и текстовые константы.

**Пример 63**

```
const char c='c'; /*символ – занимает один байт, его значение не меняется*/
char a,b;/*символьные переменные, занимают по одному байту, значения меняются*/
const char *s="Пример строки\n" ;//текстовая константа
```

*Строка* в С++ – это массив символов, заканчивающийся нуль-символом – ‘\0’ (нуль-терминатором). По положению нуль-терминатора определяется фактическая длина строки (рис. 22). Количество элементов в таком массиве на 1 больше, чем изображение строки.


“A” ‘A’

строка символ

(2байта) (1байт)

Рис. 22. Представление строки и символа

Присвоить значение строке с помощью оператора присваивания нельзя. Поместить строку в массив можно либо при вводе, либо с помощью инициализации.

**Пример 64**

```
void main()
{
 char s1[12]="string1";
 int k=sizeof(s1);
 cout<<s1<<"\t"<<k<<endl;
 char s2[]="string2";
 k=sizeof(s2);
 cout<<s2<<"\t"<<k<<endl;
 char s3[]={'s','t','r','i','n','g','3'}
 k=sizeof(s3);
 cout<<s3<<"\t"<<k<<endl;
 char  *s4="string4";//указатель  на  строку,  ее
нельзя изменить
 k=sizeof(s4);
 cout<<s4<<"\t"<<k<<endl;
}
```

Результаты:

string1 10 – выделено 10 байтов, в том числе под \0

string2 8 – выделено 8 байтов (7+1байт под \0)

string3 8 – выделено 8 байтов (7+1байт под \0)

string4 4 – размер указателя

**Пример 65.**

```
char *s="String5"; // выделяется 8 байтов для строки
char*ss; // описан указатель
ss="String6"; //память не выделяется, поэтому программа мо-
жет закончиться аварийно.
char *sss=new char[12]; //выделяем динамическую память
strcpy(sss,"String7"); //копируем строку в память
```

*Для ввода и вывода символьных данных* в библиотеке языка С определены следующие *функции:*

**int getchar(void)** – осуществляет вод одного символа их входного потока, при этом она возвращает один байт информации (символ) в виде значения типа *int*. Это сделано для распознавания ситуации, когда при чтении будет достигнут конец файла.

**int putchar (int c)** – помещает в стандартный выходной поток символ *c*.

**char* gets(char*s)** – считывает строку *s* из стандартного потока до появления символа '\n', сам символ '\n' в строку не заносится.

**int puts(const char* s)** записывает строку в стандартный поток, добавляя в конец строки символ '\n', в случае удачного завершения возвращает значение больше или равное 0 и отрицательное значение (EOF=-1) в случае ошибки.

**Пример 66**

1.
```
char s[20];
 cin>>s; //ввод строки из стандартного потока
 cout<<s; //вывод строки в стандартный поток
```

Результат работы программы:

При вводе строки "123 456 789", чтение байтов осуществляется до первого пробела, т.е. в строку s занесется только первое слово строки "123/0", следовательно, выведется: 123.

2.
```
char s[20];
gets(s); //ввод строки из стандартного потока
puts(s); //вывод строки в стандартный поток
```

Результат работы программы:

При вводе строки "123 456 789", чтение байтов осуществляется до символа '\n', т.е. в *s* занесется строка"123 456 789\n\0", при выводе строки функция puts возвращает еще один символ '\n', следовательно, будет выведена строка "123 456 789\n\n".

3.
```
char s[20];
scanf("%s",s);/*ввод строки из стандартного потока*/
printf("%s",s); /*вывод строки в стандартный поток*/
```

Результат работы программы:

При вводе строки "123 456 789", чтение байтов осуществляется до первого пробела, т.е. в строку s занесется только первое слово строки "123/0", следовательно, выведется: 123. Т.к. *s* – имя массива, т.е. адрес его первого элемента, операция & в функции *scanf* не используется.

Для работы со строками существуют специальные библиотечные функции, которые содержатся в заголовочном файле **string.h**. Некоторые из этих функций представлены в табл. 18.

Таблица 18

Функции работы со строками

| **Прототип функции** | **Краткое описание** | **Примечание** |
|---|---|---|
| unsigned **strlen** (const char*s); | Вычисляет длину строки s. | |
| int **strcmp** (const char*s1, const char *s2); | Сравнивает строки s1 и s2. | Если s1<s2, то результат отрицательный, если s1==s2, то результат равен 0, если s2>s1 – результат положительный. |
| int **strcnmp** (const char*s1, const char *s2); | Сравнивает первые n символов строк s1 и s2. | Если s1<s2, то результат отрицательный, если s1==s2, то результат равен 0, если s2>s1 – результат положительный. |
| char***strcpy**(char*s1, const char*s2); | Копирует символы строки s1 в строку s2. | |
| char***strncpy**(char*s1, const char*s2, int n); | Копирует n символов строки s1 в строку s2. | Конец строки отбрасывается или дополняется пробелами. |
| char***strcat** (char*s1, const char*s2); | Приписывает строку s2 к строке s1 | |
| char***strncat** (char*s1, const char*s2); | Приписывает первые n символов строки s2 к строке s1 | |
| char***strdup** (const char*s); | Выделяет память и переносит в нее копию строки s | При выделении памяти используются функции |

**Пример 67.** Дана строка символов, состоящая из слов, слова разделены между собой пробелами. Удалить из строки все слова, начинающиеся с цифры.

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
void main()
{
char s[250], //исходная строка
w[25], //слово
mas[12][25];//массив слов
puts("\nвведите строку");
gets(s);
```

```
    int k=0,t=0,i,len,j;
    len=strlen(s);
    while(t<len)
    {//формируем слово до пробела
    for(j=0,i=t;s[i]!=’ ‘;i++,j++)w[j]=s[i];
    w[j]=’/0’;//формируем конец строки
    strcpy(mas[k],w);//копируем слово в массив
    k++;//увеличиваем счетчик слов
    t=i+1;/*переходим к следующему слову в исходной
строке*/
    }
    strcpy(s,””);//очищаем исходную строку
    for(t=0;t<k;t++)
    if(mas[t][0]<’0’&&mas[t][0]>’9’)//если     первый
символ не цифра
    {
    strcat(s,mas[t]);//копируем в строку слово
    strcat(s,” “);//копируем в строку пробел
    }
    puts(s);//выводим результат
    getch();
    }
```

**Пример 68.** Сформировать динамический массив строк. Удалить из него строку с заданным номером.

```
    #include <iostream.h>
    #include <string.h>
    #include <conio.h>
    void main()
    {
     int n;
     cout<<«\nN=?»;cin>>n;
     char s[100];
     char**matr=new char*[n];
     for(int i=0;i<n;i++)
     {
          cout<<«\nS=?»;
          cin>>s;
          matr[i]=new char[strlen(s)];
          strcpy(matr[i],s);
     }
     for(i=0;i<n;i++)
```

```
    {
        cout<<matr[i];
        cout<<«\n»;
    }
    int k;
    cout<<«\nK=?»;
    cin>>k;
    if(k>=n){cout<<«There is not such string\n»;re-
turn;}
    char **temp=new char*[n-1];
    int j=0;

    for(i=0;i<n;i++)
        if(i!=k)
        {
            temp[j]=new char[strlen(matr[i])];
            strcpy(temp[j],matr[i]);
            j++;
        }

        n--;
    for(i=0;i<n;i++)
    {
        cout<<temp[i];
        cout<<«\n»;
    }
   getch();
   }
```

### 4.8. Функции в C++

С увеличением объема программы становится невозможно удерживать в памяти все детали. Чтобы уменьшить сложность программы, ее разбивают на части. В C++ задача может быть разделена на более простые подзадачи с помощью функций. Разделение задачи на функции также позволяет избежать избыточности кода, т.к. функцию записывают один раз, а вызывают многократно. Программу, которая содержит функции, легче отлаживать.

Часто используемые функции можно помещать в библиотеки. Таким образом, создаются более простые в отладке и сопровождении программы.

### 4.8.1. Объявление и определение функций

*Функция* – это именованная последовательность описаний и операторов, выполняющая законченное действие, например, формирование массива, печать массива и т.д. (рис. 23).

Функция, во-первых, является одним из производных типов C++, а, во-вторых, минимальным исполняемым модулем программы.

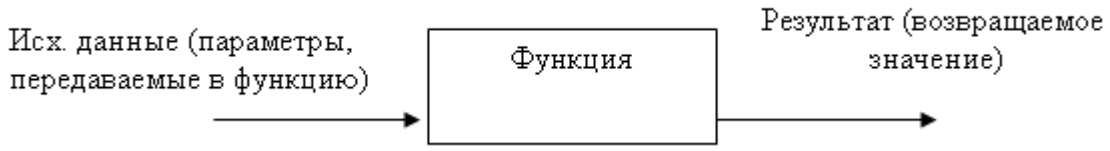



Рис. 23. Функция

Любая функция должна быть *объявлена* и *определена*.

*Объявление* функции (*прототип*, *заголовок*) задает имя функции, тип возвращаемого значения и список передаваемых параметров.

*Определение* функции содержит, кроме объявления, тело функции, которое представляет собой последовательность описаний и операторов. Общая форма определения функции:

**тип имя_функции([список_формальных_параметров])**
**{ тело_функции}**

*Тело_функции* – это блок или составной оператор. Внутри функции нельзя определить другую функцию.

В теле функции должен быть оператор, который возвращает полученное значение функции в точку вызова. Он может иметь формы:

1) return выражение;

2) return.

Форма 1) используется для возврата результата, поэтому выражение должно иметь тот же тип, что и тип функции в определении. Форма 2) используется, если функция не возвращает значения, т.е. имеет тип *void*. Программист может не использовать этот оператор в теле функции явно, компилятор добавит его автоматически в конец функции перед знаком «}».

Тип возвращаемого значения может быть любым, кроме массива и функции, но может быть указателем на массив или функцию.

*Список формальных параметров* – это те величины, которые требуется передать в функцию. Элементы списка разделяются запятыми. Для каждого параметра указывается тип и имя. В объявлении имена можно не указывать.

Для того, чтобы выполнялись операторы, записанные в теле функции, функцию необходимо вызвать. При вызове указываются: имя функции и фактические параметры. Фактические параметры заменяют формальные параметры при выполнении операторов тела функции. Фактические и формальные параметры должны совпадать по количеству и типу.

**Объявление** функции **должно находиться в тексте раньше вызова** функции, чтобы компилятор мог осуществить проверку правильности вызова. Если функция имеет тип не void, то ее вызов может быть операндом выражения.

**Пример 69.** Заданы координаты сторон треугольника. Если такой треугольник существует, то найти его площадь.

1. *Математическая модель*:

1)`l=sqrt(pow(x1-x2,2)+pow(y1-y2,2));` /*длина стороны треугольника*/

2)`p=(a+b+c)/2;` `s=sqrt(p*(p-a)*(p-b)*(p-c));` //формула Герона

3)проверка существования треугольника (a+b>c&&a+c>b&&c+b>a)

2. *Алгоритм:*

1) Ввести координаты сторон треугольника (x1,y1),(x2,y2), (x3,y3);
2) Вычислить длины сторон ab, bc, ca;
3) Проверить существует ли треугольник с такими сторонами. Если да, то вычислить площадь и вывести результат.
4) Если нет, то вывести сообщение.
5) Если все координаты равны 0, то конец, иначе возврат на п. 1.

3. *Программа*:

```
#include <iostream.h>
#include <math.h>
#include <conio.h>
double line(double x1,double y1,double x2,double y2)
{
//функция возвращает длину отрезка, заданного координатами x1,y1 и x2,y2
return sqrt(pow(x1-x2,2)+pow(y1-y2,2));
}
double square(double a, double b, double c)
{
```

```
   //функция возвращает площадь треугольника, за-
данного длинами сторон a,b,c
   double s, p=(a+b+c)/2;
   return      s=sqrt(p*(p-a)*(p-b)*(p-c));//формула
Герона
   }
   bool  triangle(double a, double b, double c)
   {
   //возвращает true, если треугольник существует
   if(a+b>c&&a+c>b&&c+b>a) return true;
   else return false;
   }
   void main()
   {
   double x1=1,y1,x2,y2,x3,y3;
   double point1_2,point1_3,point2_3;
   do
   {
   cout<<«\nEnter koordinats of triangle:»;
   cin>>x1>>y1>>x2>>y2>>x3>>y3;
   point1_2=line(x1,y1,x2,y2);
   point1_3=line(x1,y1,x3,y3);
   point2_3=line(x2,y2,x3,y3);
   if(triangle(point1_2,point1_3,point2_3)==true)
   cout<<«S=«<<square(point1_2,point2_3,point1_3)<<
«\n»;
   else cout<<«\nTriagle doesnt exist»;
   }
   while(!
(x1==0&&y1==0&&x2==0&&y2==0&&x3==0&&y3==0));
   getch();
   }
```

### 4.8.2. Прототип функции

Для того, чтобы к функции можно было обратиться, в том же файле должно находиться определение или описание функции (прототип).

```
   double line(double x1,double y1,double x2,double y2);
   double square(double a, double b, double c);
   bool triangle(double a, double b, double c);
   double line(double ,double ,double ,double);
   double square(double , double , double );
```

```
bool triangle(double , double , double );
```

Это прототипы функций, описанных выше.

При наличии прототипов вызываемые функции не обязаны размещаться в одном файле с вызывающей функцией, а могут оформляться в виде отдельных модулей и храниться в откомпилированном виде в библиотеке объектных модулей. Это относится и к функциям из стандартных модулей. В этом случае определения библиотечных функций уже оттранслированные и оформленные в виде объектных модулей, находятся в библиотеке компилятора, а описания функций необходимо включать в программу дополнительно. Это делают с помощью препроцессорных команд **include< имя файла>**.

*Имя_файла* – определяет заголовочный файл, содержащий прототипы группы стандартных для данного компилятора функций. Например, почти во всех программах мы использовали команду **#include <iostream.h>** для описания объектов потокового ввода-вывода и соответствующие им операции.

При разработке своих программ, состоящих из большого количества функций, и, размещенных в разных модулях, прототипы функций и описания внешних объектов (констант, переменных, массивов) помещают в отдельный файл, который включают в начало каждого из модулей программы с помощью директивы **include”имя_файла”**.

### 4.8.3. Параметры функции

Основным способом обмена информацией между вызываемой и вызывающей функциями является механизм параметров. Существует два способа передачи параметров в функцию: по адресу и по значению.

При передаче *по значению* выполняются следующие действия:

1) вычисляются значения выражений, стоящие на месте фактических параметров;

2) в стеке выделяется память под формальные параметры функции;

3) каждому фактическому параметру присваивается значение формального параметра, при этом проверяются соответствия типов и при необходимости выполняются их преобразования.

**Пример 70**

```
double square(double a, double b, double c)
{
//функция возвращает площадь треугольника, заданного длинами сторон a,b,c
double s, p=(a+b+c)/2;
```

```
    return      s=sqrt(p*(p-a)*(p-b)*(p-c));//формула
Герона
    }
    ...
     double s1=square(2.5,2,1);
     double a=2.5,b=2,c=1;
    double s2=square(a,b,c);
     double x1=1,y1=1,x2=3,y2=2,x3=3,y3=1;
    double        s3=square(sqrt(pow(x1-x2,2)+pow(y1-
y2,2)),//расстояние между 1 и 2
    sqrt(pow(x1-x3,2)+pow(y1-y3,2)),     //расстояние
между 1 и 3
    sqrt(pow(x3-x2,2)+pow(y3-y2,2)));    //расстояние
между 2 и 3
    ...
```

*p* и *s* – локальные переменные.

Таким образом в стек заносятся копии фактических параметров и операторы функции работают с этими копиями. Доступа к самим фактическим параметрам у функции нет, следовательно, нет возможности их изменить.

При передаче *по адресу* в стек заносятся копии адресов параметров, следовательно, у функции появляется доступ к ячейке памяти, в которой находится фактический параметр и она может его изменить.

**Пример 71**

```
void Change(int a,int b) //передача по значению
{int r=a;a=b;b=r;}

int x=1,y=5;
Change(x,y);
   A     1     5
   B     5     1
   r           1
cout<<”x=”<<x<<”y=”<<y;
выведется: x=1y=5
void Change(int *a,int *b)//передача по адресу
{int r=*a;*a=*b;*b=r;}
int x=1,y=5;
Change(&x,&y);
   A     &x    5
   B     &y    1
```

```
   r              1
cout<<"x="<<x<<"y="<<y;
```

Результат работы программы: x=5y=1

Для передачи по адресу также могут использоваться ссылки. При передаче по ссылке в функцию передается адрес указанного при вызове параметра, а внутри функции все обращения к параметру неявно разыменовываются.

**Пример 72**

```
void Change(int &a,int &b)
{int r=a;a=b;b=r;}
int x=1,y=5;
Change(x,y);
  A      &x      5
  B      &y      1
  r              1
cout<<"x="<<x<<"y="<<y;
```

Результат работы программы: x=5y=1

Использование ссылок вместо указателей улучшает читаемость программы, т.к. не надо применять операцию разыменовывания. Использование ссылок вместо передачи по значению также более эффективно, т.к. не требует копирования параметров. Если требуется запретить изменение параметра внутри функции, используется модификатор const. Рекомендуется ставить const перед всеми параметрами, изменение которых в функции не предусмотрено (по заголовку будет понятно, какие параметры в ней будут изменяться, а какие нет).

### 4.8.4. Локальные и глобальные переменные

Переменные, которые используются внутри данной функции, называются локальными. Память для них выделяется в стеке, поэтому после окончания работы функции они удаляются из памяти. Нельзя возвращать указатель на локальную переменную, т.к. память, выделенная такой переменной, будет освобождаться.

**Пример 73**

```
int*f()
{
int a;
. . . .
return&a; // НЕВЕРНО
```

```
}
```

*Глобальные переменные* – это переменные, описанные вне функций. Они видны во всех функциях, где нет локальных переменных с такими именами.

**Пример 74**

```
int a,b; //глобальные переменные
void change()
{
int r; //локальная переменная
r=a;a=b;b=r;
}
void main()
{
cin>>a,b;
change();
cout<<”a=”<<a<<”b=”<<b;
}
```

Глобальные переменные также можно использовать для передачи данных между функциями, но этого не рекомендуется делать, т.к. это затрудняет отладку программы и препятствует помещению функций в библиотеки. Нужно стремиться к тому, чтобы функции были максимально независимы, а их интерфейс полностью определялся прототипом функции.

### 4.8.5. Функции и массивы

#### 4.8.5.1. Передача одномерных массивов как параметров функции

При использовании массива как параметра функции, в функцию передается указатель на его первый элемент, т.е. массив всегда передается по адресу. При этом теряется информация о количестве элементов в массиве, поэтому размерность массива следует передавать как отдельный параметр. Так как в функцию передается указатель на начало массива (передача по адресу), то массив может быть изменен за счет операторов тела функции.

**Пример 75.** Удалить из массива все четные элементы

```
#include <iostream.h>
#include <stdlib.h>
int form(int a[100])
{
```

```
int n;
 cout<<«\nEnter n»;
 cin>>n;
 for(int i=0;i<n;i++)
     a[i]=rand()%100;
 return n;
}
void print(int a[100],int n)
{
 for(int i=0;i<n;i++)
     cout<<a[i]<<« «;
 cout<<«\n»;
}

void Dell(int a[100],int&n)
{
 int j=0,i,b[100];
 for(i=0;i<n;i++)
     if(a[i]%2!=0)
     {
         b[j]=a[i];j++;
     }
     n=j;
     for(i=0;i<n;i++)a[i]=b[i];
}
void main()
{
 int a[100];
 int n;
 n=form(a);
 print(a,n);
 Dell(a,n);
 print(a,n);
}
```

**Пример 76.** Удалить из массива все элементы, совпадающие с первым элементом, используя динамическое выделение памяти.

