

Основы программирования и баз данных

Методическое пособие

Седых Игорь Вячеславович

Оглавление

1. Введение	3
2. Области применения программирования	4
3. Ключевые понятия и определения.....	7
4. Виды и типы данных	14
5. Архитектура ЭВМ и принцип фон Неймана	18
6. Разработка программного обеспечения	21
7. Языки программирования	25
8. Структуры данных.....	30
9. Базы данных и их проектирование	36
10. Приложения	42
Приложение 1. Краткий обзор языка программирования Си.....	42
Приложение 2. Основные команды языка SQL, создание запросов.....	47
Приложение 3. Перевод чисел в разных системах счисления.....	48

1. Введение

Методическое пособие, которое Вы держите в руках, сопровождает курс «Основы программирования и баз данных» в Центре компьютерного обучения «Специалист».

Цель курса – максимально просто и интересно рассказать об основных понятиях, используемых в программировании, и заложить прочный фундамент для дальнейшего изучения современных ИТ-дисциплин. В этом пособии описываются все ключевые моменты курса, приводятся все необходимые формальные определения, предлагаются некоторые примеры, разбираемые в курсе, и иллюстрации к теоретическим конструкциям.

Пособие и курс разработаны на базе общего курса автора по программированию для студентов и школьников гуманитарных направлений различных учебных заведений. Изложение материала построено «от простого к сложному», начиная с самых общих понятий и заканчивая примерами несложных программ. Такое построение пособия позволяет дать полное представление об основах современных информационных технологий, не загружая читателя излишними подробностями и ненужными усложнениями.

Собранная в курсе информация – это объединение самых разных сведений из различных областей, таких как программирование, математическая логика, архитектура ЭВМ, базы данных, проектирование программного обеспечения и других. В этом заключается уникальность курса – окончив его, можно с уверенностью продолжать обучение по любому направлению, связанному с современными информационными технологиями. Данное пособие поможет систематизировать полученные на занятиях курса знания и закрепить их. Кроме того, оно является интересным и полезным справочным материалом, своего рода «первым учебником» для будущих ИТ-специалистов.

Само собой разумеется, что курс «Основы программирования и баз данных» является лишь первым этапом на пути будущего программиста к цели. Это своего рода предисловие к книге под названием «Мой путь программиста». Введением в нее будет начало изучения понравившегося Вам языка, главами – основная работа. Ну а этот курс – лишь рассказ о том, о чем будет Ваша книга. Об огромном и интересном мире информационных технологий.

2. Области применения программирования

Начало изучения любой дисциплины заключается в ответе на главный вопрос – с чем Вам придется иметь дело, когда Вы начнете в этой области работать. Вот и получается, что первый вопрос, на который нам придется найти ответ – что такое программирование? Чаще всего в нашей стране на этот вопрос дают неверный ответ, говоря, что программирование – это набор программ. Связано это с очень слабым распространением у нас узконаправленных ИТ-дисциплин и, естественно, нехваткой работающих в них специалистов. Набор программ обычно относится к дисциплине под названием «*кодинг*» или, более правильно, *кодирование*. На самом деле, кодирование является лишь частью программирования, наряду с анализом, проектированием, компиляцией, тестированием и отладкой, сопровождением.

Программирование – это решение задач при помощи ЭВМ, или процесс создания компьютерных программ. В общем случае, программирование решает вопросы программного управления различным оборудованием, будь то суперкомпьютер или программируемая стиральная машина. Таким образом, одним из важнейших этапов в выборе будущего приложения своих способностей для любого программиста будет определение круга задач. В первую очередь такой выбор начинается с области программирования. Для того чтобы разобраться в том, какие области программирования бывают, нужно обратиться к истории появления компьютеров.

Эра автоматизации расчетов началась в 1642 г., когда Блез Паскаль изобрел устройство, позволявшее механически выполнять сложение чисел, а в 1673 г. Готфрид Вильгельм Лейбниц сконструировал *арифмометр*, возможности которого были расширены до четырех основных арифметических действия (сложения, вычитания, умножения и деления). Эти устройства позднее получили широкое распространение, и, в частности, в Англии в XIX веке даже существовала профессия «*компьютер*» – человек, работающий с арифмометром. Основной задачей этих людей был расчет морских навигационных таблиц. Естественно, что точность таких расчетов была довольно низкой, а скорость работы невысокой. Работу этих людей нужно было автоматизировать, но сделать это без создания программируемого устройства было невозможно. Самым первым таким механизмом стала *разностная машина Бэббиджа*, созданная в 1822-м году (Рисунок 1). Машина приводилась в движение ручкой (по принципу арифмометра), но движение ее барабанов могло быть запрограммировано в зависимости от решаемой задачи. К сожалению, разработки Бэббиджа не дошли до успешного финала. Получив от Британского правительства огромные деньги на постройку полноценной расчетной машины (около 17 тысяч фунтов стерлингов – в то время этих денег хватило бы на по-

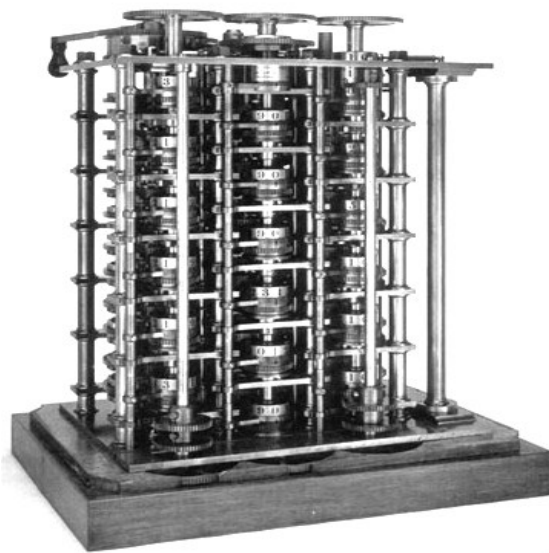


Рисунок 1 – Разностная машина

стройку трех крупных военных кораблей) он и представить себе не мог, с какими трудностями столкнется. Его проекту так и не суждено было завершиться. Однако когда к двухсотлетию со дня рождения Чарльза Бэббиджа на основе его оригинальных работ в лондонском Музее науки была собрана работающая копия *разностной машины № 2*, оказалось, что этот механизм мог проводить довольно сложные вычисления с гораздо более высокой точностью, чем ожидалось.

Следующее серьезное применение компьютеры получили во время Второй Мировой войны. Оно было связано со взломом фашистских шифров. Спецификацию компьютера разработали профессор Макс Ньюман и его коллеги; сборка *Colossus Mk I*, как называли эту машину, выполнялась в исследовательской лаборатории Почтового департамента Лондона под управлением инженера Томми Флауэрс и заняла 11 месяцев. Компьютер был построен на основе вакуумных ламп, что позволяет считать его первой *электронно-вычислительной машиной (ЭВМ)*. Создателем алгоритмов для этой машины был Алан Тьюринг – талантливый математик, в итоге заложивший основы многих современных компьютерных наук, один из группы ученых, взломавших код *Энигма*. Использовался Колосс для взлома кода, генерируемого шифровальной машиной *Lorenz SZ 40/42*, которая использовалась в переговорах высшего командования противника. К сожалению, проект был настолько секретным, что о его существовании не было ничего известно до конца XX-го века, из-за чего найти упоминания об этой машине крайне сложно. Утверждается, что ни один из десяти построенных для разных целей «Колоссов» не дожил до наших дней.

После войны компьютерные технологии, несмотря на всю секретность проектов, начали активно развиваться. Американский *ENIAC*, который часто называют

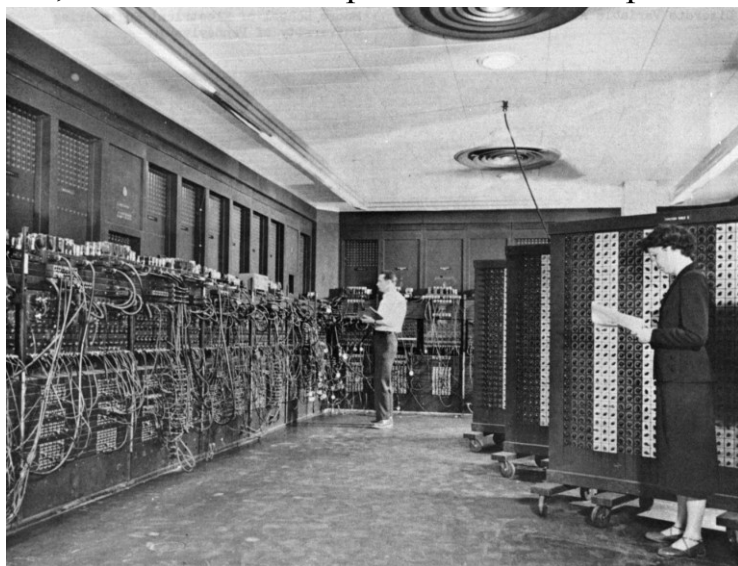


Рисунок 2 – ENIAC

первым электронным компьютером общего назначения, публично доказал применимость электроники для масштабных вычислений. Созданная под руководством Джона Моучли и Дж. Преспера Эккерта, эта машина была в 1000 раз быстрее, чем все другие машины того времени. Его основой послужили 18 000 вакуумных трубок, или электронных ламп (Рисунок 2), а первая программа была запущена в ноябре 1945 г., хотя официально об изобретении было сообщено на год позже, в начале 1946 г., когда проект рас-

секретили. В то время, многие исследователи были убеждены в том, что лампы будут сгорать очень часто, и «ЭНИАК» будет слишком много времени простаивать в ремонте, и потому будет практически бесполезен. Тем не менее, на реальной машине удавалось выполнять несколько тысяч операций в секунду в течение не-

скольких часов, до очередного сбоя из-за сгоревшей лампы. «Программа» для этой машины определялась состоянием соединительных кабелей и переключателей, что было серьезным прорывом со времени механических вычислительных устройств, но, конечно же, проигрывало машинам с хранимой программой, появившимся позже. В наше время принято считать «ЭНИАК» скорее калькулятором, нежели компьютером, но нельзя отрицать тот факт, что его изобретение стало ключевым моментом в разработке вычислительных машин, прежде всего из-за огромного прироста в скорости вычислений, но также и по причине появившихся возможностей для миниатюризации.

Успех первой ЭВМ означал начало новой эры в развитии вычислительных устройств. После «ЭНИАК» новые компьютеры стали появляться чаще и становились все быстрее. В процессе разработок первых ЭВМ к работе был привлечен известный американский математик Джон фон Нейман, который вскоре сумел сформулировать пять основных принципов функционирования универсальных вычислительных устройств, которые мы обсудим далее. На этих принципах, как на фундаменте, основывались в дальнейшем все последующие поколения машин. Особенно важен принцип программного управления. Именно из этого принципа вытекает определение программы в ее классическом понимании – как последовательного набора команд, выполняемых процессором. Отсюда возникает и необходимость в подготовке специально обученных людей, которые бы знали, какие команды и в какой последовательности нужно применять для решения определенной задачи. Это приводит к становлению программирования, как одного из видов деятельности человека.

За период с 1948 по 1975 года сменяются три так называемых *поколения* компьютеров. За это время происходит миниатюризация ЭВМ в процессе перехода от вакуумных ламп к транзисторам (изобретенным в 1947 году и внедренным в пятидесятые годы), а затем к интегральным схемам (1958 год) и изобретению микропроцессора (1971 год, микропроцессор *Intel 4004*). В это время наиболее известными



Рисунок 3 – Xerox Alto

ми производителями компьютеров были американские компании *IBM, Apple, Dell, Compaq*. Свои разработки велись и в СССР, но они, к сожалению, проигрывали зарубежным аналогам в производительности. Почти все компьютеры, производившиеся до 70-х годов XX века, имели очень простой текстовый *интерфейс*, который мы сейчас можем назвать терминалом. Это были обычные текстовые сообщения, выводимые компьютером на принтер или экран монитора. Управление осуществлялось с клавиатуры посредством ввода текстовых команд, что, естественно, очень усложняло работу с такими ЭВМ. Первым в мире компьютером, имевшим графический пользовательский интерфейс и операционную систему с «рабочим столом», стал *Xerox Alto* (Рисунок 3). Это удивительно, но именно он стал первым полностью *персональным*

компьютером в современном понимании этот термина. В то время как наиболее

известные в наши дни производители, вроде IBM, создавали сложные и громоздкие ЭВМ для инженерных расчетов, именно Xerox, почти неизвестная тогда компания, придумала дизайн ПК, который используется и сегодня. Xerox Alto даже имел манипулятор «мышь», без которого современный компьютер просто невозможно себе представить.

В январе 1984 года компания Apple выпустила первый серийный персональный компьютер с «мышью», реализовавший идею Xerox Alto в промышленном масштабе, названный *Apple Macintosh*. С этого времени работа с компьютером становится все проще и все доступнее самым разным людям. Естественно, что это требует создания все большего количества разнообразных программ, решающих при помощи ЭВМ самые разные задачи. Вот и получается, что началось применение компьютеров с решения математических задач, а закончилось почти неограниченными возможностями современной техники и программного обеспечения. Если подвести итог тому, как развивались компьютеры и программы, то можно выделить пять ключевых областей, в которых они применяются для решения задач в наши дни:

1. *«Математика»*. Имеется в виду написание функциональных частей программ; то, что заставляет приложение быть не просто картинкой на экране монитора, но еще выполнять различные математические действия, воспроизводить музыку и видео, сжимать и шифровать данные и т.д.
2. *Системное программирование*. В эту область можно отнести задачи создания операционных систем, написание интерфейсов для управления компьютерным «железом» – различными устройствами. Главная цель этой области задач – максимально упростить работу человека с компьютером и выполнять различные технические операции над ЭВМ.
3. *Прикладное программирование*. Самая большая область задач – создание программ, решающих повседневные задачи – текстовых и графических редакторов, мультимедийных приложений, словарей и баз данных, а также многого другого.
4. *WEB-программирование*. Написание программного обеспечения для сети Интернет и создание сайтов.
5. *Скриптовые языки высокого уровня*. Написание программ, предназначенных для автоматизации и управления работой существующих программ, написания дополнительных модулей в программных пакетах.

3. Ключевые понятия и определения

После того, как мы выяснили, что программирование – это решение задач с помощью ЭВМ, то и начнем мы с определения *задачи*. Само по себе определение термина «задача» – более философское, нежели конкретное понятие. Чтобы не углубляться в такие изыскания, мы остановимся не на самом определении, а на том,

что такое решение задачи применительно к программированию. И самое главное – это, конечно же, из каких этапов оно состоит.

Первым этапом решения является *постановка задачи*. Постановка задачи может быть устной, может быть письменной – это не столь принципиально – главное то, что на этом этапе должно быть четко определено, что дано, и что требуется найти. Так, если задача конкретная, то под постановкой задачи понимают ответ на два вопроса: какие исходные данные известны и что требуется определить. Если задача обобщенная, то при постановке задачи понадобится еще ответ на третий вопрос: какие данные допустимы. В итоге постановка задачи обычно включает в себя формулировку условия, определение конечных целей решения, определение формы выдачи результатов, описание используемых и конечных данных. Постановка задачи определяет требования, которые предъявляются к конечному программному продукту.

В связи с тем, что постановка задачи может быть некорректной или вообще ошибочной, обязательным является второй этап – *анализ задачи*. Этот этап позволяет определить, можно ли в принципе решить задачу в указанных при постановке рамках. Кроме того, на этом этапе Вы также можете определить, какие пути решения выбрать, с какими данными и как работать, какой язык программирования использовать и т.д. На втором этапе обязательным является формальная постановка задачи или техническое задание, предназначенное для описания жестких условий, в которых решение будет происходить.

Третьим этапом является *проектирование*. Это важнейший этап в решении любой задачи. Именно во время этого этапа происходит детальная разработка всех основных элементов будущей программы, включая все модули, интерфейс, функциональные решения и прочее. Этот этап занимает большую часть времени в решении любой задачи. По качественно созданному проекту написать программу легко и что самое главное – программирование по готовому проекту практически механическая деятельность, а значит, этот процесс можно контролировать при помощи нормативов и норм, ускоряя процесс создания программного обеспечения (ПО).

Только после проектирования приступают к *реализации*. Этот этап, как правило, занимает минимум времени в решении задачи. Его основная цель – непосредственная разработка программы, решающей поставленную задачу. Именно в этот момент кодеры набирают программу, работая вместе с проектировщиками и тестировщиками, создавая готовый продукт.

Заключительным этапом решения задачи является *модификация и обновление* созданного ПО. Этот этап нужен для того, чтобы Ваш продукт всегда был современным и востребованным на рынке.

Итак, самым важным этапом в этом процессе является проектирование. Одно из наиболее часто встречающихся понятий, связанных с проектированием задач любого класса, это *алгоритм*. Это понятие является фундаментальным в информатике, его происхождение связано с математикой и именем персидского учёного Абу

Абдуллаха Мухаммеда ибн Муса аль-Хорезми (787-850). В своей книге «Об индийском счете» он сформулировал правила записи натуральных чисел с помощью арабских цифр и правила действий над ними столбиком. Эта его книга, будучи переведенной на латинский язык получила название *Algoritmi de numero Indorum* («Алгоритмы о счёте индийском»). У самого понятия «алгоритм» несколько определений. Алгоритм можно определять неформально, в применении к программированию, математике и даже для обычной жизни. В конечном итоге, алгоритм – это формально описанная последовательность действий, которые необходимо выполнить для получения требуемого результата. В классическом понимании компьютерные алгоритмы могут содержать только три вида последовательностей команд. Линейная структура команд, разветвленная структура и циклическая. Все команды, описанные в алгоритме, выполняются всегда только в описанном порядке без каких-либо отклонений.

Линейная структура процесса вычислений предполагает, что для получения результата необходимо выполнить некоторые операции в определенной последовательности. Например, для определения площади треугольника по формуле Герона необходимо сначала определить полупериметр треугольника, а затем по формуле его площадь.


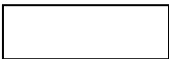
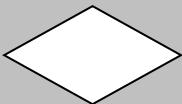


Разветвленная структура процесса вычислений предполагает, что конкретная последовательность операций зависит от значений одного или нескольких параметров. Например, если дискриминант квадратного уравнения не отрицателен, то уравнение имеет два корня, а если отрицателен, то действительных корней нет.

Циклическая структура процесса вычислений предполагает, что для получения результата некоторые действия необходимо выполнить несколько раз. Например, для того, чтобы получить таблицу значений функции на заданном интервале изменения аргумента с заданным шагом, необходимо соответствующее количество раз определить следующее значение аргумента и посчитать для него значение функции.

Существует огромное число различных видов алгоритмов. Особую роль выполняют *прикладные алгоритмы*, предназначенные для решения определённых прикладных задач. Важную роль играют *рекурсивные алгоритмы* (алгоритмы, вызывающие сами себя до тех пор, пока не будет достигнуто некоторое условие возвращения). Начиная с конца XX – начала XXI века активно разрабатываются *параллельные алгоритмы*, предназначенные для вычислительных машин, способных выполнять несколько операций одновременно. К сожалению, существуют такие задачи, создание алгоритма для которых невозможно. Такие задачи обычно называют *алгоритмически неразрешимыми*.

Обычно алгоритмы записывают в виде текста – нечто вроде плана действий. Но этот способ не всегда удобен и понятен. Поэтому очень часто используют более наглядные графически нотации для изображения алгоритмов. Наиболее популярным и несложным для изучения способом такого изображения алгоритмов являются *блок-схемы*. Этот способ довольно старый, но по-прежнему очень часто исполь-

зуется, особенно в обучении программированию. В блок-схемах приняты специальные обозначения, укажу некоторые из них:

Наименование	Обозначение	Функция
Блок начало-конец		Элемент отображает вход из внешней среды или выход из нее (наиболее частое применение – начало и конец программы).
Блок вычислений		Выполнение одной или нескольких операций, обработка данных любого вида (изменение значения данных, формы представления, расположения).
Логический блок		Отображает участок ветвления алгоритма с одним входом и, как правило, двумя альтернативными выходами, из которых только один может быть выбран после вычисления условий, определенных внутри этого элемента.
Предопределенный процесс		Символ отображает выполнение процесса, состоящего из одной или нескольких операций, который определен в другом месте программы. Например – вызов процедуры или функции.
Данные (ввод-вывод)		Преобразование данных в форму, пригодную для обработки (ввод) или отображения результатов обработки (вывод).

Используя указанные обозначения, мы можем наглядно показать, как выглядят все три классических процесса в алгоритмах:

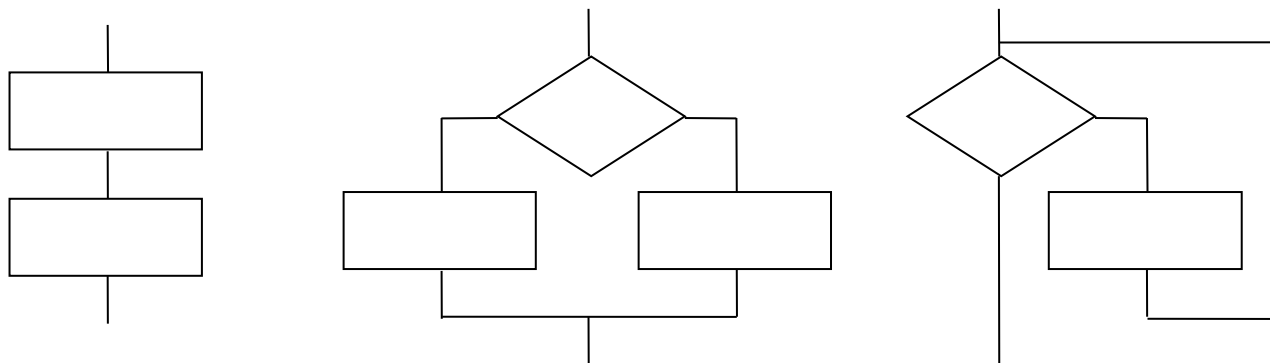


Схема 1

Схема 2

Схема 3

Рисунок 4 – Примеры блок-схем

В приведенных на Рисунок 4 примерах легко узнаются линейная структура (схема 1), ветвление (схема 2) и цикл (схема 3). Особо жестких требований к изображению элементов блок-схем нет. Однако при их создании стараются придерживаться общепринятых обозначений. Стрелочки между действиями в блок-схемах рисовать не обязательно, т.к. блок-схемы обычно читают сверху вниз.

К сожалению, в современном программировании блок-схемы не всегда можно использовать из-за сложности современных языков программирования. Блок-схемы можно использовать только для структурных языков программирования или

же структурных участков кода. Понятие структурного кода мы обсудим позже. Чаще всего в современном программировании используют *псевдокод*.

Понятие псевдокода связано с идеей создания универсального языка для общения программистов, пишущих на разных языках. Универсального переводчика с одного языка на другой не существует, а работать таким людям над одним проектом вместе бывает нужно очень часто. Вот и появляется компактный (зачастую неформальный) язык описания алгоритмов, использующий ключевые слова «выдуманных» языков программирования. Псевдокод не может быть полноценным универсальным языком, т.к. он обычно опускает некоторые несущественные для описания алгоритма детали и специфический синтаксис тех или иных языков программирования, программировать на таком языке нельзя. Псевдокод широко используется в учебниках и научно-технических публикациях, а также на начальных стадиях разработки компьютерных программ. В итоге написания такого кода программист получает промежуточную форму между повседневным языком и языком программирования, что позволяет упростить понимание алгоритма и облегчить процесс дальнейшего переноса алгоритма на машину. Наиболее часто псевдокод пишут очень похожим на язык *Pascal*, что, в общем-то, не принципиально и синтаксис псевдокода может быть любым. О таком синтаксисе программисты договариваются сами друг между другом, и он может быть написан на любом языке – русском, английском, итальянском и т.д. Единого, «международного», псевдокода не существует, потому что языки программирования слишком сильно отличаются друг от друга.

Вот мы и подошли к, пожалуй, ключевому термину в нашем курсе. К понятию программы. Начинается любая программа с постановки задачи и разработки алгоритма, проектирования решения этой задачи. А затем начинается программирование. Но что такое программа, из каких частей она состоит?

Программа – это готовое решение задачи для ЭВМ. У любой современной программы есть несколько принципиальных компонентов. Прежде всего, это *функциональное наполнение программы* – «математика» – то, без чего любая программа будет просто красивой картинкой на экране монитора без каких-либо реакций на действия пользователя. Иногда эта часть программы даже выносится в виде одного или нескольких отдельных, самостоятельных файлов. Вторая часть программы – это ее *интерфейс*. Многие современные программы рассчитаны на обычных пользователей и поэтому должны обладать удобным и простым способом управления ими. Эти возможности пользователям и дает интерфейс. Обычно считается что интерфейс – это как раз та самая красивая картинка с кнопками, редакторами и прочими графическими атрибутами современных операционных систем. Но интерфейс не обязан быть графическим. Он также может быть текстовым. Программы с таким интерфейсом обычно вызываются из *командной строки* (более правильно – *консоли* или *терминала*) продвинутыми пользователями и администраторами. И третья часть программы – то, что отвечает за ее функционирование и общее устройство – *архитектура программного обеспечения*.

Довольно важный вопрос в определении компонентов программы – будут ли внешние ресурсы, без которых программа не работает, также ее частью? Например, библиотеки *DirectX*, без которых не запускается ни одна современная компьютерная игра для компьютеров под управлением *Microsoft Windows*. С одной стороны программисты игр не пишут этой библиотеки, а лишь используют ее, с другой стороны, без нее игра не запустится. Вот и получается, что с одной стороны это вроде бы часть Вашей программы, а вроде бы и нет. Давайте считать, что если некий внешний ресурс для своей программы Вы пишете самостоятельно, то он – часть Вашей программы. А если вы только используете его – то его наличие просто становится обязательным условием ее работы.

Опишем теперь этапы создания программы. В общем случае любая программа начинает существование в виде *исходного кода*. Очень часто исходный код называют программой. Это неаккуратно. Поэтому мы будем придерживаться правильной терминологии и называть первый этап исходным кодом или «*исходником*». Исходный код программы – это обычный текстовый документ, который можно набрать в любом текстовом редакторе, но делать это гораздо удобнее можно в специальной *среде языка программирования*, которая позволяет не только писать программы, но и упрощает поиск ошибок и часто ускоряет процесс написания кода. Исходный код представляет собою перевод алгоритма (записанного в любом виде – блок-схема, псевдокод и т.п.) с повседневного языка в термины формального языка программирования. Полученный текст исполняться компьютером не может. Машина просто не поймет, что в файле написано. Для того чтобы программа смогла заработать, ее «исходник» нужно перевести в понятный машине язык. Он называется *машинным кодом* или *программой*. Для этого действия нужен *компилятор* – важнейшая часть любого языка программирования, переводящая исходный код в машинный. Если в процессе создания Вы используете компилятор, то это означает, что Ваш язык программирования *компилируемый*. В процессе компиляции могут возникнуть ошибки, называемые обычно *ошибками компиляции* или *синтаксическими ошибками*. Эти ошибки совсем не страшные – обычно их очень легко исправить, особенно, если учесть тот факт, что такие ошибки особо отмечаются компилятором (вплоть до строки, где такая ошибка произошла). Гораздо страшнее *логические ошибки*, эффект которых проявляется в программах, которые работают, но либо дают неверный результат, либо сбоят в процессе решения задачи. Искать такие ошибки очень сложно и в их поиске обычно сильно помогает использование *систем отладки* или *debug* сред языков программирования.



Рисунок 6 – Схема компиляции

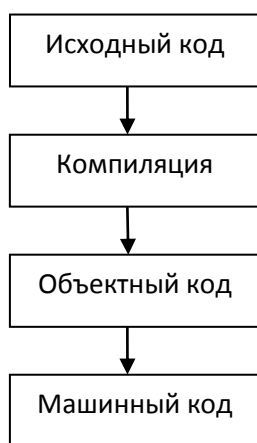


Рисунок 5 – Объектный код

Получается, что в классических языках программирования можно нарисовать условную схему, показывающую шаги, которые проходит программа при создании (см. Рисунок 6). Но такая схема не един-

ственная. Существуют более простая ситуация и более сложная. Сначала опишем более сложную схему. В ней появляется дополнительный шаг, называемый *объектным кодом* (см. Рисунок 5). Объектный код, часто называемый *объектным модулем* или *объектным файлом*, это особым образом преобразованный компилятором исходный код программы. Фактически он является промежуточным представлением исходного кода программы, содержащий в себе особым образом подготовленный код, который может быть объединён с другими объектными файлами при помощи редактора связей (*компоновщика* или *линковщика*, если опираться на зарубежную терминологию) для получения готового машинного кода. Объектный код объединяет готовый универсальный машинный код и данные, созданные программистом, что позволяет собирать готовую программу даже из исходных кодов, написанных на разных языках программирования. Частично, именно благодаря объектному коду (а если быть более точным, то *библиотекам* объектных кодов) мы можем писать программы на разных языках и использовать в своей работе коды друг друга. Немаловажно также и то, что объектный код позволяет упростить и ускорить процесс написания сложных программ. Некоторые компиляторы языков программирования создают объектный код автоматически, для некоторых компиляторов эта возможность компилятора включается отдельно. Никаких специальных программ для создания объектного кода не требуется.