```
#include <iostream.h>
#include <stdlib.h>
int* form(int&n)
{
 cout<<«\nEnter n»;
```

```
     cin>>n;
     int*a=new  int[n];/*указатель  на  динамическую
область памяти*/
     for(int i=0;i<n;i++)
          a[i]=rand()%100;
     return a;
    }
    void print(int*a,int n)
    {
     for(int i=0;i<n;i++)
          cout<<a[i]<<« «;
     cout<<«\n»;
    }

    int*Dell(int *a,int&n)
    {
     int k,j,i;
     for(k=0,i=0;i<n;i++)
          if(a[i]!=a[0])k++;
          int*b;
          b=new int [k];
          for(j=0,i=0;i<n;i++)
          if(a[i]!=a[0])
          {
               b[j]=a[i];j++;
          }
          n=k;
          return b;
    }
    void main()
    {
     int *a;
     int n;
     a=form(n);
     print(a,n);
     a=Dell(a,n);
     print(a,n);
    }
```

### 4.8.5.2. Передача строк в качестве параметров функций

Строки при передаче в функции могут передаваться как одномерные массивы типа char или как указатели типа *char**. В отличие от обычных массивов в функции не указывается длина строки, т.к. в конце строки есть признак конца строки /0.

**Пример 77.** Функция поиска заданного символа в строке

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

int find (char *s,char c)
{
for (int I=0;I<strlen(s);I++)
if(s[I]==c) return I;
return -1;
}
/*С помощью этой функции подсчитаем количество
гласных букв в строке.*/
void main()
{
char s[255];
cin>>s;
char*gl=”aouiey”;
for(int I=0,k=0;I<strlen(gl);I++)
if(find(s,gl[I])>0)k++;
cout<<k;
}
```

### 4.8.5.3. Передача многомерных массивов в функцию

При передаче многомерных массивов в функцию все размерности должны передаваться в качестве параметров. По определению многомерные массивы в С и С++ не существуют. Если мы описываем массив с несколькими индексами, например, массив int mas[3][4], то это означает, что мы описали одномерный массив mas, элементами которого являются указатели на одномерные массивы int[4].

**Пример 78.** Транспонирование квадратной матрицы

Если определить заголовок функции:

`void transp(int a[][],int n){…..}` – то получится, что мы хотим передать в функцию массив с неизвестными размерами.

По определению массив должен быть одномерным, и его элементы должны иметь одинаковую длину. При передаче массива ничего не сказано и о размере элементов, поэтому компилятор выдаст ошибку.

Самый простой вариант решения этой проблемы определить функцию следующим образом:

`void transp(int a[][4],int n)`, тогда размер каждой строки будет 4, а размер массива указателей будет вычисляться.

```
#include<iostream.h>
const int N=4;//глобальная переменная
void transp(int a[][N],int n)
{
int r;
for(int I=0;I<n;I++)
for(int j=0;j<n;j++)
if(I<j)
{
r[a[I][j];a[I][j]=a[j][I];a[j][I]=r;
}
}
void main()
{
int mas[N][N];
for(int I=0;I<N;I++)
for(int j=0;j<Nlj++)
cin>>mas[I][j];
for(I=0;I<N;I++)
{
for(j=0;j<N;j++)
cout<<mas[I][j]
cout<<”\n”;
}
transp(N,mas);
for(I=0;I<N;I++)
{
for(j=0;j<N;j++)
cout<<mas[I][j]
cout<<”\n”;
}
}
```

### 4.8.6. Функции с начальными (умалчиваемыми) значениями параметров

В определении функции может содержаться начальное (умалчиваемое) значение параметра. Это значение используется, если при вызове функции соответствующий параметр опущен. Все параметры, описанные справа от такого параметра также должны быть умалчиваемыми.

**Пример 79**

```
void print(char*name="Номер дома: ",int value=1)
{cout<<"\n"<<name<<value;}
```

Вызовы:

1. print();

Вывод: Номер дома: 1

2. print("Номер квартиры",15);

Вывод: Номер квартиры: 15

3. print(,15); - ошибка, т.к. параметры можно пускать только с конца

Поэтому функцию лучше переписать так:

```
void print(int value=1, char*name="Номер дома:
")
{cout<<"\n"<<name<<value;}
```

Вызовы:

1. print();

Вывод: Номер дома: 1

2. print(15);

Вывод: Номер дома: 15

3. print(6, "Размерность пространства");

Вывод: Размерность пространства: 6

### 4.8.7. Подставляемые (inline) функции

Некоторые функции в C++ можно определить с использованием служебного слова inline. Такая функция называется подставляемой или встраиваемой.

**Пример 80**

```
inline float Line(float x1,float y1,float x2=0,
float y2=0)
{return    sqrt(pow(x1-x2)+pow(y1-y2,2));}
```
//функция возвращает расстояние от точки с координатами(x1,y1)(по умолчанию центр координат) до точки с координатами (x2,y2).

Обрабатывая каждый вызов подставляемой функции, компилятор пытается подставить в текст программы код операторов ее тела. Спецификатор *inline* определяет для функции так называемое внутреннее связывание, которое заключается в том, что компилятор вместо вызова функции подставляет команды ее кода. При этом может увеличиваться размер программы, но исключаются затраты на передачу управления к вызываемой функции и возврата из нее. Подставляемые функции используют, если тело функции состоит из нескольких операторов.

*Подставляемыми не могут быть*:

1. рекурсивные функции;
2. функции, у которых вызов размещается до ее определения;
3. функции, которые вызываются более одного раза в выражении;
4. функции, содержащие циклы, переключатели и операторы переходов;
5. функции, которые имеют слишком большой размер, чтобы сделать подстановку.

### 4.8.8. Функции с переменным числом параметров

В C++ допустимы функции, у которых при компиляции не фиксируется число параметров, и, кроме того может быть неизвестен тип этих параметров. Количество и тип параметров становится известным только в момент вызова, когда явно задан список фактических параметров. Каждая функция с переменным числом параметров должна иметь хотя бы один обязательный параметр. Определение функции с переменным числом параметров:

**тип имя (явные параметры,. . . )**

**{тело функции }**

После списка обязательных параметров ставится запятая, а затем многоточие, которое показывает, что дальнейший контроль соответствия количества и типов параметров при обработке вызова функции производить не нужно. Сложность заключается в определении начала и конца списка параметров, поэтому каждая функция с переменным числом параметров должна иметь механизм определения количества и типов параметров.

Существует *два подхода*:

1) известно количество параметров, которое передается как обязательный параметр;

2) известен признак конца списка параметров;

**Пример 81.** Найти среднее арифметическое последовательности чисел, если известен признак конца списка параметров (подход 1).

```
#include<iostream.h>
float sum(int k, . . .)
{
int *p=&k;//настроили указатель на параметр k
int s=0;
for(;k!=0;k--)
s+=*(++p);
return s/k;
}
void main()
{
cout<<"\n4+6="<<sum(2,4,6);/*находит      среднее
арифметическое 4+6*/
cout<<"\n1+2++3+4="<<sum(4,1,2,3,4);/*находит
среднее арифметическое 1+2+3+4*/
}
```

В примере 81 для доступа к списку параметров используется указатель *p типа int. Он устанавливается на начало списка параметров в памяти, а затем перемещается по адресам фактических параметров (++p).

**Пример 82.** Найти среднее арифметическое последовательности чисел, если известен признак конца списка параметров (подход 2).

```
#include<iostream.h>
int sum(int k, . . .)
{
int *p=&k;//настроили указатель на параметр k
int  s=*p;/*значение первого параметра присвоили
s*/
for(int i=1;p!=0;i++)//пока нет конца списка
s+=*(++p);
return s/(i-1);
}
void main()
{
cout<<"\n4+6="<<sum(4,6,0);/*находит      среднее
арифметическое 4+6*/
cout<<"\n1+2++3+4="<<sum(1,2,3,4,0);/*находит
среднее арифметическое 1+2+3+4*/
}
```

### 4.8.9. Перегрузка функций

Цель перегрузки состоит в том, чтобы функция с одним именем поразному выполнялась и возвращала разные значения при обращении к ней с различными типами и различным числом фактических параметров. Для *обеспечения перегрузки* необходимо для каждой перегруженной функции определить возвращаемые значения и передаваемые параметры так, чтобы каждая перегруженная функция отличалась от другой функции с тем же именем. Компилятор определяет какую функцию выбрать по типу фактических параметров.