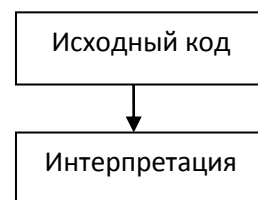


Рисунок 7 - Интерпретация

Некоторые языки программирования, например скриптовые, являются *интерпретируемыми*. Это и есть та самая простая ситуация создания программы, о которой шла речь выше. Программы, написанные на таких языках нельзя запустить без *интерпретатора* – программы, которая и выполняет написанный Вами код (см. Рисунок 7). Получается, что такие программы сильно зависят от установленного на компьютере программного обеспечения. Например, программы, написанные на языке 1С, работают только в системе 1С; для того, чтобы увидеть не текст на языке html, а красивый сайт, требуется интернет-браузер и т.д. Интерпретируемые языки программирования – одни из самых простых для изучения. Конечно же, не из-за того, что эти языки программирования простые, а из-за того, что количество доступных команд в них ограничено возможностями интерпретатора. Среди классических языков программирования также были интерпретируемые языки программирования – это классический Бейсик.

В конце главы хотелось бы сформулировать ряд важных и полезных для изучения информационных технологий терминов и понятий. Прежде всего, определим понятие *информации*. Информация – скорее философское понятие, обозначающее сведения о чем-либо, независимо от формы их представления. С точки зрения компьютера информацией являются все данные, которые попадают в его память. На основе этого понятия можно определить термин *логическое выражение*. Логическим выражением является любая информация, значение которой в каждый момент времени может быть однозначно определено как истинное или ложное. И еще несколько классических определений:

1. *Операнд* – аргумент операции, данное, которое обрабатывается командой. Может быть достаточно сложным, например комбинацией других операндов или функцией.
2. В большинстве языков программирования *низкого* и *высокого* уровня (эти термины мы сформулируем в 7-ой главе) определён набор встроенных *операций отношения*, позволяющих строить «простые» логические выражения. А также *арифметические операции* над операндами для выполнения четырех базовых математических действий: сложения, вычитания, умножения и деления.
3. *Идентификатором* называется последовательность цифр и букв, а также специальных символов, при условии, что первой стоит буква или специальный символ. В программировании идентификаторы (или имена) используются для обращения к специальным объектам внутри программы.
4. *Переменная* в программировании это именованное имя памяти. У переменной в любом языке программирования всегда есть три части: идентификатор, которым пользуется программист; место в оперативной памяти, которое будет заполняться компьютером в процессе выполнения программы; адрес, позволяющий обращаться напрямую к памяти в программе.
5. *Константа* в программировании – способ адресования данных, изменение которых программой запрещено. Константы бывают разного вида – это бывают неизменяемые переменные, а также обычные статичные объекты – например, символьные или числовые константы.

4. Виды и типы данных

Рассказ о типах данных и способах их представления на компьютере нужно начать с небольшого уточнения. Любая информация, которая в том или ином виде попадает в компьютер, сохраняется в нем в виде специальных наборов цифровой информации, кодируемой в двоичной форме числа. Таким образом, в конце концов, все данные, с которыми вы работаете, на компьютере преобразуются в числа и *числовой формат* данных – основной способ хранения информации в компьютере. Другими важными типами данных являются *числовой* и *логический (булевский)*.

Внутри современного компьютера нет более сложных устройств, чем наборы огромного количества выключателей, которые могут находиться в двух состояниях: включено и выключено. В этом смысле современный компьютер ничем не отличается от устройств Бэббиджа или от машины ЭНИАК. Хотя современные ЭВМ гораздо сложнее с технической точки зрения и могут выполнять миллиарды операций в секунду – их принципиальное устройство остается таким же, как и 10, 20 и даже 30 лет назад. Повышение мощности современно компьютера достигается за счет повышения количества операций в секунду. К сожалению, до бесконечности таким способом повышать мощность нельзя. Это соображение приводит к появлению параллельных алгоритмов, вероятностных алгоритмов, нейронных сетей и т.д.

Основной вид информации в ЭВМ, таким образом, это числа или сигналы 1 и 0. Для операций над такими объектами чаще всего прибегают к понятиям *математической логики* – науки, которая занимается изучением логических выражений. Поэтому давайте довольно подробно остановимся на том, что такое логика, и как устроены логические операции. Основные правила таких действий определяет алгебра логики – раздел математики, в котором изучаются логические операции над *высказываниями* (или, как мы их называем, логическими выражениями). Высказывания, как и логические выражения, могут быть истинными, ложными или содержащими истину и ложь в разных соотношениях, причем значение высказывания в любой момент времени может быть однозначно вычислено.

Основных операций математической (и программной) логики, с которыми нужно познакомиться в начале изучения программирования, немного. Их три. Давайте этим операциям определения и приведем примеры их использования.

1. *Отрицание* – смысл этой операции в изменении значения высказывания на противоположное. Например: построение цикла «работать до тех пор, пока *не* достигнут конец файла».
2. *Конъюнкция* – также называется логическим "И", логическим умножением, или просто "И". Операция, применяемая к двум высказываниям и имеющая истинное значение в том и только в том случае, когда оба высказывания истинны. Например: точка на числовой прямой лежит внутри отрезка, когда ее координата меньше правой границы отрезка *и* больше левой границы.
3. *Дизъюнкция* – также называется логическим "ИЛИ", логическим сложением, или просто "ИЛИ". Операция, применяемая к двум высказываниям и имеющая ложное значение в том и только в том случае, когда оба высказывания ложны. Например: если число не делится на 2 *или* на 3, то оно не делится на 6.

В компьютерной логике ключевыми константами являются логический ноль (ложь) и логическая единица (истина). Наиболее формально эти операции можно выписать в виде следующих таблиц:

Отрицание		Конъюнкция			Дизъюнкция		
		А	В	А И В	А	В	А ИЛИ В
А	НЕ А	1	1	1	1	1	1
1	0	1	0	0	1	0	1
0	1	0	1	0	0	1	1
		0	0	0	0	0	0

Описанных логических операций, несмотря на их внешнюю простоту, достаточно для создания практически любых логических выражений в программировании. Они могут быть очень сложными, но такое построение всегда возможно.

Для хранения любых типов данных в компьютере используется *двоичная система счисления*, кодирующая любую информацию в виде 1 и 0. Таким образом, в конечном итоге, компьютер работает с данными исключительно в виде двоичной их записи. Более того, десятичная система счисления на компьютере не используется вообще. Из используемых в компьютере систем счисления для удобства программирования стоит также выделить *восьмеричную* и *шестнадцатеричную* системы счисления. Мы не будем подробно сейчас останавливаться на переводе чисел из одной системы счисления в другую, т.к. это сухая и скучная информация. Отметим лишь то, что компьютер, в конечном итоге, работает именно в двоичной системе счисления и переводит все вводимые Вами числа сразу, не дожидаясь сохранения информации в памяти.

1. Чем меньше значений существует в системе, тем проще изготовить отдельные элементы, оперирующие этими значениями. В частности, две цифры двоичной системы счисления могут быть легко представлены многими физическими явлениями: есть ток – нет тока, индукция магнитного поля больше пороговой величины или нет и т. д.
2. Чем меньше состояний у технического элемента, тем выше помехоустойчивость и тем быстрее он может работать. Например, чтобы закодировать три состояния через величину индукции магнитного поля, потребуется ввести два пороговых значения, что не будет способствовать помехоустойчивости и надёжности хранения информации.
3. Двоичная арифметика является довольно простой. Простыми являются таблицы сложения и умножения – основных действий над числами.

Теперь давайте рассмотрим два вида наиболее важных и часто используемых в программировании типов данных – *чисел* и *символов*.

Память ЭВМ построена из запоминающих элементов, обладающих двумя устойчивыми состояниями, одно из которых соответствует нулю, а другое – единице. Таким физическим элементом представляется в памяти ЭВМ каждый разряд двоичного числа (бит). Совокупность определенного количества этих элементов служит для представления многоразрядных двоичных чисел и составляет разрядную сетку ЭВМ. В любой ЭВМ биты организованы в байты – каждый байт является объединением 8 бит, каждый байт пронумерован. Номер байта называется его адресом.

Самым важным (и, одновременно, простым) типом данных на компьютере является, конечно, *числовой формат данных*. Для понимания того, как числа хранятся в компьютере, необходимо отметить, что любая информация на машине, в конечном итоге, кодируется в двоичной системе счисления. Таким образом, числа на компьютере также хранятся в своей двоичной записи – как целые, так и дробные. Организация хранения целых чисел особого труда не представляет – такие числа просто переводят из десятичной системы счисления в двоичную и сразу записывают в память. Для компьютерного представления целых чисел обычно используется несколько различных типов данных, отличающихся друг от друга коли-

чеством разрядов. Чаще всего используется восьми-, шестнадцати- и тридцати-двухразрядное представление чисел (один, два или четыре байта соответственно).

Для целых чисел существуют два представления: беззнаковое (только для неотрицательных целых чисел) и со знаком. Очевидно, что отрицательные числа можно представлять только в знаковом виде. Различие в представлении целых чисел со знаком и без знака вызвано тем, что в ячейках одного и того же размера в беззнаковом типе можно представить больше различных положительных чисел, чем в знаковом. В ЭВМ в целях упрощения выполнения арифметических операций применяют специальные коды для представления чисел. Использование кодов позволяет свести операцию вычитания чисел к операции поразрядного сложения кодов этих чисел. В записи целых чисел используют прямой и дополнительный коды, которые позволяют записывать любые целые числа в нужном формате. Диапазон представления целых чисел в ЭВМ зависит от разрядности остановленной операционной системы.

Гораздо сложнее обстоят дела с вещественными числами. Их нужно хранить специально – т.к. базово компьютер поддерживает работу только с целочисленными данными. Это означает тем самым, что такие числа придется обработать перед сохранением в памяти в особый формат. Современных форматов хранения вещественных чисел два – число с фиксированной и с плавающей точкой.

Формат чисел с *фиксированной точкой* наиболее простой. Его реализация подразумевает, что целая и дробная части одного вещественного числа хранятся отдельно как целые числа в памяти компьютера. Однако этот метод, несмотря на явное удобство использования, наглядность и простоту обработки данных, является не очень удачным. Недостатком представления чисел в формате с фиксированной запятой является небольшой диапазон представления величин, недостаточный для решения математических, физических, экономических и других задач, в которых используются как очень малые, так и очень большие числа. Немногие языки программирования предоставляют встроенную поддержку чисел с фиксированной запятой, поскольку для большинства применений двоичное или десятичное представление чисел с *плавающей точкой* проще и достаточно точно. Числа с плавающей точкой из-за их большего динамического диапазона устроены гораздо удобнее, и, что немаловажно в сложных расчетах, для них не нужно предварительно задавать количество цифр после запятой. В этом формате положение точки в записи числа может изменяться в зависимости от выбранного представления числа. Название *плавающая точка* происходит от того, что точка в позиционном представлении числа может быть помещена где угодно относительно цифр в строке, это положение запятой указывается отдельно во внутреннем представлении числа. Таким образом, представление числа в форме с плавающей запятой может рассматриваться как компьютерная реализация экспоненциальной записи чисел, в которой может быть записано любое число. Реализация математических операций с числами с плавающей запятой в вычислительных системах может быть как аппаратная, так и программная.

Рассмотрим, как выглядит произвольное число A в таком виде:

$$A = m \times q^p$$

Здесь m – *мантисса* числа, q – основание системы счисления (для компьютера равно двум) и p – *степень* числа. Также в этом формате требуется указать *знак мантиссы* – отвечающий за знак самого числа A . В современных операционных системах диапазон хранения в формате с плавающей точкой составляет от 10^{-308} степени до 10^{308} степени. И это не учитывая специальных форматов увеличенной точности.

Другим, не менее важным форматом данных является *символьный тип данных*. На самом деле появившийся как обычный целочисленный формат. Дело в том, что в программировании изначально символов не было. Они просто не были нужны. Но с появлением мониторов, позволяющих отображать результаты работы программ, а также операционные системы и символы псевдографики, появилась необходимость и в символах. Переменные символьного типа, как правило, могут хранить только одиночный символ. В памяти такая переменная занимает 1 байт, соответственно может принимать 256 различных значений (на самом деле в памяти хранятся лишь коды символов – которые соответствуют инструкциям как именно изображать символы на экране). Зависимость между кодами символов и самими символами устанавливается операционной системой. Однако традиционно первые 128 из них – это так называемые ASCII-символы. Первые 32 символа называются управляющими (например, признак конца строки, подача звукового сигнала, перевод курсора на новую строку), остальные изображаемыми. К сожалению, стандарт ASCII не стал универсальным, и в современном мире для стандартизации символов была создана специальная таблица – *UNICODE (Юникод)*. При помощи этого стандарта появилась возможность записать в универсальном виде практически любые возможные символы. Более того, Юникод может расширяться без потери универсальности почти безгранично. Достигается это за счет особого вида этой таблицы символов. По мере изменения и пополнения таблицы символов системы Юникода и выхода новых версий этой системы (эта работа ведётся постоянно), выходят и новые стандарты. Актуальная на 2012 год версия имеет номер 6.1. Стандарт и устройство Юникода позволяет работать с более чем 2 млрд. символов и, несмотря на то, что принято использовать лишь 1 112 064 символов для совместимости с кодировкой UTF-16, этого более чем достаточно. В настоящее время используется «все» около 110 тысяч символов.

5. Архитектура ЭВМ и принцип фон Неймана

Продолжая рассказ о типах данных, очень важно отметить, почему информация, хранящаяся в компьютере, может быть систематизирована и отнесена к различным видам данных. Почему современные компьютеры на программном уровне устроены похожим образом? За счет чего компьютеры хранят информацию одинаково? Почему текстовые документы переносятся с компьютера на компьютер одинаково? Что если компьютер сам обрабатывает информацию и кодирует ее по-своему? Переносимость информации на разные компьютеры, одинаковая интер-

претация ее в программах возможна благодаря принципам Джона фон Неймана. В соответствии с его принципами создаются современные компьютеры и программы, именно благодаря им современные компьютеры на программном уровне устроены одинаково.