**Пример 83**

```
#include<iostream.h>
#include <string.h>
int max(int a,int b)
{
if(a>b)return a;
 else return b;
}
float max(float a,float b)
{
if(a>b)return a;
 else return b;
}
char*max(char*a,char*b)
{
 if(strcmp(a,b)>0) return a;
 else return b;
}
void main()
{
int a1,b1;
float a2, b2;
 char s1[20];
 char s2[20];
cout<<«\nfor int:\n»;
cout<<«a=?»;cin>>a1;
cout<<«b=?»;cin>>b1;
cout<<«\nMAX=«<<max(a1,b1)<<«\n»;
cout<<«\nfor float:\n»;
cout<<«a=?»;cin>>a2;
cout<<«b=?»;cin>>b2;
```

```
cout<<«\nMAX=«<<max(a2,b2)<<«\n»;
cout<<«\nfor char*:\n»;
cout<<«a=?»;cin>>s1;
cout<<«b=?»;cin>>s2;
cout<<«\nMAX=«<<max(s1,s2)<<«\n»;
}
```

*Правила описания перегруженных функций*:

1) Перегруженные функции должны находиться в одной области видимости.

2)Перегруженные функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать. В разных вариантах перегруженных функций может быть разное количество умалчиваемых параметров.

3)Функции не могут быть перегружены, если описание их параметров отличается только модификатором const или наличием ссылки.

Например, функции `int&f1(int&,const int&){. . . }` и `int f1(int,int){. . . }` – не являются перегруженными, т.к. компилятор не сможет узнать какая из функций вызывается: нет синтаксических отличий между вызовом функции, которая передает параметр по значению и функции, которая передает параметр по ссылке.

### 4.8.10. Шаблоны функций

Шаблоны вводятся для того, чтобы автоматизировать создание функций, обрабатывающих разнотипные данные. Например, алгоритм сортировки можно использовать для массивов различных типов. При перегрузке функции для каждого используемого типа определяется своя функция. Шаблон функции определяется один раз, но определение параметризируется, т.е. тип данных передается как параметр шаблона. Формат шаблона:

**template <class имя_типа [,class имя_типа]>**
**заголовок_функции**
**{тело функции}**

Таким образом, шаблон семейства функций состоит из 2 частей – заголовка шаблона: **template<список параметров шаблона>** и обыкновенного определения функции, в котором вместо типа возвращаемого значения и/или типа параметров, записывается имя типа, определенное в заголовке шаблона.

**Пример 84**

```
    //шаблон функции, которая находит абсолютное
значение числа любого типа
    template<class type>//type – имя параметризируе-
мого типа
    type abs(type x)
    {
    if(x<0)return -x;
    else return x;
    }
```

Шаблон служит для автоматического формирования конкретных описаний функций по тем вызовам, которые компилятор обнаруживает в программе. Например, если в программе вызов функции осуществляется как abs(-1.5), то компилятор сформирует определение функции double abs(double x){. . . }.

**Пример 85**

```
    //шаблон функции, которая меняет местами две
переменных
    template <class T>//T – имя параметризируемого
типа
    void change(T*x,T*y)
    {T z=*x;*x=*y;*y=z;}
    Вызов этой функции может быть :
    long k=10,l=5;
    change(&k,&l);
```

Компилятор сформирует определение:

void change(long*x,long*y){ long z=*x;*x=*y;*y=z;}

**Пример 86**

```
    #include<iostream.h>
    template<class Data>
    Data&rmax(int n,Data a[])
    {
    int im=0;
    for(int i=0;i<n;i++)
    if(a[im]<a[im])im=i;
    return d[im];//возвращает ссылку на максимальный
элемент в массиве
    }
    void main()
    {int n=5;
    int x[]={10,20,30,15};
```

```
cout<<"\nrmax(n,x)="<<rmax(n,x)<<"\n";
rmax(n,x)=0;
for(int i=0;i<n;i++)
cout<<x[i]<<" ";
cout<<"\n";
float y[]={10.4,20.2,30.6,15.5};
cout<<"\nrmax(n,y)="<<rmax(n,y)<<"\n";
rmax(4,y)=0;
for(in i=0;i<n;i++)
cout<<y[i]<<" ";
cout<<"\n";
}
```

Результаты:
rmax(n,x)=30
10 20 0 15
rmax(n,y)=30.6
10.4 20.2 0 15.5

*Основные свойства* параметров шаблона функций

1. Имена параметров должны быть уникальными во всем определении шаблона.
2. Список параметров шаблона не может быть пустым.
3. В списке параметров шаблона может быть несколько параметров, каждый из них начинается со слова class.

### 4.8.11. Указатель на функцию

Каждая функция характеризуется типом возвращаемого значения, именем и списком типов ее параметров. Если имя функции использовать без последующих скобок и параметров, то он будет выступать в качестве указателя на эту функцию, и его значением будет выступать адрес размещения функции в памяти. Это значение можно будет присвоить другому указателю. Тогда этот новый указатель можно будет использовать для вызова функции. Указатель на функцию определяется следующим образом:

**тип_функции(*имя_указателя)(спецификация параметров)**

**Пример 87**

1. `int f1(char c){. . . . }` //определение функции
`int(*ptrf1)(char);` //определение указателя на функцию f1
2. `char*f2(int k,char c){. . . .}` //определение функции
`char*ptrf2(int,char);` //определение указателя

В определении указателя количество и тип параметров должны совпадать с соответствующими типами в определении функции, на которую ставится указатель.

Вызов функции с помощью указателя имеет вид:

**(*имя_указателя)(список фактических параметров);**

**Пример 88**

```
#include <iostream.h>
void f1()
{cout<<”\nfunction f1”;}
void f2()
{cout<<”\nfunction f2”;}
void main()
{
void(*ptr)();//указатель на функцию
ptr=f2;/*указателю присваивается адрес функции
f2*/
(*ptr)();//вызов функции f2
ptr=f1;/*указателю присваивается адрес функции
f1*/
(*ptr)();//вызов функции f1с помощью указателя
}
```

При определении указатель на функцию может быть сразу проинициализирован: `void (*ptr)()=f1;`

Указатели на функции могут быть объединены в массивы. Например, `float(*ptrMas[4])(char)` – описание массива, который содержит 4 указателя на функции. Каждая функция имеет параметр типа char и возвращает значение типа *float*. Обратиться к такой функции можно следующим образом:

```
float a=(*ptrMas[1])('f');/*обращение ко второй
функции*/
```

**Пример 89**

```
#include <iostream.h>
#include <stdlib.h>
void f1()
{cout<<«\nThe end of work»;exit(0);}
void f2()
{cout<<«\nThe work #1»;}
void f3(){cout<<«\nThe work #2»;}
void main()
{
```

```
void(*fptr[])()={f1,f2,f3};
int n;
while(1)//бесконечный цикл
{
cout<<«\n Enter the number»;
cin>>n;
fptr[n]();//вызов функции с номером n
}
}
```

Указатели на функции удобно использовать в тех случаях, когда функцию надо передать в другую функцию как параметр.

**Пример 90**

```
#include <iostream.h>
#include <math.h>
typedef float(*fptr)(float);/*тип – указатель на функцию*/
float root(fptr f, float a, float b, float e);/*решение уравнения методом половинного деления уравнение передается с помощью указателя на функ-цию*/
{float x;
do
{
x=(a+b)/2;
if ((*f)(a)*f(x)<0)b=x; else a=x;
}
while((*f)(x)>e&&fabs(a-b)>e);
return x;
}
float testf(float x)
{return x*x-1;}
void main()
{
float res=root(testf,0,2,0.0001);
cout<<”\nX=”<<res;
}
```

### 4.8.12. Ссылки на функцию

Подобно указателю на функцию определяется и ссылка на функцию:

**тип_функции(&имя_ссылки)(параметры)**
**инициализирующее_выражение;**

**Пример 91.**

```
int f(float a,int b){. . . }/*определение функ-
ции*/
int(&fref)(float,int)=f;//определение ссылки
```

Использование имени функции без параметров и скобок будет восприниматься как адрес функции. Ссылка на функцию является синонимом имени функции. Изменить значение ссылки на функцию нельзя, поэтому более широко используются указатели на функции, а не ссылки.

**Пример 92.**

```
#include <iostream.h>
void f(char c)
{cout<<"\n"<<c;}
void main()
{
void (*pf)(char);//указатель на функцию
void(&rf)(char);//ссылка на функцию
f('A');//вызов по имени
pf=f;//указатель ставится на функцию
(*pf)('B');//вызов с помощью указателя
rf('C');//вызов по ссылке
}
```

## 4.9. Типы данных, определяемые пользователем

### 4.9.1. Переименование типов

Типу можно задавать имя с помощью ключевого слова *typedef*:
**typedef тип имя_типа [размерность];**

**Пример 93**

```
typedef unsigned int UNIT;
typedef char Msg[100];
```

Такое имя можно затем использовать также как и стандартное имя типа:

```
UNIT a,b,c;//переменные типа unsigned int
Msg str[12];/* массив из 10 строк по 100 симво-
лов*/
```

### 4.9.2. Перечисления

Если надо определить несколько именованных констант таким образом, чтобы все они имели разные значения, можно воспользоваться перечисляемым типом:

**enum [имя_типа] {список констант};**

Константы должны быть целочисленными и могут инициализироваться обычным образом. Если инициализатор отсутствует, то первая константа обнуляется, а остальным присваиваются значение на единицу большее, чем предыдущее.

**Пример 94**

```
enum Err{ErrRead, ErrWrite, ErrConvert);
Err error;
. . . .
switch(error)
{
case ErrRead: …..
case ErrWrite: …..
case ErrConvert: …..
}
```

### 4.9.3. Структуры

*Структура* – это объединенное в единое целое множество поименованных элементов данных. Элементы структуры (поля) могут быть различного типа, они все должны иметь различные имена.

*Форматы определения структурного типа* следующие:

1. **struct имя_типа //способ 1**
   **{**
   **тип 1 элемент1;**
   **тип2 элемент2;**
   **...**
   **};**

**Пример 95**

```
struct Date//определение структуры
{
int day;
int month;
int year;
};
Date birthday;//переменная типа Date
```

2. **struct //способ 2**
**{**
**тип 1 элемент1;**
**тип2 элемент2;**
**. . .**
**} список идентификаторов;**

**Пример 96**

```
struct
{
int min;
int sec;
int msec;
}time_beg,time_end;
```

В первом случае описание структур определяет новый тип, имя которого можно использовать наряду со стандартными типами.

Во втором случае описание структуры служит определением переменных.

3. Структурный тип можно также задать с помощью ключевого слова *typedef*:

**Пример 97.** способ 3

```
Typedef struct
{
floar re;
float im;
}Complex;
Complex a[100];/*массив из 100 комплексных чи-
сел.*/
```

### Инициализация структур

Для инициализации структур значения ее полей перечисляют в фигурных скобках.

**Пример 98**

1.
```
struct Student
{
 char name[20];
int kurs;
float rating;
};
Student s={”Иванов”,1,3.5};
```

2. `struct`

```
{
char name[20];
char title[30];
float rate;
}employee={“Петров», “директор”,10000};
```

### Присваивание структур

Для переменных одного и того же структурного типа определена операция присваивания. При этом происходит поэлементное копирование.

```
Student ss=s;
```

### Доступ к элементам структур

Доступ к элементам структур обеспечивается с помощью уточненных имен:

**Имя_структуры.имя_элемента**

**Пример 99**

```
#include <conio.h>
#include <iostream.h>
void main()
{
struct Student
{
char name[30];
char group[12];
float rating;
};
Student mas[35];
//ввод значений массива
for(int i=0;i<35;i++)
{
cout<<”\nEnter name:”;cin>>mas[i].name;
cout<<”\nEnter group:”;cin>>mas[i].group;
cout<<”\nEnter rating:”;cin>>mas[i].rating;
}
cout<<”Raitng <3:”;
for( i=0;i<35;i++)
if(mas[i].rating<3)
cout<<”\n”<<mas[i].name;
getch();
}
```

### Указатели на структуры

Указатели на структуры определяются также как и указатели на другие типы:

```
Student*ps;
```

**Пример 100.** Ввод указателя для типа *struct*, не имеющего имени (способ 2):

```
Struct
 {
char *name;
int age;
} *person;//указатель на структуру
```

При определении указатель на структуру может быть сразу же проинициализирован:

```
Student *ps=&mas[0];
```

Указатель на структуру обеспечивает доступ к ее элементам двумя способами:

**1.(*указатель).имя_элемента – прамой доступ:**

```
cin>>(*ps).name;
```

**2. указатель->имя_элемента – косвенный доступ:**

```
cin>>ps->title.
```

### 4.9.5. Битовые поля

*Битовые поля* – это особый вид полей структуры. При описании битового поля указывается его длина в битах (целая положительная константа).

**Пример 101.**

```
struct {
int a:10;
int b:14;
}xx,*pxx;
. . . .
xx.a=1;
pxx=&xx;
pxx->b=8;
```

Битовые поля могут быть любого целого типа. Они используются для плотной упаковки данных. Например, с их помощью удобно реализовать флажки типа «да» / «нет».

Особенностью битовых полей является то, что нельзя получить их адрес. Размещение битовых полей в памяти зависит от компилятора и аппаратуры.

### 4.9.6. Объединения

*Объединение* (*union*) – это частный случай структуры. Все поля объединения располагаются по одному и тому же адресу. Длина объединения равна наибольшей из длин его полей. В каждый момент времени в такой переменной может храниться только одно значение. Объединения применяют для экономии памяти, если известно, что более одного поля не потребуется. Также объединение обеспечивает доступ к одному участку памяти с помощью переменных разного типа.

**Пример 102**

```
union{
char s[12];
int x;
}u1;
```

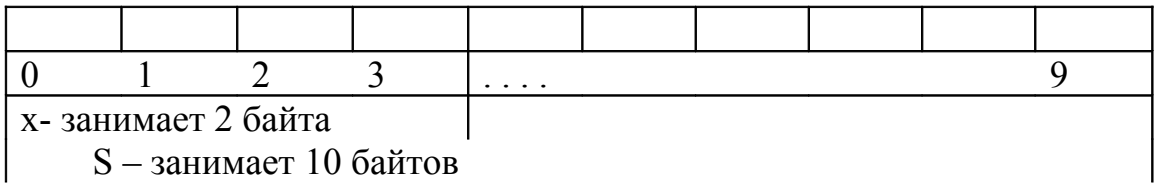



Рис. 24. Расположение объединения в памяти

Переменные *s* и *x* располагаются на одном участке памяти. Размер такого объединения будет равен 10 байтам.

**Пример 103.** Использование объединений

```
enum paytype{CARD,CHECK};//тип оплаты
struct{
  paytype ptype;/*поле, которое определяет с ка-
ким полем объединения будет*/
// выполняться работа
   union{
      char card[25];
      long check;
      };
   }info;
   switch (info.ptype)
   {
```

```
    case CARD: cout<<"\nОплата по карте:"<<info.-
card;break;
    case CHECK:cout<<"\nОплата че-
ком:"<<info.check;break;}
```

### 4.10. Динамические структуры данных

Во многих задачах требуется использовать данные, у которых конфигурация, размеры и состав могут меняться в процессе выполнения программы. Для их представления используют динамические информационные структуры. К таким структурам относят:

1) линейные списки;
2) стеки;
3) очереди;
4) бинарные деревья;

Они отличаются способом связи отдельных элементов и допустимыми операциями. Динамическая структура может занимать несмежные участки динамической памяти.

Наиболее простой динамической структурой является линейный однонаправленный список, элементами которого служат объекты структурного типа (рис. 25).

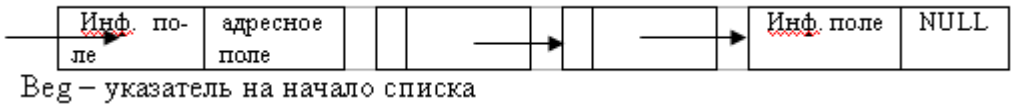



Рис. 25. Линейный однонаправленный список

#### 4.10.1. Линейный однонаправленный список

Описание простейшего элемента такого списка выглядит следующим образом:

**struct имя_типа**
**{**
**информационное поле;**
**адресное поле;**
**};**

*Информационное поле* – это поле любого, ранее объявленного или стандартного, типа. Информационных полей может быть несколько.

*Адресное поле* – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента списка.

**Пример 104**

1. 
```
struct Node
{
int key; //информационное поле
Node*next; //адресное поле
};
```
2. 
```
struct point
{
char*name;//информационное поле
int age;//информационное поле
point*next;//адресное поле
};
```

Каждый элемент списка содержит ключ, который идентифицирует этот элемент. Ключ обычно бывает либо целым числом (пример 104 1.), либо строкой (пример 104 2.).

*Над списками* можно *выполнять* следующие *операции*:

1) начальное формирование списка (создание первого элемента);
2) добавление элемента в конец списка;
3) добавление элемента в начало списка;
4) удаление элемента с заданным номером;
5) чтение элемента с заданным ключом;
6) вставка элемента в заданное место списка (до или после элемента с заданным ключом);
7) упорядочивание списка по ключу;
8) и др.

**Пример 105.** Создание и печать однонаправленного списка

```
#include <iostream.h>
#include<string.h>
//описание структуры
struct point
{char *name;//информационное поле
int age;//информационное поле
point*next;//адресное поле
};

point* make_point()
//создание одного элемента
{
 point*p=new(point);//выделить память
 char s[20];
```

```
     cout<<«\nEnter the name:»;
     cin>>s;
    p->name=new  char[strlen(s)+1];/*выделить  память
под динамическую строку символов*/
     strcpy(p->name,s);/*записать информацию в стро-
ку символов*/
     cout<<«\nEnter the age»;
     cin>>p->age;
     p->next=0;//сформировать адресное поле
    return p;
    }
    void print_point(point*p)
    /*печать  информационных  полей  одного  элемента
списка*/
    {
     cout<<«\nNAME:»<<p->name;
     cout<<«\nAGE:»<<p->age;
     cout<<«\n--------------------------------\n»;
    }
    point* make_list(int n)
    //формирование списка из n элементов
    {
     point*  beg=make_point();/*сформировать  первый
элемент*/
     point*r;
     for(int i=1;i<n;i++)
     {
          r=make_point();/*сформировать      следующий
элемент*/
          //добавление в начало списка
          r->next=beg;//сформировать адресное поле
          beg=r;/*изменить  адрес  первого  элемента
списка*/
     }
     return beg;//вернуть адрес начала списка
    }
    int print_list(point*beg)
    /*печать списка, на который указывает указатель
beg*/
    {
```

```
    point*p=beg;//p присвоить адрес первого элемен-
та списка
   int k=0;/*счетчик количества напечатанных эле-
ментов */
    while(p)//пока нет конца списка
    {
        print_point(p);/*печать элемента, на кото-
рый указывает элемент p*/
        p=p->next;//переход к следующему элементу
   k++;
    }
   return k;//количество элементов в списке
   }

   void main()
   {
    int n;
    cout<<«\nEnter the size of list»;
        cin>>n;
    point*beg=make_list(n);//формирование списка
   if(!print_list(beg))    cout<<«\nThe    list    is
empty»;}//печать списка
```