1. *Принцип двоичного кодирования.* Согласно этому принципу, вся информация, поступающая в ЭВМ, кодируется с помощью двоичных сигналов (двоичных цифр, битов) и разделяется на единицы, называемые словами.
2. *Принцип однородности памяти.* Программы и данные хранятся в одной и той же памяти, поэтому компьютер не различает, что хранится в данной ячейке памяти – число, текст или команда. То есть, фактически, с точки зрения компьютера, информация не имеет смысла. За обработку информации отвечают программы.
3. *Принцип адресуемости памяти.* Структурно основная память состоит из пронумерованных ячеек. Процессору в произвольный момент времени доступна любая ячейка. Отсюда следует возможность давать имена областям памяти так, чтобы к запомненным в них значениям можно было впоследствии обращаться или менять их в процессе выполнения программ с использованием присвоенных имен.
4. *Принцип программного управления.* Программа состоит из набора команд, выполняющихся процессором автоматически в определенной последовательности, однозначно определяемой программой.
5. *Принцип жесткости архитектуры.* Первой выполняется команда, заданная пусковым адресом программы. Обычно это адрес первой команды программы. Адрес следующей команды однозначно определяется в процессе выполнения текущей команды и может быть либо адресом следующей по порядку команды, либо адресом любой другой команды. Процесс вычислений продолжается до тех пор, пока не будет выполнена команда, предписывающая прекращение вычислений.

Компьютеры, построенные на перечисленных принципах, относятся к типу фон-неймановских. Почти все современные компьютеры относятся именно к этому типу компьютеров. Существуют и другие машины, не удовлетворяющие некоторым принципам фон Неймана, но они являются специальными и в обычной жизни не встречаются.

Конечно, любому программисту также важно знать, как именно устроен компьютер «изнутри». Не только для того, чтобы знать его основные компоненты, но также и для того, чтобы представлять, с чем именно ему приходится работать. Архитектура современного компьютера также была разработана фон Нейманом. В 1946 году вместе с Г. Гольдстейном и А. Берксом он написал и выпустил отчет "Предварительное обсуждение логической конструкции электронной вычислительной машины". Поскольку имя фон Неймана как выдающегося физика и математика

было уже хорошо известно в широких научных кругах, все высказанные положения в отчете приписывались ему. Более того, архитектура первых двух поколений ЭВМ с последовательным выполнением команд, на основе которой строятся и современные компьютеры, получила название "фон-неймановской архитектуры ЭВМ" (см. Рисунок 8).

В данной схеме отмечен только один вид памяти – оперативной. На самом деле, конечно, в современных компьютерах видов используемой памяти гораздо больше. Всю память в компьютере можно разделить на две большие группы – *ОЗУ* (оперативно запоминающие устройства) и *ПЗУ* (постоянно запоминающие устройства). Чаще всего программист работает с *оперативной памятью* и с информацией на *внешних носителях – файлами*.

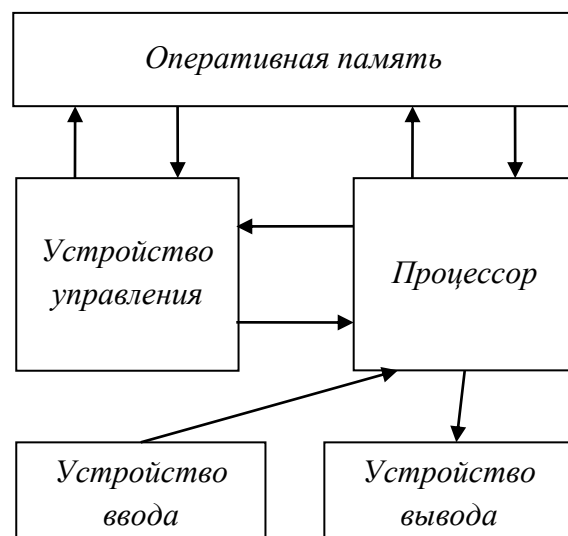


Рисунок 8 – Архитектура ЭВМ

Как правило, вся информация на компьютере хранится следующим образом: в ОЗУ хранятся те данные, с которыми в данный момент работает компьютер. В ПЗУ – то, что нужно сохранить надолго. Информация, хранящаяся в ОЗУ компьютера, уничтожается при отключении питания от такой памяти. Самое простое разделение такой памяти может быть на две категории – *кэш процессора* и *оперативную память*. Ряд моделей центральных процессоров обладают собственным кэшем, для того чтобы минимизировать доступ к оперативной памяти, которая медленнее, чем регистры. Кэш-память может давать значительный выигрыш в производительности в случае, когда тактовая частота оперативной памяти значительно меньше тактовой частоты процессора. Тактовая частота для кэш-памяти обычно почти совпадает с частотой процессора. Кэш центрального процессора разделён на несколько уровней, в универсальном процессоре в настоящее время число уровней может достигать трех. Самой быстрой памятью является *кэш первого уровня*, являющийся неотъемлемой частью процессора, поскольку эта память расположена на одном с ним кристалле и входит в состав функциональных блоков. Объём обычно невелик — не более 384 Кбайт. Вторым по быстродействию является *кэш второго уровня*, обычно он тоже расположен на кристалле и его объём может колебаться от 128 Кбайт до 1–12 Мбайт для современных процессоров. Кэш третьего уровня наименее быстродействующий, но он может быть очень внушительного размера – более 24 Мбайт, причем, несмотря на его низкое быстродействие, по сравнению с другими уровнями кэш-памяти, он все равно быстрее оперативной памяти. Иногда существует и четвертый уровень кэша, который обычно расположен на отдельной микросхеме, такой кэш появляется, как правило, в высокопроизводительных серверах и суперкомпьютерах.

Оперативная память – это основной вид ОЗУ в компьютере. В ней временно хранятся данные и команды, необходимые программам, запущенным на компьютере, а также процессору, для выполнения предписанных операций. Основное требование к оперативной памяти – достаточно высокое быстродействие и большой объем хранимой информации. Содержащиеся в оперативной памяти данные доступны только тогда, когда на модули памяти подаётся напряжение. Прекращение подачи питания на модули памяти, даже кратковременное, приводит к искажению либо полной потере содержимого памяти.

Основным видом ПЗУ является, конечно же, *магнитный диск*. На основе этой технологии долгое время строились такие носители информации как *дискеты*, *жесткие диски* и т.д. Более современными носителями информации являются *оптические диски*, а также *флэш-память*. Все эти виды носителей информации позволяют хранить данные, даже когда питание от них отключено.

6. Разработка программного обеспечения

Вопросы проектирования программного обеспечения – одни из самых важных вопросов в программировании. Проектирование позволяет существенно сократить время разработки ПО, уменьшить число ошибок. Но – обо всем по порядку.

Прежде всего, нужно отдельно отметить цели и задачи, которые ставятся перед программированием в каждой области его применения. Назначение программирования во всех областях на самом деле одинаково: упростить и облегчить работу специалистов, разработать новые инструменты для повышения эффективности решения задач. Цели программирования – разные. В математике – это создание как можно более точного и быстро работающего кода. В системном программировании – написание компактных, умело работающих с аппаратными ресурсами машины, операционных систем и других системных оболочек. В прикладном программировании – сделать так, чтобы программы выполняли как можно больше задач и были легкими в применении. И так далее. В любой области программирования можно с легкостью выделить те требования, что предъявляются к программам.

Вне зависимости от целей и назначения программирования, разработка программного обеспечения, по большому счету, идет всегда в соответствии с некими общими правилами и этапами создания программ. Формально об этом нам могут рассказать этапы разработки и версии ПО. Начнем с описания числового формата версии программы. На самом деле для отслеживания изменений в программах и сообщении пользователям о том, с какой программой они сейчас работают (более новой, или уже устаревшей, например) было создано большое количество схем присвоения номеров версиям программного обеспечения. Версии пробовали нумеровать целыми числами, что оказалось неудобно из-за невозможности разделить важные и не очень важные изменения. Иногда используют буквенные обозначения, текущие даты, Дональд Кнут нумерует версии системы компьютерной вёрстки *TEX* последовательными приближениями числа π , но это уже, конечно, экзотика. Наиболее распространенный и современный способ нумерации – это использова-

ние специального формата, в котором легко отражаются все изменения в программе, как важные и заметные пользователю, так и «внутренние» – интересные больше программистам:

1.2.3.4

В этом обозначении у каждого из четырех чисел есть свой смысл, определяющий значимость перемен между стадиями разработки ПО. В данном случае приняты такие правила: первое число (обозначенное цифрой 1) может быть изменено только тогда, когда код программы практически полностью изменен, когда программа претерпела существенные изменения, заметные пользователю. Мы можем назвать это число *основной официальной версией* продукта. Второе число (обозначенное цифрой 2) обычно используется для указания на текущие обновления программы, заметные и важные, но не приводящие к кардинальному изменению программной оболочки. Мы можем назвать это число *промежуточной официальной версией* продукта. Третье число (обозначенное цифрой 3) изменяется обычно незаметно для пользователя и связано с внутренними изменениями в программе. Иногда такие изменения довольно важны для программистов, но могут не иметь никакого значения с точки зрения пользователей. Мы можем назвать это число *внутренней версией* продукта. Наименее значимым для пользователя является четвертое число (обозначенное цифрой 4), оно изменяется при незначительных переменах в интерфейсе программы, внешних ресурсах, документации. Очень часто четвертое число отражает дату последнего обновления программы или изменения дополнительных модулей (как, например, обновление антивирусных баз). Как правило, его называют *номером сборки* или *билдом* (от английского обозначения *build*). Такой подход позволяет пользователям (и даже потенциальным последователям создателей программы), оценить, насколько было протестировано в реальных условиях данное программное обеспечение, насколько современной является используемая версия. Зачастую при изменении версий ПО возникают конфликты, связанные с тем, что существенные изменения программ разработчики уже могут снова продавать, а менее важные – должны предоставлять бесплатно. Иногда желание заработать денег перевешивает ответственность перед пользователями и в создании номеров версий приводит к намеренным ошибкам в изменении версий. Например, разработчики могут изменить номер версии, даже если ни одна строчка кода не была переписана, лишь для того чтобы создать ложное впечатление, что были внесены значительные изменения в код программы. Или просто перерисовать иконки в интерфейсе ПО и сказать, что это отражает существенное изменение функционала программы. Доказать их неправоты в этих случаях очень сложно.

Типичная нумерация в данной схеме может выглядеть так (опуская непринципиальное четвертое число): 0.9, 0.9.1, 0.9.2, 0.9.3 – для начала работы над программой; 1.0, 1.0.1, 1.0.2, 1.1, 1.1.1 – для эволюций первой версии; 2.0, 2.0.1, 2.0.2, 2.1, 2.1.1, 2.1.2, 2.2 – для второй и т. д. Разработчики порой перескакивают от версии 5.0 сразу к 5.5, для того чтобы обозначить добавление нескольких значимых функций в программе, однако их недостаточно, чтобы изменить главный номер версии. Не все согласны с такой практикой и подобные скачки используются редко.

Очень часто промежуточные версии не выпускаются, в этом случае версии программы всегда целые и означают накопление существенных изменений для выхода новой версии продукта. Так, например, часто поступает компания *Adobe* со своими продуктами. В большинстве программного обеспечения, первая официальная версия программного продукта имеет версию 1, но это не правило, а скорее некая общепринятая практика.

Другой подход использования главных и второстепенных номеров версий заключается в добавлении буквенно-цифровой последовательности, определяя тем самым стадию разработки релиза: «альфа», «бета», «релиз-кандидат». На самом деле эти обозначения позволяют не только увидеть, с какой версией идет работа, но и понять, на какой стадии находится разработка программы. В разработке программного обеспечения стадии разработки программного обеспечения используются для описания степени готовности программного продукта и могут применяться для любых версий создаваемого ПО, но наиболее важны в момент, когда выпускается первая версия программы. Стадия разработки может отражать количество реализованных функций, запланированных для определённой версии программы, степень их готовности, уровень доступности программы пользователю (бесплатная пробная версия или готовое платное ПО). Стадии либо могут быть официально объявлены и регламентируются разработчиками, либо иногда этот термин используется неофициально для описания состояния продукта. В любом случае, использование специальных обозначений в номере версии, как правило, требует написания соответствующего лицензионного соглашения. Этапы разработки присваиваются любому создаваемому программному обеспечению, как правило, в виде следующих названий:

1. *Пре-альфа* версия. Начальная стадия разработки – момент, когда программа только начинает разрабатываться или модернизироваться из существующей стабильной версии. На этом этапе новый программный продукт содержит множество ошибок, и может не обладать полным функционалом, но показывает, в какую сторону двигаются разработчики. Такое название присваивают программам, прошедшим стадию разработки, для первичной оценки функциональных возможностей в действии.
2. *Альфа* версия. Внутреннее тестирование – стадия начала тестирования программы в целом специалистами-тестерами, обычно не разработчиками программного продукта, но, как правило, внутри организации, разрабатывающей продукт. На этой стадии могут добавляться новые функциональные возможности. Программы, находящиеся на первых двух стадиях разработки, могут применяться только для ознакомления с будущими возможностями и не становятся доступны пользователям.
3. *Бета* версия. Публичное тестирование – стадия активного тестирования и отладки программы, прошедшей первые этапы, если они были. Программы этого уровня могут быть использованы другими разработчиками программного обеспечения для испытания совместимости. Их

могут начинать демонстрировать пользователям, зачастую набирая добровольных тестировщиков среди них (как часто бывает с компьютерными играми). Программы на этом этапе разработки по-прежнему могут содержать достаточно большое количество ошибок. Бета-версия ПО не является финальной версией, и публичное тестирование производится на страх и риск пользователя, разработчик не несёт никакой ответственности за ущерб, причинённый в результате использования бета-версии. Этим часто пользуются создатели программ, предоставляя пользователям только бета-версии продукта. Например, разработчики ICQ в 2003-м году использовали это, выпустив бета-версию программы. Финальной версии ICQ 2003 так и не появилось, вместо этого два года спустя вышли версии ICQ 4 и ICQ 5. Многими сообществами признано, что 2003 версия продукта была самой «глючной» и с самым большим содержанием рекламы за всю историю этого мессенджера.

4. *Релиз-кандидат* или *rc* (от англ. *release candidate*) – стадия-кандидат на то, чтобы стать итоговой. Программы этой стадии прошли комплексное тестирование, благодаря чему были исправлены все найденные критические ошибки, тем не менее, некоторые ошибки могут быть выявлены на этом этапе. Как правило, релиз-кандидаты используются для презентации новых продуктов и часто создаются исключительно для маркетинговых целей.
5. *Релиз* или *RTM* (от англ. *release to manufacturing* промышленное издание) – издание продукта, готового к тиражированию и продаже. Это стабильная версия программы, как правило, не содержащая ошибок, прошедшая все предыдущие стадии.