**Пример 106.** Удаление из однонаправленного списка элемента с номером *k* (рис. 26.).

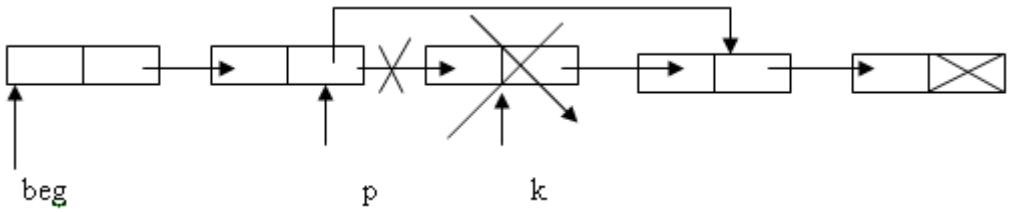



Рис. 26. Удаление элемента с номером k из однонаправленного списка

```
     point*del_point(point*beg,int k)
     //удаление элемента с номером к
     {
        point*p=beg,/*поставить     вспомогательную
  переменную на начало списка*/
     *r;//вспомогательная переменная для удаления
        int i=0;//счетчик элементов в списке
        if(k==0)
        {//удалить первый элемент
```

```
            beg=p->next;
            delete[]p->name;/*удалить  динамическое
поле name*/
            delete[]p;//удалить элемент из списка
            return beg;/*вернуть адрес первого эле-
мента списка*/
        }
        while(p)//пока нет конца списка
        {
            if(i==k-1)/*дошли до элемента с номером
k-1, чтобы поменять его поле next*/
            {//удалить элемент
                r=p->next;/*поставить r на удаляе-
мый элемент*/
                if(r)//если p не последний элемент
                {
                p->next=r->next;/*исключить  r  из
списка*/
                delete[]r->name;/*удалить динамиче-
ское поле name*/
                delete[]r;/*удалить   элемент   из
списка*/
                }
            else p->next=0;/*если p -последний эле-
мент, то в поле next присвоить NULL*/
            }
            p=p->next;/*переход  к  следующему  эле-
менту списка*/
            i++;//увеличить счетчик элементов
        }
    return beg;//вернуть адрес первого элемента}
```

### 4.10.2. Работа с двунаправленным списком

Двунаправленный список представлен на рис. 27. Пример 107 даёт представление о работе с двунаправленным списком.

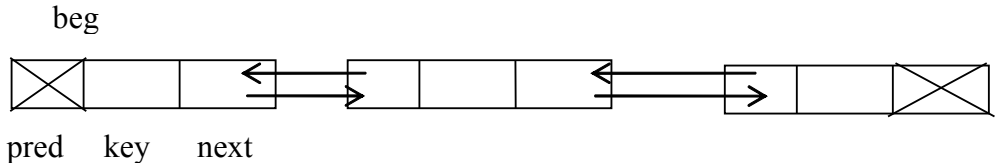



Рис. 27. Двунаправленный список

**Пример 107.** Создать двунаправленный список, выполнить удаление элемента с заданным номером, добавление элемента с заданным номером, печать полученных списков.

```
#include <iostream.h>
struct point//описание структуры
{
 int key;//ключевое поле
 point* pred,*next;//адресные поля
};
point*make_list()
{
 int n;
 cout<<«n-?»;cin>>n;
 point *p,*r,*beg;
 p=new (point);//создать первый элемент
 beg=p;/*запомнить адрес в переменную beg, в которой хранится начало списка*/
 cout<<«key-?»;cin>>p->key;/*заполнить ключевое поле*/
 p->pred=0;p->next=0;//запомнить адресные поля
 for(int i=1;i<n;i++)/*добавить элементы в конец списка*/
 {
     r=new(point);//новый элемент
     cout<<«key-?»;cin>>r->key;//адресное поле
     p->next=r;//связать начало списка с r
     r->pred=p;//связать r с началом списка
     r->next=0;/*обнулить последнее адресное поле*/
     p=r;/*передвинуть p на последний элемент списка*/
 }
 return beg;//вернуть первый элемент списка
}
```

```
void print_list(point *beg)
{
 if (beg==0)//если список пустой
 {
    cout<<«The list is empty\n»;
     return;
 }
 point*p=beg;
 while(p)//пока не конец списка
 {
     cout<<p->key<<«\t»;
     p=p->next;//перейти на следующий
 }
 cout<<«\n»;
}
point* del_point(point*beg, int k)
{
 point *p=beg;
 if(k==0)//удалить первый элемент
 {
     beg=beg->next;/*переставить начало списка
на следующий элемент*/
     if(beg==0)return 0;/*если в списке только
один элемент*/
     beg->pred=0;/*обнулить адрес предыдущего
элемента */
     delete p;//удалить первый
     return beg;//вернуть начало списка
 }
//если удаляется элемент из середины списка
 for(int i=0;i<k-1&&p!=0;i++,p=p->next);/*пройти
по списку либо до элемента с предыдущим номером,
либо до конца списка*/
 if(p==0||p->next==0)return beg;//если в списке
нет элемента с номером k
 point*r=p->next;//встать на удаляемый элемент
 p->next=r->next;//изменить ссылку
 delete r;//удалить r
 r=p->next;//встать на следующий
 if(r!=0)r->pred=p;/*если r существует, то свя-
зать элементы*/
```

```
    return beg;//вернуть начало списка
   }
   point* add_point(point *beg,int k)
   {
    point *p;
    p=new(point);/*создать новый элемент и запол-
нить ключевое поле*/
    cout<<«key-?»;cin>>p->key;
    if(k==0)//если добавляется первый элемент
    {
        p->next=beg;//добавить перед beg
        p->pred=0;//обнулить адрес предыдущего
        beg->pred=p;/*связать список с добавленным
элементом*/
        beg=p;//запомнить первый элемент в beg
        return beg;//вернуть начало списка
    }
        point*r=beg;//встать на начало списка
    for(int          i=0;i<k-1&&r->next!=0;i++,r=r-
>next);/*пройти по списку либо до конца списка,
либо до элемента с номером k-1*/
    p->next=r->next;//связать p с концом списка
    if(r->next!=0)r->next->pred=p;/*если элемент не
последний, то связать конец списка с p*/
    p->pred=r;//связать p и r
    r->next=p;
    return beg;//вернуть начало списка
   }


   void main()
   {
    point*beg;
   int i,k;
   do
   {
    cout<<«1.Make list\n»;
    cout<<«2.Print list\n»;
    cout<<«3.Add point\n»;
    cout<<«4.Del point\n»;
    cout<<«5.Exit\n»;
```

```
cin>>i;
 switch(i)
 {
 case 1:
      {beg=make_list();break;}
 case 2:
      {print_list(beg);break;}
 case 3:
      {
           cout<<«\nk-?»;cin>>k;
           beg=add_point(beg,k);
           break;
      }
 case 4:
      {
           cout<<«\nk-?»;cin>>k;
           beg=del_point(beg,k);
           break;
      }
 }
}
while(i!=5);
}
```

## 4.11. Ввод-вывод в C++

### 4.11.1. Потоковый ввод-вывод

*Файл* – это именованная область внешней памяти. Файл имеет следующие характерные особенности:

1) имеет имя на диске, что дает возможность программам работать с несколькими файлами;

2) длина файла ограничивается только емкостью диска.

Особенностью языка C++ является отсутствие в этом языке структурированных файлов. Все файлы рассматриваются как не структурированная последовательность байтов. При таком подходе понятие файла распространяется и на различные устройства. Одни и те же функции используются как для обмена данными с файлами, так и для обмена с устройствами.

Библиотека C++ поддерживает *три уровня ввода-вывода*:

1) потоковый ввод-вывод;

2) ввод-вывод нижнего уровня;

3) ввод-вывод для консоли портов (зависит от конкретной ОС).

На уровне потокового ввода-вывода обмен данными производится побайтно, т.е. за одно обращение к устройству (файлу) производится считывание или запись фиксированной порции данных (512 или 1024 байта). При вводе с диска или при считывании из файла данные помещаются в буфер ОС, а затем побайтно или порциями передаются программе пользователя. При выводе в файл данные также накапливаются в буфере, а при заполнении буфера записываются в виде единого блока на диск. Буферы ОС реализуются в виде участков основной памяти. Таким образом, поток – это файл вместе с предоставленными средствами буферизации. Функции библиотеки С, поддерживающие обмен данными на уровне потока позволяют обрабатывать данные различных размеров и форматов. При работе с потоком можно:

1) открывать и закрывать потоки (при этом указатели на поток связываются с конкретными файлами);

2) вводить и выводить строки, символы, форматированные данные, порции данных произвольной длины;

3) управлять буферизацией потока и размером буфера;

4) получать и устанавливать указатель текущей позиции в файле.

Прототипы функций ввода-вывода находятся в заголовочном файле <*stdio.h*>, который также содержит определения констант, типов и структур, необходимых для обмена с потоком.

#### 4.11.1.1 Открытие и закрытие потока

Прежде, чем начать работать с потоком, его надо инициировать, т. е. открыть. При этом поток связывается со структурой предопределенного типа FILE, определение которой находится в файле <*stdio.h*>. В структуре находится указатель на буфер, указатель на текущую позицию и т.п. При открытии потока возвращается указатель на поток, т.е. на объект типа FILE. Указатель на поток должен быть объявлен следующим образом:

```
#include <stdio.h>
. . . . . .
FILE*f; //указатель на поток
```

Указатель на поток приобретает значение в результате выполнения функции открытия потока:

**FILE* fopen(const char*filename,const char*mode);**

где **const char*filename** – строка, которая содержит имя файла, связанного с потоком,

**const char*mode** – строка режимов открытия файла.

**Пример 108.**

```
f=fopen("t.txt","r");
```

где *t.txt* – имя файла, *r* – режим открытия файла.

Файл связанный с потоком можно открыть в одном из 6 режимов, представленных в табл. 19.

Таблица 19

Режимы открытия файла, связанного с потоком

| Режим | Описание режима открытия файла |
|---|---|
| r | Файл открывается для чтения, если файл не существует, то выдается ошибка при исполнении программы |
| w | Файл открывается для записи, если файл не существует, то он будет создан, если файл уже существует, то вся информация из него стирается |
| a | Файл открывается для добавления, если фай не существует, то он будет создан, если существует, то информация из него не стирается, можно выполнять запись в конец файла |
| r+ | Файл открывается для чтения и записи, изменить размер файла нельзя, если файл не существует, то выдается ошибка при исполнении программы |
| w+ | Файл открывается для чтения и записи, если файл не существует, то он будет создан, если файл уже существует, то вся информация из него стирается |
| a+ | Файл открывается для чтения и записи, если фай не существует, то он будет создан, если существует, то информация из него не стирается, можно выполнять запись в конец файла |

Поток можно открывать в текстовом (*t*) или двоичном режиме (*b*). В *текстовом режиме* поток рассматривается как совокупность строк, в конце каждой строки находится управляющий символ ‘\n’. В *двоичном режиме* поток рассматривается как набор двоичной информации. Текстовый режим устанавливается по умолчанию.

В файле *stdio.h* определена константа EOF, которая сообщает об окончании файла (отрицательное целое число).

*При открытии потока* могут возникать следующие *ошибки*:

1) файл, связанный с потоком не найден (при чтении из файла);

2) диск заполнен (при записи);

3) диск защищен от записи (при записи) и т.п.

В этих случаях указатель на поток приобретет значение NULL (0). Указатель на поток, отличный от аварийного не равен 0.

Для *вывода об ошибке при открытии потока* используется стандартная библиотечная функция из файла <*stdio.h*>:

**void perror (const char*s);**

Эта функция выводит строку символов, не которую указывает указатель *s*, за этой строкой размещается двоеточие пробел и сообщение об ошибке. Текст сообщения выбирается на основании номера ошибки. Номер ошибки заносится в переменную *int errno* (определена в заголовочном файле *errno.h*).

После того как файл открыт, в него можно записывать информацию или считывать информацию, в зависимости от режима.

Открытые *файлы* после окончания работы рекомендуется *закрыть явно*. Для этого используется функция:

**int fclose(FILE*f);**

Изменить режим работы с файлом можно только после закрытия файла.

**Пример 109**

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
void main()
{
FILE *f;
char filename[20];
cout<<"\nEnter the name of file:"; cin>>file-
name;
if(f=fopen(filename,"rb")==0)/*открываем для
чтения в бинарном режиме и проверяем*/
// возникает ли ошибка при открытии файла
{
perror(strcat"error in
file :",filename);//strcat складывает две строки
exit(0);//выход из программы
}
. . . . .
fclose(f);
}
```

Для *текстового файла*:

```
if (f=fopen(filename,"rt")==0) /*открываем для чтения
```

и проверяем возникает ли ошибка при //открытии файла*/

```
if (f=fopen(filename,"r")==0) /*открываем для чтения и
```

проверяем возникает ли ошибка при //открытии файла.*/

### 4.11.2. Стандартные файлы и функции для работы с ними

Когда программа начинает выполняться, автоматически открываются несколько потоков, из которых основными являются:

1. стандартный поток ввода (*stdin*);
2. стандартный поток вывода (*stdout*);
3. стандартный поток вывода об ошибках (*stderr*).

По умолчанию *stdin* ставится в соответствие клавиатура, а потокам *stdout* и *stderr* – монитор. Для ввода-вывода с помощью стандартных потоков используются функции:

1. **getchar()/putchar()** – ввод-вывод отдельного символа;
2. **gets()/puts()** – ввод-вывод строки;
3. **scanf()/printf()** – форматированный ввод/вывод.

Функции рассматривались, когда мы рассматривали строковые и символьные данные. Теперь мы можем связать их со стандартными потоками: ввод осуществляется из стандартного потока *stdin*, вывод осуществляется в стандартный поток *stdout*. Аналогично работе со стандартными потоками выполняется ввод-вывод в потоки, связанные с файлами.

### 4.11.3. Символьный ввод-вывод

Для символьного ввода-вывода используются функции:

**int fgetc(FILE*fp)**, где *fp* – указатель на поток, из которого выполняется считывание. Функция возвращает очередной символ в форме *int* из потока *fp*. Если символ не может быть прочитан, то возвращается значение EOF.

**int fputc(int c, FILE*fp)**, где *fp* – указатель на поток, в который выполняется запись, c – переменная типа *int*, в которой содержится записываемый в поток символ. Функция возвращает записанный в поток fp символ в форме *int*. Если символ не может быть записан, то возвращается значение EOF.

**Пример 110**

```
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
void main()
{
FILE *f;
char c;
char *filename=”f.txt”;
```

```
    if((f=fopen(filename,”r”)==0)
    {
    perror(filename);exit(0);
    }
    while(c=fgetc(f)!=EOF)
    putchar(c);/*вывод с на стандартное устройство
вывода*/
    fclose(f);
    }
```

### 4.11.4. Строковый ввод-вывод

Для построчного ввода-вывода используются следующие функции:

1. **char* fgets(char* s,int n,FILE* f)**, где

*char*s* – адрес, по которому размещаются считанные байты,

*int n* – количество считанных байтов,

*FILE* f* – указатель на файл, из которого производится считывание.

Прием байтов заканчивается после передачи *n*-1 байтов или при получении управляющего символа ‘\n’. Управляющий символ тоже передается в принимающую строку. Строка в любом случае заканчивается ‘\0’. При успешном завершении считывания функция возвращает указатель на прочитанную строку, при неуспешном – 0.

2. **int puts(char* s, FILE* f)**, где

*char*s* – адрес, из которого берутся записываемые в файл байты,

*FILE* f* – указатель на файл, в который производится запись.

Символ конца строки (‘\0’) в файл не записывается. Функция возвращает EOF, если при записи в файл произошла ошибка, при успешной записи возвращает неотрицательное число.

**Пример 111.** Копирование файла *in* в файл *out*

```
    int MAXLINE=255;//максимальная длина строки
    FILE *in;//исходный файл
    *out;//принимающий файл
    char* buf[MAXLINE];/*строка, с помощью которой
выполняется копирование*/
    in=fopen(“f1.txt”,”r”);/*открыть исходный файл
для чтения*/
    out=fopen(“f2.txt”,”w”);/*открыть принимающий
файл для записи*/
    while(fgets(buf,MAXLINE,in)!=0)/*прочитать байты
из файла in в строку buf*/
```

```
    fputs(buf,out);/*записать байты из строки buf в
файл out*/
    fclose(in);fclose(out);//закрыть оба файла
```

### 4.11.5. Блоковый ввод-вывод

Для блокового ввода-вывода используются функции:

1. **int fread(void*ptr,int size, int n, FILE*f)**,

где *void*ptr* – указатель на область памяти, в которой размещаются считанные из файла данные,

*int size* – размер одного считываемого элемента,

*int n* – количество считываемых элементов,

*FILE*f* – указатель на файл, из которого производится считывание.

В случае успешного считывания функция возвращает количество считанных элементов, иначе – EOF.

2. **int fwrite(void*ptr,int size, int n, FILE*f)**,

где *void*ptr* – указатель на область памяти, в которой размещаются считанные из файла данные,

*int size* – размер одного записываемого элемента,

*int n* – количество записываемых элементов,

*FILE*f* – указатель на файл, в который производится запись.

В случае успешной записи функция возвращает количество записанных элементов, иначе – EOF.

**Пример 112**

```
struct Employee
{
char name[30];
char title[30];
float rate;
};
void main()
{
Employee e;
FILE *f;
if((f=fopen("f.dat","wb"))==NULL)
{
cout<<"\nCannot open file for writing";
exit(1);
}
int n;
//запись в файл
```

```
printf("\nN-?");
scanf("%d",&n);
for(int i=0;i<n;i++)
{
//формируем структуру e
printf("\nname:");scanf("%s",&e.name);
printf("\ntitle:");scanf("%s",&e.title);
printf("\nrate:");scanf("%s",&e.rate);
//записываем e в файл
fwrite(&e,sizeof(Employee),1,f);
}
fclose(f);
//чтение из файла
if((f=fopen("f.dat","rb"))==NULL)
{
cout<<"\nCannot open file for reading";
exit(2);
}
while(fread(&e,sizeof(Employee)1,f)
{
printf("%s % s%f", e.name, e.title, e.rate)
}
fclose(f);
}
```

### 4.11.6. Форматированный ввод-вывод

В некоторых случаях информацию удобно записывать в файл без преобразования, т.е. в символьном виде пригодном для непосредственного отображения на экран. Для этого можно использовать функции форматированного ввода-вывода:

1. **int fprintf(FILE *f, const char*fmt,. . .)**, где

*FILE*f* – указатель на файл, в который производится запись,

*const char*fmt* – форматная строка,

. . . – список переменных, которые записываются в файл.

Функция возвращает число записанных символов.

2. **int fscanf(FILE *f, const char*fmt, par1,par2, . . .)**, где

*FILE*f* – указатель на файл, из которого производится чтение,

*const char*fmt* – форматная строка,

*par*1, *par*2, … – список переменных, в которые заносится информация из файла.

Функция возвращает число переменных, которым присвоено значение.

**Пример 113**