Следует отметить, что стадии *Бета* и *Альфа*, выносимые иногда «на суд общественности», не являются показателями нестабильности данной версии программы, т.к. присваиваются программе один раз или один раз за серию (изменения в первом числе в номере версии), они могут присваиваться нескольким релизам подряд. Важной характеристикой использования таких схем обозначения версий и этапов является соблюдение идентичности стадий разработки версий. Например, нельзя вносить никаких изменений между последней бета-версией и первым релиз-кандидатом или последним релиз-кандидатом и готовым продуктом. Если это сделано, необходимо выпустить другую версию на более низкой стадии разработки, чтобы не прерывать нумерацию вносимых изменений в программу. В случае если в процессе использования программы пользователями в ней обнаруживаются новые ошибки, то их также устраняют, выпуская *заплатки* или *патчи*. Наряду с обновлениями программы они тоже вносят изменения в версию продукта.

Теперь можно сформулировать некоторые правила и законы проектирования программного обеспечения. *Проектирование ПО* – это целая наука и, конечно, все рассказать в рамках короткого курса невозможно, но некоторые интересные факты мы обсудим. Проектированию подлежат все три составных части программы – интерфейс, функциональная часть и архитектура. Причем в каждой области, проекти-

рование – серьезная и сложная задача, требующая разработки специалистами в данной области, также проектирование – неотъемлемая часть программирования. Существуют разные подходы к проектированию ПО, как «ручные» так и автоматизированные. Если проект несложный, то его можно спроектировать просто «на бумаге», набросав план дальнейших действий и разрабатывая программу в соответствии с ним. Но если создаваемая программа сложна и состоит из большого числа компонентов, то в этом случае часто прибегают к помощи вспомогательных программ. Одним из наиболее известных производителей такого специализированного ПО является компания *IBM Rational Software*. Этой компанией были разработаны различные средства автоматизации проектирования программных продуктов и баз данных, в том числе *RUP (Rational Unified Process)*. В основе Rational Unified Process лежат следующие принципы:

- раннее обнаружение и непрерывное (до окончания проекта) устранение основных рисков, связанных с разработкой ПО;
- концентрация на выполнении требований технического задания;
- ожидание изменений в требованиях, проектных решениях и реализации в процессе разработки;
- компонентная архитектура ПО, реализуемая и тестируемая на ранних стадиях проекта;
- постоянное обеспечение качества на всех этапах разработки проекта;
- работа над проектом в команде, в которой ключевая роль принадлежит проектировщикам.

RUP – это, в своем роде, пособие по проектированию, оно помогает создавать новые виды программного обеспечения, уменьшая риски, сокращая сроки выполнения работ и т.п. Немного внимания мы обязательно уделим проектированию в главе 9 – когда будем обсуждать вопросы проектирования баз данных. Теперь нужно перейти к, пожалуй, самой важной главе и части курса – обсуждение разновидностей языков программирования.

7. Языки программирования

Прежде, чем начинать рассказ о том где, в каких областях и для решения каких задач используются языки программирования, необходимо рассказать о способах деления языков программирования на общепринятые в мире группы.

Одно из таких разделений – на низкоуровневые и высокоуровневые языки программирования. *Низкоуровневый язык программирования* – это такой язык программирования, программы на котором пишутся непосредственно для процессора или операционной системы. При этом, как правило, такие языки дают возможность работать непосредственно с командами процессора или машинными кодами. Кроме машинных команд языки программирования низкого уровня могут предоставлять дополнительные возможности, такие как *макроопределения* (все чаще называемые

в последнее время макросами), а также возможность использования специальных команд или инструкций. При помощи инструкций в таких языках появляется возможность управлять процессом трансляции машинных кодов, создавая более осмысленные тексты программ, понятные не только знакомым с процессорными командами специалистам, но и программистам более широкого круга. Часто эти языки позволяют работать вместо конкретных ячеек памяти с переменными, но при этом язык может зависеть от особенностей конкретного семейства процессоров. Классический пример одного из самых низкоуровневых языков – язык *ассемблера*, (группа языков ассемблера – что более корректно). Если говорить о более «продвинутых» языках, то к низкоуровневым языкам можно отнести и некоторые классические языки программирования, например, Си (хоть это и не совсем аккуратно). К плюсам таких языков можно отнести чрезвычайно высокую скорость работы получаемого кода, оптимальное использование и оптимальная работа с машинными ресурсами. Минусы низкоуровневых языков – довольно высокая сложность исходного кода и наличие жестких ограничений на создаваемый код.

Высокоуровневый язык программирования – язык программирования, разработанный для быстроты и удобства использования программистом. Их главное отличие и преимущество перед низкоуровневыми языками – это абстракция, то есть введение особых конструкций языка, кратко описывающих такие структуры данных и операции над ними, описания которых на машинном коде (или другом низкоуровневом языке программирования) могут быть очень громоздкими и сложными для понимания даже опытными программистами. Высокоуровневые языки позволяют не только облегчить решение сложных задач, но и существенно упростить процесс написания программного обеспечения, благодаря использованию, возможно, менее универсальных, зато более понятных и простых программных решений. К сожалению, тот факт, что высокоуровневые языки программы довольно далеки от аппаратной реализации компьютера, имеет свои минусы. Например, код, формируемый такими языками, работает гораздо медленнее низкоуровневых аналогов, программы получаются менее эффективными, на таких языках нельзя писать программные инструкции к используемому оборудованию (*драйверы*) и т.д. Почти все современные языки программирования могут быть отнесены в категорию высокоуровневых. Причем многие из них, например, скрипты, еще и зависят от установленного на компьютере программного обеспечения (*Java, PHP, 1C*). История высокоуровневых языков программирования, как считается, началась в 40-х годах прошлого века, но наиболее широкое распространение эти языки получили с созданием Фортрана в 1957 году.

Еще одно деление языков программирования в современности может уже считаться устаревшим, но оно позволяет лучше понять разницу между еще двумя специальными терминами – блоками и модулями. *Блочный язык программирования* – язык, в котором вся программа делится на части (подпрограммы), находящиеся в определенной иерархической связи друг с другом. Эти участки кода называются *блоками* и должны содержаться в одном файле. Блочная структура программы позволяет при распределении памяти отводить одни и те же поля памяти машины для хранения величин, описанных в блоках, которые не пересекаются, что приводит к

существенной экономии памяти во время работы программы. *Модульные языки программирования* разделяют программу на *модули* – отдельные файлы, в которых хранятся законченные фрагменты кода. Это деление было популярно в 60-70-х годах прошлого века, когда формировались такие языки программирования как *Си*, *Фортран*, *Паскаль*, *Кобол*, *Ада* и другие. Однако оно очень быстро исчерпало себя, и современные языки программирования являются блочно-модульными. То есть, позволяют хранить подпрограммы как в блоках, так и в отдельных модулях.

Наиболее важным делением языков программирования является деление по принципу используемых *методологий программирования*. Методология (или технология) программирования описывает совокупность правил, идей, методов, способов осуществления программистской деятельности. В зависимости от выбранной методологии различаются исходный код программы, способы создания ПО, используемые ресурсы и внешний вид конечного продукта. Мы опишем несколько наиболее важных и популярных методологий.

Методология *структурного программирования* в самой краткой формулировке есть проектирование методом «сверху вниз» (или методом *нисходящего проектирования*), т.е. написание текста программы от общего к частному, от наиболее общих шагов алгоритма к максимально подробной детализации каждого шага. Программирование для универсальных компьютеров начиналось с программирования в машинных кодах, затем появились и начали своё развитие языки высокого уровня. Еще не так давно использование языков высокого уровня было невозможно из-за малого объема внутренней памяти программ. Однако с развитием компьютерных технологий появилась потребность унифицировать процесс разработки программного обеспечения и ускорить его. Применение структурного программирования позволяет существенно увеличить скорость написания программ, уменьшить сложность кода, сократить число ошибок и облегчить отладку написанной программы. Данная методология программирования была предложена в 70-х годах XX века Э. Дейкстрой, и, впоследствии, разработана и дополнена Николаусом Виртом – автором известной книги «Алгоритмы и структуры данных», описывающей основные идеи разработки алгоритмов в программировании, построения эффективных и надежных программ. В соответствии с данной методологией любая программа представляет собой набор блоков или модулей, подчиненных общей иерархической схеме. Виртом были предложены следующие законы, которые легли в основу структурного программирования:

1. Любая программа представляет собой структуру, построенную из трёх типов базовых конструкций, повторяющих свои аналоги из классических алгоритмов:
 - a. последовательное исполнение – однократное выполнение операций в том порядке, в котором они записаны в тексте программы;
 - b. ветвление – однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия;

- с. цикл – многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла).
- 2. В программе базовые конструкции могут быть вложены друг в друга произвольным образом, но никаких других средств управления последовательностью выполнения операций не предусматривается. Таким образом, некоторые специальные команды языков программирования – например, оператор безусловного перехода (*goto*) формально не являются разрешенными.
- 3. Повторяющиеся фрагменты программы (или же логически завершённые участки кода) могут оформляться в виде *подпрограмм (процедур или функций)*.
- 4. Разработка программы ведётся пошагово, методом нисходящего проектирования.

Опишем основные преимущества такого подхода. Прежде всего, в случае применения структурного программирования сначала пишется текст основной программы, в котором вместо каждого связного логического фрагмента текста вставляется вызов подпрограммы, которая будет выполнять этот фрагмент. В результате основной исходный код становится более компактным, более понятным. Вместо помещённого в подпрограмму фрагмента в код просто вставляется инструкция вызова подпрограммы. Зачастую для ускорения разработки кода и упрощения отладки вместо настоящих, работающих, подпрограмм в программу вставляются пустые подпрограммы, не выполняющие никаких действий, не нагружающие результирующий машинный код. Если полученная программа работает правильно и без сбоев, программист продолжает написание кода, постепенно заменяя все подпрограммы-заглушки на реально работающие. В этом случае гарантируется работоспособность программы на каждом из этапов разработки. Программисту не приходится работать со всем текстом программы сразу – только с отдельными небольшими фрагментами. В случае возникновения необходимости внесения изменений такие изменения вносятся, не затрагивая части программы, непосредственно не связанные с ними. Это ещё больше повышает надёжность разработки кода. К сожалению, несмотря на все плюсы, структурное программирование имеет и свои минусы. Программы, которые пишутся таким методом, всегда конкретные. Они нацелены на решение определенных задач, что не очень удобно при разработке программного обеспечения для современных графических операционных систем. В этих задачах чаще требуется универсальность подхода, сборка программ из уже готовых «кирпичиков» – абстрактных участков кода, ускоряющих разработку ПО.

Такие возможности предоставляет более современная методология программирования. *Объектно-ориентированное программирование* (часто сокращаемое до *ООП*) – методология программирования, в основу которой положены понятия объектов и классов. Появление и развитие ООП начинается с 60-х годов прошлого века (язык *Simula 67*, появившийся в 1967 году). Тем не менее, наиболее широко и полно идеи этой методологии развились позже – к 80-м годам XX века, т.к. изначально

идеи ООП не были восприняты как нечто очень важное и принципиальное для развития программирования. Здесь огромную роль сыграли Алан Кэй и Дэн Ингаллс, создав язык *Smalltalk*. Именно он стал первым широко распространённым объектно-ориентированным языком программирования, на основе идей этого языка Бьерн Страуструп разрабатывал наиболее популярный в современном программировании язык Си++. В настоящее время большинство программных проектов реализуются в стиле ООП, благодаря возможности с большой скоростью разрабатывать новые проекты из фактически уже готовых элементов – классов. Принципы, положенные в основу ООП, очень просты – мы живем в мире *объектов*. Есть автомобили, дома, компьютеры, и многое другое, что обладает не только своими уникальными особенностями, но и общими чертами. Существуют абстрактные объекты – числа, символы, кнопки на поверхности современного графического приложения. Все они могут быть описаны абстрактно, а затем каждый конкретный объект появляется из такого абстрактного описания. Каждый объект характеризуется своими атрибутами и выполняемыми действиями, все их можно универсальным образом описать в виде *класса* – абстрактной конструкции, выражающей общие черты каждого уникального объекта. Понятно, что написание программ, использующих уже готовые классы, идет в несколько раз быстрее, чем если бы каждый объект пришлось описывать уникальным способом. Основными понятиями ООП становятся несколько принципов, описывающих уникальную организацию таких языков программирования:

1. *Абстракция данных*. Все программы строятся на основе классов – абстрактных конструкций, отображающих общие свойства конкретных объектов. Объекты представляют собой упрощенное, идеализированное описание реальных сущностей предметной области.
2. *Инкапсуляция*. Согласно этому принципу, любой класс должен рассматриваться как чёрный ящик – пользователь класса должен видеть и использовать только интерфейсную часть класса (т. е. список «внешних» декларируемых свойств и методов класса) и не вникать в его внутреннюю реализацию. Тем самым повышается простота использования классов и надежность их использования, без опасений сломать что-то «внутри» класса.
3. *Соккрытие данных*. Неотделимая часть ООП, логичное следствие инкапсуляции, управляющая областями видимости. Ограничивает доступ к данным внутри класса.
4. *Наследование*. Наследованием называется возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка и добавляя, при необходимости, новые свойства и методы.
5. *Полиморфизм*. Полиморфизмом называют явление, при котором функции (методу) с одним и тем же именем соответствует разный программный код (полиморфный код) в зависимости от того, объект какого класса используется при вызове данного метода.

Среди плюсов ООП однозначно нужно выделить высокую скорость разработки кода и максимальное упрощение программирования для программистов, лишь использующих классы. Среди минусов такого подхода нужно выделить, к сожалению, более медленную, по сравнению со структурными языками, работу результирующего машинного кода. А также достаточно высокую сложность разработки собственно классов, для создания которых важно понимать внутреннее устройство всех специальных механизмов ООП. Часто можно встретить фразу, что знание других методологий не помогает в изучении ООП. Отчасти это правда. Принципы и приемы ООП сильно отличаются от других языков программирования. Например, язык Си++ очень сильно отличается от простого Си, по большому счету, у этих языков немного похож синтаксис. Но верно и другое – понимание принципов структурного или какого-либо другого подхода к программированию позволяет быстрее изучать ООП и создавать более эффективный и грамотный код.

Как я уже писал, существуют и другие, более специальные методологии программирования: логическое программирование, функциональное программирование, программирование в ограничениях и прочие. Эти методологии также применяются в современном программировании, но в узком круге задач.

Давайте опишем некоторые особенности современных языков программирования, а также – какие задачи с их помощью решают. Наиболее важными для программирования и в настоящее время остаются языки программирования структурные и низкоуровневые. Благодаря высокой скорости рабочего кода и возможности работать напрямую с аппаратной частью компьютера они обеспечивают функционирование всех современных программ. Именно на этих языках пишутся функциональные части программ – «математика», без них современные операционные системы, сложное программное обеспечение, расчетные модули, архивация и шифрование файлов работали бы в разы медленнее.

Важные задачи решают и высокоуровневые, как правило, объектно-ориентированные языки. Они облегчают задачу написания интерфейсов программ, в том числе графических. На них создаются многие современные прикладные программы и программное обеспечение для мобильных устройств. Эти языки привлекают к себе простой организацией процесса создания новых приложений и достаточно легким в изучении синтаксисом. Но внутреннее устройство таких языков может быть достаточно сложным и их изучение – интересный и не всегда простой процесс. К сожалению, совсем простых языков не бывает – программирование сложная, но интересная область работы.

8. Структуры данных

Для понимания внутреннего устройства баз данных, а также того, как они были придуманы, нужно ввести еще два понятия, широко используемых в программировании. Это специальные структуры данных, предназначенные для хранения разных данных, в том числе, сложных.

Один из таких типов данных – *массив*. В создании внутренней конструкции баз данных этот тип данных оказался самым важным. Массив – это непрерывный, именованный набор однотипных переменных, доступ к которым осуществляется по *индексу*. Индексом массива, как правило, является целое число (одно или несколько), в некоторых скриптовых языках, например *JavaScript*, *PHP* применяются также *ассоциативные массивы*, в которых переменные не обязаны быть однотипными, и доступ к ним не обязательно осуществляется по индексу, но это специфика данных языков программирования, не имеющая никакого отношения к классическому определению.

Наиболее часто встречающийся в программировании вид массива – одномерный, который можно представить себе как просто данные, записанные в ряд. С такими массивами не всегда удобно работать, но они очень эффективно хранятся в памяти и компьютеру с ними работать удобнее. Именно они чаще всего встречаются в расчетных задачах, и они же нам понадобятся для конструирования формата файла баз данных. Также встречаются двумерные массивы, или таблицы, и трехмерные массивы. Массивы более высокой размерности существуют, но ими в программировании пользуются крайне редко – это ненужно и неудобно.

Основные преимущества массива заключаются в операциях, которые совершаются над ними очень быстро. Прежде всего, это доступ к данным и поиск. Благодаря тому, что информация в памяти хранится непрерывно, в любой момент программист может быстро получить доступ к данным по индексу массива, что позволяет быстро читать и менять данные в массиве. Поиск, благодаря быстрому доступу, тоже работает быстро – конечно, простой перебор не эффективен, но его всегда можно оптимизировать. В связи с тем, что в массиве легко быстро искать данные, в массиве легко собирать статистическую информацию (сколько в массиве одинаковых данных или пустых и т.д.), и сортировать данные. Все эти качества важны не только для хранения данных в оперативной памяти, но и для баз данных. Одна проблема – в массиве можно хранить только одинаковые данные. А информация – разная. Но эту проблему легко обойти, для этого нужно рассмотреть еще один вид специальных конструкций данных.

Записи – это особый вид данных, используемый наряду с массивами – это структурированный тип, содержащий набор объектов разных типов. Составляющие запись объекты называются ее *полями*. В записи каждое поле имеет свое собственное имя. Чтобы описать запись, необходимо указать ее имя, имена объектов, составляющих запись и их типы. Для нас очень важным является то, что записи позволяют хранить разнотипные данные, что позволит одновременно с этим организовать хранение таких данных в массиве.

Теперь принцип устройства формата файлов базы данных видно невооруженным глазом. Правило и в самом деле очень простое. Вся информация, которую необходимо сохранить в базе разбивают на *записи* (будущие строки базы) в которых вся информация хранится в фиксированном виде – в *полях* (столбцы). Каждое поле имеет фиксированный размер и бывает числовым, текстовым, датой и т.д. Всю информацию записывают в файл в виде массива – все записи пишутся подряд, без

промежутков и занимают в файлах одну и ту же длину. В результате мы получаем, что файл базы данных – это просто массив записей, хранящийся на диске, а не в оперативной памяти. Вот такая простая идея. Она оказалась настолько удачной, что с момента создания баз данных и развития идей их устройства (1955-1960 гг.) принципиально этот формат не менялся. Терминология, связанная со структурой баз данных также берется из программирования. В базах данных не используются термины *строка* или *столбец* (более того, использование термина *таблица* применительно к базе данных некорректно – но это мы еще отметим), вместо этого говорят *запись* и *поле* соответственно. Формальное определение базы данных может звучать так:

1. База данных – организованная в соответствии с определёнными правилами и поддерживаемая в памяти компьютера совокупность данных, характеризующая актуальное состояние некоторой предметной области и используемая для удовлетворения информационных потребностей пользователей.
2. База данных – совокупность данных, хранимых в соответствии со схемой данных, манипулирование которыми выполняют в соответствии с правилами средств моделирования данных.
3. База данных – некоторый набор перманентных (постоянно хранимых) данных, используемых прикладными программными системами какого-либо предприятия.
4. База данных – совместно используемый набор логически связанных данных (и описание этих данных), предназначенный для удовлетворения информационных потребностей организации.

По большому счету, все эти определения говорят об одном и том же и могут использоваться равноправно. Существует огромное количество разновидностей баз данных, отличающихся по различным критериям. Например, в «Энциклопедии технологий баз данных» (Когаловский М.Р.), определяются свыше 50 видов БД. Мы отметим некоторые, наиболее часто используемые.

1. Классификация по модели данных:
 - a. Иерархическая
 - b. Сетевая
 - c. Реляционная
 - d. Объектная и объектно-ориентированная
 - e. Объектно-реляционная
 - f. Функциональная.
2. Классификация по среде постоянного хранения:
 - a. Во вторичной памяти, или традиционная: средой постоянного хранения является периферийная энергонезависимая память (вто-

ричная память) — как правило, жёсткий диск. В оперативную память СУБД помещает лишь временный кэш и данные для текущей обработки.

- b. В оперативной памяти: все данные на стадии исполнения находятся в оперативной памяти.
- c. В третичной памяти: средой постоянного хранения является отсоединяемое от сервера устройство массового хранения (третичная память), как правило, на основе магнитных лент или оптических дисков. Во вторичной памяти сервера хранится лишь каталог данных третичной памяти, файловый кэш и данные для текущей обработки; загрузка же самих данных требует специальной процедуры.

3. Классификация по содержанию:

- a. Географическая.
- b. Историческая.
- c. Научная.
- d. Мультимедийная.

4. Классификация по степени распределения:

- a. Централизованная, или сосредоточенная: БД, полностью поддерживаемая на одном компьютере.
- b. Распределённая: БД, составные части которой размещаются в различных узлах компьютерной сети в соответствии с каким-либо критерием.
- c. Неоднородная: фрагменты распределённой БД в разных узлах сети поддерживаются средствами более одной СУБД
- d. Однородная: фрагменты распределённой БД в разных узлах сети поддерживаются средствами одной и той же СУБД.

Существуют и другие виды баз данных, более специфические, при этом, каждая конкретная база данных может относиться к нескольким типам сразу. Вопросы проектирования и работы с базами данных мы рассмотрим в главе 9. Но прежде, обратимся еще к нескольким интересным фактам, относительно структурных видов данных.

На основе записей можно построить специальные *динамические конструкции данных*, широко используемые в создании алгоритмов расчетных и системных задач. Вряд ли любому программисту понадобится разбираться в этих конструкциях, но знать о них и представлять себе то, как они работают, безусловно, важно.

Абстрактные структуры данных предназначены для удобного хранения и доступа к информации. Они предоставляют удобный интерфейс для типичных операций с хранимыми объектами, скрывая детали реализации от пользователя. Ко-

нечно, это весьма удобно и позволяет добиться большей модульности программы. Среди обычных типов данных также присутствуют абстрактные структуры данных – например, числа. Языки программирования высокого уровня (Паскаль, Си) предоставляют удобный интерфейс для чисел: операции $+$, $*$, $=$, ..., но при этом скрывают саму реализацию этих операций, машинные команды. В языках объектно-ориентированных идут еще дальше, давая возможность создавать целые программы из абстрактных конструкций.

Динамические структуры по определению характеризуются отсутствием физической связи между элементами структуры в памяти, непостоянством и непредсказуемостью размера (числа элементов) структуры в процессе ее обработки. Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента, то есть элементы таких структур «ничего не знают друг о друге». Для установления связи между элементами динамической структуры используются специальные *ссылки* (в языке Си – *указатели*). Такое представление данных в памяти называется связным. Элемент динамической структуры состоит из двух полей:

- информационного поля или поля данных, в котором содержатся те данные, ради которых и создается структура;
- поле для осуществления связей, в котором содержатся одна или несколько ссылок, связывающих данные элементы с другими элементами структуры.

Когда связное представление данных используется для решения прикладной задачи, для конечного пользователя «видимым» делается только содержимое информационного поля, а поле связей используется только программистом-разработчиком, согласно принципам абстракции. Преимуществом такого подхода

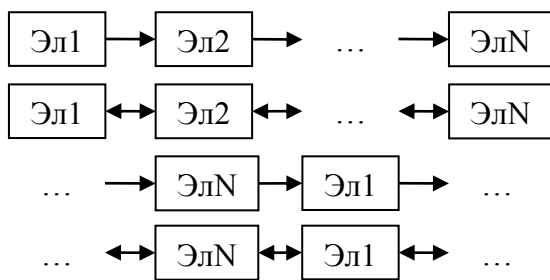


Рисунок 9 – Виды списков

является возможность произвольно менять размеры структур, как угодно переставлять связи между элементами, быстро добавлять и удалять элементы в любое место структуры. Естественно, за эти преимущества приходится платить – расходом лишней оперативной памяти на ссылки, более медленным доступом к данным, чем, например, в массиве, довольно сложным программированием

механизма связей. Основной минус здесь – это замедление доступа к данным. В таких структурах теряет смысл индексация, принятая в массивах, получить доступ к данным можно только посредством ссылок и последовательного просмотра всех элементов.

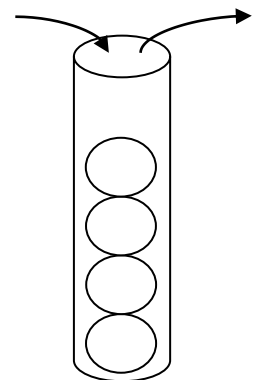
Одними из наиболее важных видов динамических структур данных являются списки. *Списком* называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции *включения*, *исключения* (добав-

ления и удаления элементов соответственно). Всего в программировании используют несколько принципиально разных видов списков, их принципиальное устройство показано на Рисунок 9. У этих списков есть специальные названия (сверху вниз):

- *Линейный однонаправленный.*
- *Линейный двунаправленный.*
- *Циклический (кольцевой) однонаправленный.*
- *Циклический (кольцевой) двунаправленный.*

На базе списков можно построить еще более интересные конструкции: стеки, деки и очереди, а также, деревья.

Стек – линейный список с переменной длиной, включение и исключение элементов из которого выполняются только с одной стороны списка, называемого вершиной стека, принцип работы стека часто выражают аббревиатурой LIFO (Last - In - First- Out – «последним пришел – первым исключается»). Основные операции над стеком – включение нового элемента (английское название *push*) и исключение элемента из стека (англ. *pop*). Формальное описание стека – вещь скучная, лучше всего представить себе стек в виде стакана, куда складывают шарики (см. Ри-



сунук 10), тогда все законы становятся более понятными – почему элементы можно добавлять и брать только «сверху» – для извлечения шариков из середины стека будут «мешаться стенки». Чаще всего используются стеки в системном программировании – именно в них операционная система хранит информацию о переменных, используемых программами.

Рисунок 10 – Стек

Дек – особый вид очереди, это такой последовательный список, в котором как включение, так и исключение элементов может осуществляться с любого из двух концов списка. Наиболее часто встречающийся вид очереди – *очередь FIFO* (First - In - First- Out – «первым пришел – первым исключается») – это последовательный список с переменной длиной, в котором включение элементов выполняется только с одной стороны списка (эту сторону часто называют концом или хвостом очереди), а исключение – с другой стороны (называемой началом или головой очереди). Классическим примером очереди может быть обычная очередь или автоматизированный конвейер.

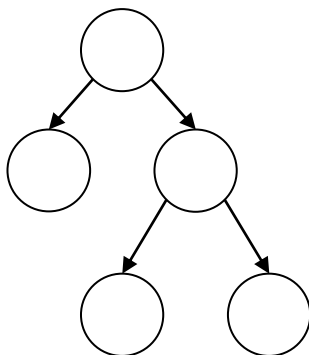


Рисунок 11 – Дерево

Все эти структуры – одномерные: в них один элемент следует за другим. *Дерево* – двумерная связанная структура, на которых основаны многие из наиболее важных алгоритмов, например алгоритмы сжатия данных, шифрования данных. Полное описание деревьев могло бы занять не одну книгу, потому что они возникают во многих задачах, даже

вне информатики, и довольно интенсивно изучаются как математический объект. Мы рассматривать деревья подробно не будем, лишь укажем на то, какие виды деревьев бывают, и опишем основные определения.

Самыми простыми из деревьев считаются бинарные деревья. *Бинарное дерево* – это конечное множество элементов, которое либо пусто, либо содержит один элемент, называемый *корнем дерева*, а остальные элементы множества делятся на два непересекающихся подмножества, каждое из которых само является бинарным деревом. Эти подмножества называются *левым* и *правым поддеревьями исходного дерева*. Каждый элемент бинарного дерева называется *узлом дерева* (см. Рисунок 11).

Также бывают *красно-черные деревья*, *деревья поиска* (используемые при программировании DNS-серверов) и т.д.

9. Базы данных и их проектирование

В этой главе мы с Вами обсудим вопросы работы с базами данных на программном уровне и проектирования баз данных и таких программ.

Начнем с самого сложного – это самая сложная с точки разработки и самая функциональная возможность работать с базами данных – *информационная система*. Информационная система (ИС) может быть определена двумя способами – как в широком, так и в узком смысле. В широком смысле информационная система это совокупность технического, программного и организационного обеспечения, а также персонала, предназначенная для того, чтобы своевременно обеспечивать надлежащих людей надлежащей информацией. Зачастую в это определение персонал не включают, что, на мой взгляд, не очень правильно. Дело в том, что большая часть информационных систем – это очень сложное программное обеспечение и для работы с ним нужны высококвалифицированные специалисты. Зачастую работать с ИС может только специалист с соответствующим сертификатом. Таким образом, такие люди тоже могут быть неотъемлемой частью информационной системы. Используют это определение, когда говорят про работу в информационном пространстве компании, когда учитывают и компьютеры, и сети, и программное обеспечение, и персонал.

В узком смысле информационной системой называют только подмножество компонентов ИС в широком смысле, включающее базы данных, СУБД и специализированные прикладные программы. Определение ИС в узком смысле используют, когда речь идет о непосредственной работе с программой.