```
void main()
{
FILE *f;
int n;
if((f=fopen("int.dat","w"))==0)
{
perror("int.dat");
exit(0);
}
for(n=1;n<11;n++)
fprinf(f,"\n%d   %d",n,n*n);
fclose(f);
if((f=fopen("int.dat","r"))==0)
{
perror("int.dat");
exit(1);
}
int nn;
while(fscanf(f, "%d%d",&n,&nn))
printf("\n%d %d",n,nn);
fclose(f);
}
```

### 4.11.6.1 Прямой доступ к файлам

Рассмотренные ранее средства обмена с файлами позволяют записывать и считывать данные только последовательно. Операции чтения/записи всегда производятся, начиная с текущей позиции в потоке. Начальная позиция устанавливается при открытии потока и может соответствовать начальному или конечному байту потока в зависимости от режима открытия файла. При открытии потока в режимах "*r*" и "*w*" указатель текущей позиции устанавливается на начальный байт потока, при открытии в режиме "*a*" – за последним байтом в конец файла. При выполнении каждой операции указатель перемещается на новую текущую позицию в соответствии с числом записанных/прочитанных байтов.

Средства прямого доступа дают возможность перемещать указатель текущей позиции в потоке на нужный байт. Для этого используется функция

**int fseek(FILE *f, long off, int org)**, где

*FILE *f* – указатель на файл,
*long off* – позиция смещения
*int org* – начало отсчета.

Смещение задается выражение или переменной и может быть отрицательным, т.е. возможно перемещение как в прямом, так и в обратном направлениях. Начало отсчета задается одной из определенных в файле <*stdio.h*> констант:

SEEK_SET ==0 – начало файла;
SEEK_CUR==1 – текущая позиция;
SEEK_END ==2 – конец файла.

Функция возвращает 0, если перемещение в потоке выполнено успешно, иначе возвращает ненулевое значение.

**Пример 114**

```
fseek(f,0L,SEEK_SET); /*перемещение к началу потока из текущей позиции*/
fseek(f,0L,SEEK_END); /*перемещение к концу потока из текущей позиции*/
fseek(f,-(long)sizeof(a),SEEK_SET);  //перемещение назад на длину переменной a.
```

Кроме этой функции, для прямого доступа к файлу используются:

```
long tell(FILE *f);/*получает значение указателя текущей позиции в потоке*/
void rewind(FILE *f);/*установить значение указателя на начало потока.*/
```

### 4.11.6.2 Удаление и добавление элементов в файле

Удаление и добавление элементов в файле наглядно представлено в примере 115 и примере 116.

**Пример 115**

```
void del(char *filename)
{
//удаление записи с номером x
FILE *f, *temp;
f=fopen(filename,”rb”);//открыть исходный файл для чтения
temp=fopen(“temp”,”wb”)//открыть вспомогательный файл для записи
student a;
```

```
      for(long  i=0;.fread(&a,sizeof(student),1,f);i+
+)
          if(i!=x)
          {
         fwrite(&a,sizeof(student)1,temp);
          }
          else
          {
              cout<<a<<« - is deleting...»;
          }
          fclose(f); fclose(temp);
          remove(filename);
         rename("temp", filename);
    }
```

**Пример 116**

```
    void add(char *filename)
    {
    //добавление в файл
     student a;
     int n;
     f=fopen(filename,"ab");/*открыть  файл  для  до-
бавления*/
         cout<<«\nHow many records would you add to
file?»;
     cin>>n;
     for(int i=0;i<n;i++)
     {
         прочитать объект
          fwrite(&a,sizeof(student),1,f);/*записать в
файл*/
     }
    fclose(f);//закрыть файл
    }
```

## 4.12 Вопросы для самоконтроля

1. На каком этапе происходит компиляция программы, написанная на языке С++.
2. Вещественный тип данных в языке С++.
3. Зарезервированное слово в языке С++, используемое для построения оператора цикла.

4. Знак, с которого начинается директива препроцессора в языке С++.
5. К какому классу задач относится задача: перевернуть массив?
6. Определение текстовой константы в языке С++.
7. Зарезервированное слово в языке С++, используемое для обозначения перечислений.
8. Наиболее простая динамическая структура.
9. Этапы обработки исходной программы, подготовленной на С++ в виде текстового файла.
10. Ключевые слова, которые применяются для обозначения операторов цикла языка С++.
11. Методы, применяемые для сортировки.
12. Синонимы понятия «объявление функции» в языке С++.
13. Динамические структуры данных.
14. Программная реализация блока в С++.
15. Общий вид условной (тернарной) операции.
16. Алгоритм для нахождения максимального элемента массива.

# СПИСОК ЛИТЕРАТУРЫ


1. Турский В. Методология программирования. – М.: Мир, 1981. – 547 с.
2. Буч Г. Объектно-ориентированное проектирование с примерами применения: пер. с англ. – М.: Конкорд, 1992. – 214 с.
3. Жоголев Е.А. Система программирования с использованием библиотеки подпрограмм // Система автоматизация программирования. – М.: Физматгиз, 1961. – С. 15–52.
4. Информатика. Базовый курс / С.В. Симонович и др. – СПб.: Питер, 1999. – 640 с.
5. История развития вычислительной техники [Электронный ресурс]. – Режим доступа: http://www.kolomna-school7-ict.narod.ru/st10501.htm (дата обращения: 27.09.2011)
6. Каймин В.А. Информатика: учебник. – М.: ИНФРА-М, 2000. – 232 с.
7. Калиш Г.Г. Основы вычислительной техники: учеб. пособ. для ср. проф. уч. заведений. – М.: Высш. шк., 2000. – 271 с.
8. Кнут Д.Э. Искусство программирования: учеб. пособие. – М.: Вильямс, 2000. – 832 с.
9. Могилёв А.В. и др. Информатика: учеб. пособие для студ. пед. вузов. – М.: Академия, 1999. – 816 с.
10. Обобщённое программирование [Электронный ресурс]. – Режим доступа: http://ru.wikipedia.org/wiki (дата обращения: 29.09.2011)
11. Павловская Т.А. C/C++. Программирование на языке высокого уровня. – СПб.: Питер, 2010.
12. Программирование на C++ и C#. [Электронный ресурс]. – Режим доступа: http://trubetskoy1.narod.ru/ (дата обращения: 30.09.2011)
13. Рогановой Н.А., Андреева С.В. Практическая информатика, Часть 1 [Электронный ресурс]. – Режим доступа: http://www.ctc.msiu.ru/materials/books.php (дата обращения: 28.09.2011).

14. Дейкстра Э. Заметки по структурному программированию // Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М.: Мир, 1975. – С. 7–97.

# ПРИЛОЖЕНИЕ

Таблица

Поколения ЭВМ

| **Показатель** | **Поколения ЭВМ** | | |
|---|---|---|---|
| | Первое 1951–1954 | Второе 1958–1960 | Третье 1965–1966 |
| Элементная база процессора | Электронные лампы | Транзисторы | Интегральные схемы (ИС) |
| Элементная база ОЗУ | Электронно-лучевы трубки | Ферритовые сердечники | Ферритовые сердечники |
| Максимальная ёмкость ОЗУ, байт | $10^2$ | $10^3$ | $10^4$ |
| Максимальное быстродействие процессора (оп\с) | $10^4$ | $10^6$ | $10^7$ |
| Языки программирования | Машинный код | + Ассемблер | +Процедурные языки высокого уровня (ЯВУ) |
| Средства связи пользователя с ЭВМ | Пульт управления и перфокарты | Перфокарты и перфоленты | Алфавитно-цифровой терминал |

Продолжение таблицы

| Показатель | **Поколения ЭВМ** | | |
|---|---|---|---|
| | Четвёртое А 1976–1979 | Четвёртое Б 1985 – ? | Пятое |
| Элементная база процессора | Большие ИС (БИС) | Сверхбольшие ИС (СБИС) | +оптоэлектроника, +криоэлектроника |
| Элементная база ОЗУ | БИС | СБИС | СБИС |

| | | | |
|---|---|---|---|
| Максимальная ёмкость ОЗУ, байт | $10^5$ | $10^7$ | $10^8$ |
| Максимальное быстродействие процессора (оп\с) | $10^8$ | $10^9$ + много-процессор-ность | $10^{12}$ + многопро-цессорность |
| Языки программирования | +Новые процедур-ные ЯВУ | +Непроцедур-ные ЯВУ | +Новые н епроцедурные ЯВУ |
| Средства связи пользователя с ЭВМ | Монохромный графический дисплей, клавиатура | Цветной графический дисплей, клавиатура, мышь и др. | + Устройство голосовой связи с ЭВМ |

Учебное издание

МАМОНОВА Татьяна Егоровна

**И Н Ф О Р М А Т И К А**

**Программирование на С++**

Учебное пособие

**Издано в авторской редакции**

Научный редактор
*доктор технических наук,*
*профессор А.М. Малышенко*

Редактор

Верстка *Л.А. Егорова*

**Отпечатано в Издательстве ТПУ в полном соответствии с качеством предоставленного оригинал-макета**

Подписано к печати Формат 60×84/16.
Бумага «Снегурочка». Печать Xerox.
Усл. печ. л. . Уч.-изд. л. .
Заказ . Тираж экз.

Национальный исследовательский
Томский политехнический университет
Система менеджмента качества
Издательства Томского политехнического университета сертифицирована
NATIONAL QUALITY ASSURANCE по стандарту BS EN ISO 9001:2008

ИЗДАТЕЛЬСТВО ТПУ. 634050, г. Томск, пр. Ленина, 30.
Тел./факс: 8(3822)56-35-35, www.tpu.ru