Какое бы из определений мы не использовали, основной задачей ИС является удовлетворение конкретных информационных потребностей в рамках конкретной предметной области. В связи с этим, любая ИС содержит в себе язык программирования, чтобы можно было решать в рамках этой системы на самом деле любые задачи. Современные ИС состоят из двух принципиальных частей – системы управления базами данных и набора вспомогательных программ. Таким образом, совре-

менные ИС ассоциируются не столько с программным обеспечением, сколько с самими базами данных и СУБД. Но благодаря тому, что в ИС всегда есть дополнительные программы, выполняющие вспомогательные функции, их функции гораздо шире.

В идеале в рамках предприятия должна функционировать единая корпоративная информационная система, удовлетворяющая все существующие информационные потребности всех сотрудников, служб и подразделений. Однако на практике создание такой всеобъемлющей ИС слишком затруднено или даже невозможно, вследствие чего на предприятии обычно функционируют несколько различных ИС, решающих отдельные группы задач: управление производством, финансово-хозяйственная деятельность и т.д. Часть задач бывает «покрыта» одновременно несколькими ИС, часть задач — вовсе не автоматизирована. Такая ситуация получила название *лоскутной автоматизации* и является довольно типичной для многих предприятий.

В любом случае, проектирование такой системы это очень сложная задача для разработчика — и именно на этом примере мы рассмотрим вопросы, связанные с проектированием программного обеспечения.

Прежде всего, разработчикам программного обеспечения приходится столкнуться с требованиями, которые предъявляют пользователи к создаваемым программам. Если мы говорим про базы данных, то самое главное требование в современном мире — это, конечно же, обеспечение надежного и безопасного хранения информации. Учитывая, как часто происходят кражи информации в наши дни, это требование является самым главным и его необходимо удовлетворить в первую очередь. Затем находят компромисс между быстродействием, надежностью передачи данных и простотой использования. Эти требования редко удается удовлетворить все сразу, поэтому и приходится делать выбор.

Теперь опишем, какие задачи приходится решать в процессе проектирования базы данных:

1. Обеспечение хранения в БД всей необходимой информации. Мы должны гарантировать, что любые новые данные, которые будут появляться в процессе работы с базой, смогут быть сохранены в ней без изменения ее структуры.
2. Обеспечение возможности получения данных по всем необходимым запросам. То есть, структура и формат базы данных должны подразумевать возможность поиска любой нужной информации с любыми критериями выбора.
3. Сокращение избыточности и дублирования данных. Это требование крайне важно — оно позволяет гарантировать то, что база не будет расходовать лишний объем памяти, и данные не будут повторять друг друга (это опасно — может привести к путанице).

4. Обеспечение целостности данных (правильности их содержания). Это требование к базе позволяет исключить противоречия в содержании данных, исключение их потери, запись данных в неверные поля и т.д.

Как видите, задачи, решаемые в процессе проектирования, охватывают все вопросы, связанные с хранением информации. Разработка новой информационной системы еще и связана с созданием большого набора вспомогательных программ, таким образом, создание информационной системы – один из наиболее сложных видов проектирования. Опишем основные этапы проектирования:

Концептуальное (инфологическое) проектирование – построение концепции устройства базы данных, способов организации информации и устройства программного обеспечения. Это разработка идеи – основных, общих конструкций новой ИС. Такая модель создается без ориентации на какую-либо конкретную реализацию и модель данных – без конкретики. Термины *семантическая модель*, *концептуальная модель* и *инфологическая модель* являются синонимами. Конкретный вид и содержание концептуальной модели базы данных определяется выбранным способом проектирования. Чаще всего используются графические нотации, например, *ER-диаграммы* и специальные *UseCase модели*. Чаще всего концептуальная модель базы данных включает в себя описание методов работы с данными и структуру данных, описание того, какие

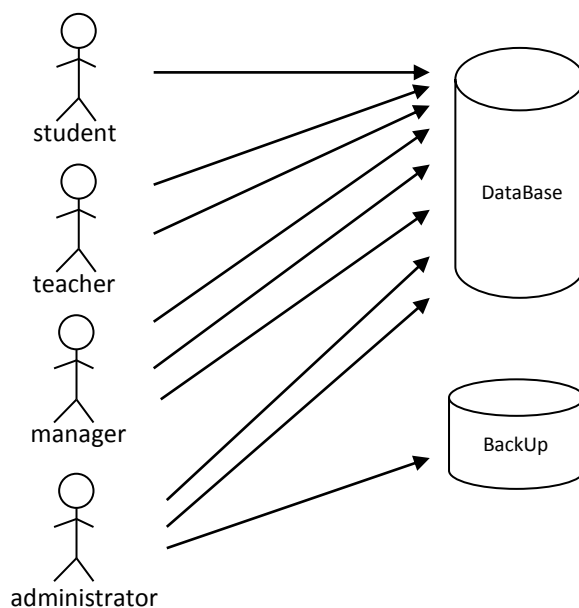


Рисунок 12 – UseCase модель

пользователи будут работать с базой и т.д. В процессе проектирования одна из ключевых задач – определение того, какие пользователи, как и с чем смогут взаимодействовать с создаваемым ПО. Для решения такой задачи как раз и могут пригодиться ER-модели и UseCase-модели. На примере покажем, как это может выглядеть. Обратите внимание на Рисунок 12, на нем показано графическое представление UseCase модели для базы данных, скажем, электронного журнала некоего учебного заведения. По этой схеме видно, что разным видам пользователей мы даем разные возможности доступа к данным в базе. Например, студенты могут только читать данные из базы, в то время как менеджеры (скажем, сотрудники деканата) могут еще и редактировать информацию. Ну а самым главным является администратор, который, помимо прочего, имеет еще и доступ к аварийной копии базы. Кстати о наличии этой копии многие при проектировании базы забывают, в то время как для полноценного функционирования базы ее наличие обязательно.

Для построения ER-моделей используют различные способы визуального представления данных. Они уже гораздо сложнее UseCase диаграммы и имеют да-

же именные названия. Выделим ряд наиболее популярных способов отображения данных:

Нотация Питера Чена. Множества сущностей (составляющих части программ или функциональных элементов) изображаются в виде прямоугольников, множества отношений изображаются в виде ромбов. Если сущность участвует в отношении, они связаны линией. Если отношение не является обязательным, то линия пунктирная. Атрибуты изображаются в виде овалов и связываются линией с одним отношением или с одной сущностью.

Crow's Foot. Данная нотация была предложена Гордоном Эверестом под названием *перевернутая стрелка*, однако сейчас чаще называемая Crow's Foot (воронья лапка) или *Fork (вилка)*. Согласно данной нотации, сущность изображается в виде прямоугольника, содержащем её имя, выражаемое существительным. Имя сущности должно быть уникальным в рамках одной модели. Связь изображается линией, которая связывает две сущности, участвующие в отношении. Степень конца связи указывается графически, множественность связи изображается в виде своего рода вилки на конце связи. Атрибуты сущности записываются внутри прямоугольника, изображающего сущность, и выражаются существительным в единственном числе (возможно, с уточняющими словами).

Также есть и другие виды нотаций – Bachman notation, EXPRESS, IDEF1x, UML и т.д.

Логическое (даталогическое) проектирование – создание схемы базы данных на основе конкретной модели данных, например, реляционной модели данных. Для реляционной модели данных даталогическая модель – набор схем отношений, обычно с указанием первичных ключей, а также «связей» между отношениями, представляющих собой внешние ключи. Преобразование концептуальной модели в логическую модель, как правило, осуществляется по формальным правилам. Этот этап может быть в значительной степени автоматизирован. На этапе логического проектирования учитывается специфика конкретной модели данных, но может не учитываться специфика конкретной СУБД. Этот этап – один из самых важных в проектировании. На этом этапе, фактически, строится итоговый вариант ПО. После этого этапа риски «провалить» проект минимальны. Для проектирования на этом этапе также используются различные нотации, но уже гораздо более сложные, чем на этапе концепции.

Физическое проектирование – создание схемы базы данных для конкретной СУБД. Специфика конкретной СУБД может включать в себя ограничения на именование объектов базы данных, ограничения на поддерживаемые типы данных и т.п. Кроме того, специфика конкретной СУБД при физическом проектировании включает выбор решений, связанных с физической средой хранения данных (выбор методов управления дисковой памятью, разделение БД по файлам и устройствам, методов доступа к данным), создание индексов и т.д.

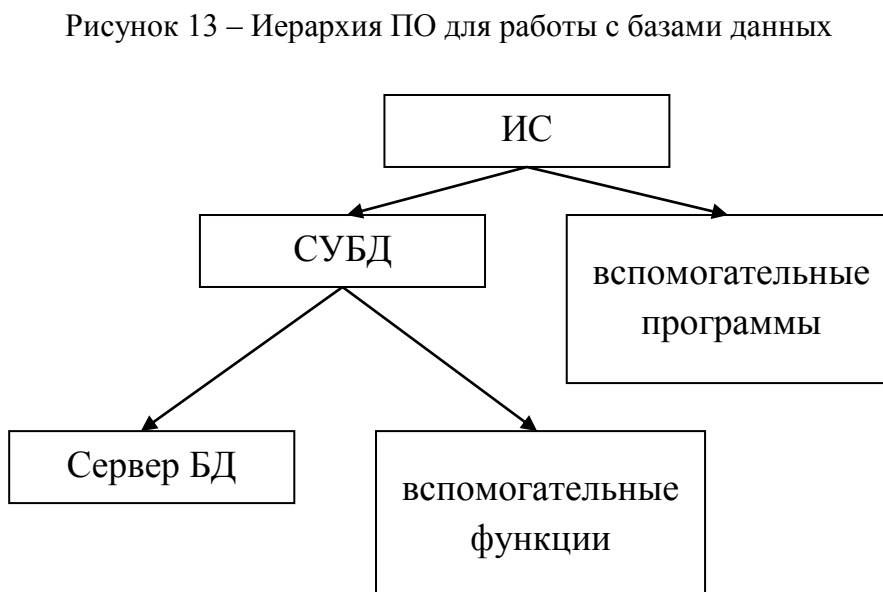
Более простая программа для работы с базами данных – это *СУБД* или *система управления базами данных*. Этот вид программного обеспечения по работе с

БД оптимален для небольших компаний и решения небольших задач по хранению информации. Основными функциями СУБД являются вполне конкретные задачи, в отличие от ИС, которая может удовлетворять любые информационные потребности пользователей. Укажем функции СУБД:

1. управление данными во внешней памяти (на дисках);
2. управление данными в оперативной памяти с использованием дискового кэша;
3. журнализация изменений, резервное копирование и восстановление базы данных после сбоев, создание отчетов, вспомогательные функции для конструирования запросов;
4. поддержка языков БД (язык определения данных, язык манипулирования данными).

Обычно современная СУБД содержит следующие компоненты: ядро, которое отвечает за управление данными во внешней и оперативной памяти, и журнализацию. Систему обработки языка базы данных, обеспечивающую независимый от самой системы способ извлечения данных – например, язык SQL. Систему, обеспечивающую работу пользовательского интерфейса и сервисные программы (внешние утилиты), обеспечивающие ряд дополнительных возможностей.

Самая простая и компактная программа по работе с информацией – это *сервер баз данных*. Такие программы имеют несколько отличительных черт. Они, как правило, очень компактные и быстродействующие. Сервера всегда обладают минимальным функционалом, как правило, обеспечивая первичный доступ к данным – красиво обработать и подать его пользователю нужно отдельно, в другой программе. Таким образом, сервера используются в качестве посредников для обработки данных другими приложениями. Цепочку, связывающую разные системы для работы с базами данных по их сложности и уровню решаемых задач, можно изобразить так:



Базы данных бывают разные, по способу внутренней организации данных их можно разделить на следующие группы:

- Иерархические
- Сетевые
- Реляционные
- Объектно-ориентированные
- Объектно-реляционные

Наиболее интересной и важной разновидностью баз данных являются реляционные базы. Их создание требует длительного проектирования, но при этом они позволяют решать любые задачи, предъявляемые пользователями к базам данных. *Реляционная база данных* – база данных, основанная на реляционной модели данных, эта тавтология – наиболее популярное определение этого вида БД. Дело в том, что эти базы обязаны своим появлением особому термину «реляционный», происходящему от англ. relation – *отношение*. Для работы с реляционными БД применяют реляционные СУБД, умеющие работать не с таблицами баз данных, а с их отношениями. Использование реляционных баз данных было предложено Эдгаром Франком Коддом из компании IBM в 1970 году. Им были разработаны все термины и правила работы с такими базами данных, которые используются и в наше время. Основная конструкция таких баз данных – это *нормальная форма*, устройство отношения между разными наборами данных в реляционной модели данных. Нормальная форма определяется как совокупность требований, которым должно удовлетворять отношение. Процесс преобразования отношений в БД к виду, отвечающему нормальным формам, называется *нормализацией*. Нормализация предназначена для приведения структуры БД к виду, обеспечивающему минимальную логическую избыточность, и не имеет целью уменьшение или увеличение производительности работы или же уменьшение или увеличение физического объёма базы данных. Конечной целью нормализации является уменьшение потенциальной противоречивости хранимой в базе данных информации, обеспечение решения всех задач, которые ставят пользователи перед БД, и хранение всей возможной информации.

При том, что идеи нормализации весьма полезны для проектирования баз данных, они отнюдь не являются универсальным или исчерпывающим средством повышения качества проекта БД. Это связано с тем, что существует слишком большое разнообразие возможных ошибок и недостатков в структуре БД, которые нормализацией не устраняются. Тем не менее, этот процесс и сами реляционные базы данных являются очень популярными для хранения информации.

В результате нормализации решения задачи хранения данных получается реляционная база в одной из нормальных форм – особых видов отношений в базе, при которых убираются противоречия в данных. Всего таких форм семь: первая нормальная форма (1NF), вторая нормальная форма (2NF), третья нормальная форма (3NF), нормальная форма Бойса-Кодда (BCNF), четвёртая нормальная форма

(4NF), пятая нормальная форма (5NF), доменно-ключевая нормальная форма (DKNF) и шестая нормальная форма (6NF).

Все современные разработки в области баз данных в настоящее время ведутся в области сетевых и распределенных баз данных. *Распределенные БД* представляют собою набор узлов, связанных коммуникационной сетью, в которой:

- каждый узел – это полноценная СУБД сама по себе;
- узлы взаимодействуют между собой таким образом, что пользователь любого из них может получить доступ к любым данным в сети так, как будто они находятся на его собственном узле.

Каждый узел сам по себе является системой базы данных. Любой пользователь может выполнить операции над данными на своём локальном узле точно так же, как если бы этот узел вовсе не входил в распределённую систему. Распределённую систему баз данных можно рассматривать как партнёрство между отдельными локальными СУБД на отдельных локальных узлах. Основные проблемы таких баз – это необходимость работать с невероятно большими объемами данных, обеспечивать надежность работы таких систем и безопасность хранения данных. Для разработки таких БД вводят даже особый список из дополнительных правил, всего их двенадцать. Среди них очевидные требования – вроде независимости от операционных систем или работа базы даже при отключении некоторых узлов, а также нетривиальные правила – независимость обработки данных от их расположения и независимость от фрагментации данных (разреженность в данных, их хранения в разных узлах распределенной БД). Эти и другие технологии работы с большими базами данных в настоящее время наиболее активно развиваются как в программном, так и в техническом смысле.

10. Приложения

Приложение 1. Краткий обзор языка программирования Си

В этом приложении мы кратко познакомимся с основными конструкциями языка Си. Наша цель – показать на реальных программах существенные элементы языка, не вдаваясь в мелкие детали, формальные правила и исключения из них. Для этого мы должны рассмотреть некоторые несложные элементы языка: структуру кода, примеры некоторых команд и операторов.

Единственный способ выучить новый язык программирования – это писать на нем программы. Для того чтобы понять, как устроены структурные языки программирования крайне важно, вначале, изучить язык Си. При изучении любого языка любой программист первой пишет всегда одну и ту же программу – которая выводит на экран монитора сообщение «Hello World!». Простейшая Си-программа, печатающая этот текст, выглядит так:

```
#include <stdio.h>

int main(void)
```

```

{
    printf("Hello World!\n");
    return 0;
}

```

Как запустить эту программу, зависит от системы, которую вы используете. Мы рассмотрим в качестве примера компиляцию и запуск программы в системе UNIX. Для компиляции подают в консоли следующую команду:

```
cc hello.c
```

Если при компиляции не возникло ошибок, то появится исполняемый файл со стандартным именем для этой ОС – *a.out*. Этот файл можно запустить на выполнение и на экране монитора появится текст:

```
Hello World!
```

И в конце этой строки произойдет перевод курсора на новую строку. Произойдет это благодаря команде *\n*, которая в языке Си является специальным символом, можно так сказать, имитирующим нажатие клавиши *Enter*. Для того чтобы пояснить написанный текст, дадим несколько пояснений. Программа на Си (и на многих других структурных языках, построенных на основе Си), каких бы размеров она ни была, состоит из функций и переменных. Функции содержат инструкции, описывающие вычисления, которые необходимо выполнить, а переменные хранят значения, используемые в процессе этих вычислений. Функции в Си похожи на подпрограммы других языков программирования, в приведенной программе – описана *главная функция* языка с именем *main*, это особое имя: любая программа начинает работу с первой команды функции *main*.

Первая строка программы:

```
#include <stdio.h>
```

сообщает компилятору, что он должен использовать в компиляции информацию о стандартной библиотеке ввода-вывода (standard input/output). Эта строка встречается в начале многих исходных файлов работающих с монитором и файлами Си-программ.

Единственная команда программы, заключается в вызове функции *printf*:

```
printf("Hello, world\n");
```

Функция вызывается по имени *функции*, после которого, в скобках, указывается список *аргументов*. Функция *printf* – это библиотечная функция, которая в данном случае напечатает последовательность символов, заключенную в двойные кавычки. Фигурные скобки в данном примере играют ту же роль, что и, например, ключевые слова *begin* и *end* в Паскале – указывают начало и конец блока программы.

Рассмотрим более сложный пример:

```

#include <stdio.h>

int main(void)

```

```

{
    int a,b,c;
    b=5;
    c=2;
    a=b+c;
    printf("Result = %d\n",a);
    return 0;
}

```

В данной программе наиболее интересными являются строки, содержащие переменные. Прежде всего, это строка

```
int a,b,c;
```

в которой мы объявляем три переменные целого типа. Такая строка обязательна во всех структурных языках программирования, она информирует компилятор о том, как в дальнейшем работать с такими переменными. В Си любая переменная должна быть объявлена раньше, чем она будет использована; обычно все переменные объявляются в начале функции перед первой исполняемой командой. В объявлении описываются свойства переменных, которые могут состоять не только из указания типа данных. Помимо типа данных `int` в Си имеется еще несколько базовых типов для данных, это:

- `char` – символьный тип;
- `long` – длинное целое;
- `float` – число с плавающей точкой;
- `double` – с плавающей точкой с двойной точностью.

Размеры объектов указанных типов зависят от машины и разрядности операционной системы.

Любая команда программы заканчивается точкой с запятой. Некоторые команды являются вычисляемыми выражениями. Например:

```
a=b+c;
```

В этой команде происходит очевидное действие – вычисление суммы значений двух переменных *b* и *c*. Результат сохраняется в переменную *a*. Таких операций в языках программирования много. Среди стандартных команд, общих для всех языков можно выделить такие:

- `+` – сложение;
- `-` – вычитание;
- `*` – умножение;
- `/` – деление;

- < – сравнение «меньше»;
- > – сравнение «больше»;
- <= – сравнение «меньше или равно»;
- >= – сравнение «больше или равно»;

Казалось бы, в этом списке должны быть и команды сравнения на равенство, но вот они, как ни странно, универсальными не являются и зависят от языка.

Также необходимо отметить еще две важные конструкции языка Си – это операторы условия и цикла. Чтобы рассмотреть эти два действия, посмотрим, для начала, на такой пример:

```
#include <stdio.h>
int main(void)
{
    int a,b,c;
    b=5;
    c=0;
    if (c!=0)
    {
        a=b+c;
        printf("Result = %d\n",a);
    }
    else
    {
        printf("Error! Division by zero!\n");
    }
    return 0;
}
```

Здесь ключевую роль играет оператор *if-else*, оператор условия. В рассмотренном примере на экран будет выведена строка

Error! Division by zero!

Поясним это. Структура оператора *if* может быть представлена в виде схемы (см. Рисунок 14). В соответствии с нею любой оператор условия работает следующим образом: если условие, указанное в операторе выполнено, то работает блок программы, написанный сразу после него. Если же условие не выполнено, то тот блок, что стоит после

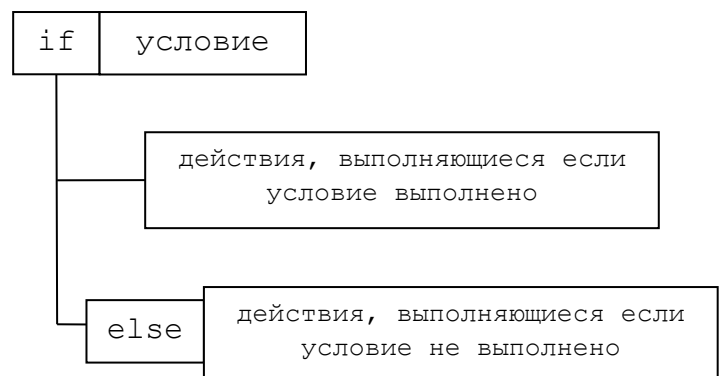


Рисунок 14 – Структура оператора if-else

ключевого слова *else*. Вторая часть условия не обязательна – оператор *if* может работать и без второй части. В связи с тем, что значение переменной *c* – 0 и условие, записанное в скобках, значит «*c* не равно нулю», то выполняется тот блок команд, что написан после *else* – а там как раз и стоит вывод на экран сообщения об ошибке.

Видов *операторов цикла* несколько. Универсальными и присутствующими во всех языках программирования циклами являются циклы *for*, *while* и *do-while*. Мы рассмотрим один вид – цикл перечислимого типа *for*. Два других вида циклов используются для более сложных общих задач и для простого примера не годятся.

Рассмотрим новый пример:

```
#include <stdio.h>

int main(void)
{
    int i;
    for (i=0; i<10; i++)
    {
        printf("i = %d\n",i);
    }
    return 0;
}
```

Здесь главную роль играет оператор цикла *for*. Структура записи в стиле языка Си может показаться непонятной и запутанной, но на самом деле, если подойти к этому оператору с простой точки зрения, то данная программа делает следующее: она печатает на экране столбец значений переменной *i* от 0 до 9. Начальное значение переменной задается командой

i=0

условие работы цикла командой

i<10

а шаг цикла – изменение переменной *i* на единицу – командой

i++

Переменная *i* в цикле *for* называется, обычно, итерационной переменной – от английского *iteration*. Это общепринятое и очень популярное название для переменной, которая «считает» шаги цикла. Довольно интересной в данном примере является команда *i++* – это специальная команда языка Си – она существенно ускоряет работу программы, но, к сожалению, в других языках программирования такие команды встречаются редко. Единственное действие, которое выполняется в этом цикле – печать на экран сообщений со значением итерационной переменной. Вот и получается такой результат:

i = 0

```
i = 1  
i = 2  
i = 3  
i = 4  
i = 5  
i = 6  
i = 7  
i = 8  
i = 9
```

Нумерация начинается с нуля по традиции, а также из-за специфического устройства массивов в языке Си – нумерация элементов массива в языке Си начинается с нуля.

К сожалению, обсудить все особенности языка Си невозможно в этом маленьком приложении – это один из самых элегантных, компактных и быстрых языков программирования. Изучать его нужно не столько для изучения языка, сколько для понимания того, как устроены практически все современные языки программирования «изнутри».

Приложение 2. Основные команды языка SQL, создание запросов

SQL – это специализированный язык запросов. Наиболее распространенное заблуждение – что SQL это язык программирования. На самом деле, это язык, который дает Вам возможность создавать и работать в базах данных, являющихся наборами связанной информации, сохраняемой в таблицах. Присутствует во всех системах управления базами данных и нужен для обеспечения надежного и простого доступа к данным.

Удобство, простота и независимость от специфики компьютерных технологий, а также его поддержка лидерами промышленности в области технологии реляционных баз данных, сделало SQL (и, вероятно, в течение обозримого будущего оставит его) основным стандартным языком для работы с БД.

Стандарт SQL определяется ANSI и в данное время также принимается ISO. Однако большинство коммерческих программ баз данных расширяют SQL без уведомления ANSI, добавляя различные особенности в этот язык, которые, как они считают, будут весьма полезны. Иногда они несколько нарушают стандарт языка, хотя хорошие идеи имеют тенденцию развиваться и вскоре становиться стандартами «рынка» сами по себе в силу полезности своих качеств. Чаще всего такие изменения происходят в сложных информационных системах. Классические примеры – Transact SQL и PL/SQL, используемые компаниями Microsoft и ORACLE соответственно.

В данном приложении мы не будем рассматривать все команды языка. Мы рассмотрим список основных команд языка для работы с данными и основную команду – команду выбора. Таких основных команд всего четыре:

- SELECT – выбрать
- INSERT – вставить
- UPDATE – обновить
- DELETE – удалить

Наиболее важной командой языка манипулирования данными является команда SELECT. Самое главное в изучении этой команды – научиться ее использовать. Синтаксис команд SQL очень простой – это будет понятно из тех примеров, что мы рассмотрим.

1. *SELECT * FROM Table1* – выбор всех данных из базы, по всем полям.
2. *SELECT name, surname FROM Table1* – выбор всех имен и фамилий из базы, в результате него формируется ответ в котором идут сначала имя, потом фамилия.
3. *SELECT * FROM Table1 WHERE surname='Иванов'* – выбирает из базы все данные по людям с фамилией Иванов.
4. *SELECT * FROM Table1 ORDER BY surname* – выбор всей информации из базы, упорядоченной по алфавиту, по полю фамилий.

Как видите, запросы устроены крайне просто. Фактически, работает правило «как пишется, так и читается» – главное в этих запросах не сама структура команды, а умение их писать. Поэтому изучение языка SQL, в основном, сводится к изучению и умению создавать различные запросы, что выливается в отсутствие специализированных курсов по этому языку. Его изучение всегда привязывают к конкретному программному обеспечению и решаемым задачам.

Приложение 3. Перевод чисел в разных системах счисления.

Для перевода чисел из десятичной системы счисления в двоичную используют так называемый *алгоритм замещения*, состоящий из следующей последовательности действий:

1. Делим десятичное число A на 2. Частное q запоминаем для следующего шага, а остаток a записываем, как младший бит двоичного числа.
2. Если частное q не равно 0, принимаем его за новое делимое и повторяем процедуру, описанную в шаге 1. Каждый новый остаток (0 или 1) записывается в разряды двоичного числа в направлении от младшего бита к старшему.

Алгоритм продолжается до тех пор, пока в результате выполнения шагов 1 и 2 не получится частное $q = 0$ и остаток $a = 1$.

Например, требуется перевести десятичное число 247 в двоичный вид. В соответствии с приведенным алгоритмом получим:

$$247_{10} \div 2 = 123_{10}$$

$$247_{10} - 246_{10} = 1$$

остаток 1 записывается в младший бит двоичного числа

$$123_{10} \div 2 = 61_{10}$$

$$61_{10} - 60_{10} = 1$$

остаток 1 записываем в следующий после младшего бита разряд двоичного числа

$$61_{10} \div 2 = 30_{10}$$

$$61_{10} - 60_{10} = 1$$

остаток 1 записываем в старший разряд двоичного числа

$$30_{10} \div 2 = 15_{10}$$

$$30_{10} - 30_{10} = 0$$

остаток 0 записываем в старший разряд двоичного числа

$$15_{10} \div 2 = 7_{10}$$

$$15_{10} - 14_{10} = 1$$

остаток 1 записываем в старший разряд двоичного числа

$$7_{10} \div 2 = 3_{10}$$

$$7_{10} - 6_{10} = 1$$

остаток 1 записываем в старший разряд двоичного числа

$$3_{10} \div 2 = 1_{10}$$

$$3_{10} - 2_{10} = 1$$

остаток 1 записываем в старший разряд двоичного числа

$$1_{10} \div 2 = 0_{10}$$

остаток 1 записываем в старший разряд двоичного числа

Таким образом, искомое двоичное число равно 11110111_2 .

Аналогичным способом можно переводить числа в другие системы счисления. Например – во вторую по значению систему счисления для компьютера – шестнадцатеричную:

1. Делим десятичное число A на 16. Частное q запоминаем для следующего шага, а остаток a записываем как младший бит шестнадцатеричного числа.
2. Если частное q не равно 0, принимаем его за новое делимое и повторяем процедуру, описанную в шаге 1. Каждый новый остаток записывается в разряды шестнадцатеричного числа в направлении от младшего бита к старшему.

Алгоритм продолжается до тех пор, пока в результате выполнения шагов 1 и 2 не получится частное $q = 0$ и остаток a меньше 16.

Сам компьютер использует эти и другие алгоритмы для перевода чисел непосредственно сразу при их обработке. Таким образом, время, затрачиваемое на эти процессы при расчете эффективности алгоритмов можно не учитывать.