

Библия для программиста в среде DELPHI

Автор: Horrific aka Фленов Михаил

e-mail: vr_online@cydsoft.com

Copyright 2002 год.

Содержание

Содержание	2
Введение	3
Структура книги	5
Глава 1. Основные принципы работы компьютера.....	8
1.1 Основы работы персонального компьютера.	8
1.2 Двоичная система работы процессора.	8
1.3 Машинный язык.	12
1.4 История языков программирования.	12
1.5 Исполнение машинных инструкций.	16
Глава 2. Машинная математика.	19
2.1 Основы машинной математики.	19
2.2 Блок-схемы.	21
2.3 Машинная логика и циклы.	23
2.4 Программирование машинной логики.	25

Введение

Эта книга посвящена популярному в нашей стране и перспективному во всём мире языку программирования Delphi. Она направлена на всех программистов, от начинающего, до профессионала. В любом случае, я советую всем прочитать её полностью. Как показывает практика, большинство людей научились программированию по книгам, но ни одна из книг, которые я видел, не объясняет принципиальных основ работы Windows и компьютера в целом. Без понимания этих вещей не возможно написать эффективный код.

Я решил восполнить этот пробел. Я постараюсь написать так, чтобы, прочитав мой труд, любой человек смог стать настоящим программистом. Несмотря на это, я не гарантирую, что именно ты сможешь стать профессионалом. Как показала практика, из всех обучающихся программированию, только 30% становятся настоящими программистами и только к ним относится понятие профессионал. Я обучил достаточно много людей и у меня этот показатель выше 70%. Оставшиеся 30% смогли научиться писать программы, смогли понять основы, но почему-то не обладают способностью самостоятельно мыслить. У них постоянно возникают вопросы, ответы на которые можно получить, затратив всего лишь небольшое усилие. Надо просто немного подумать. Но у них это не получается или не хотят. Может это лень, а может просто человеку не интересно самостоятельно мыслить.

Я могу научить многому, но без стремления самому додумывать то, что я не досказал, ты не сможешь самостоятельно писать программы. В течение всей книги я буду рассказывать различные методы программирования, напишу некоторые шаблоны, покажу некоторые приёмы и хитрости, но описать абсолютно всё я не смогу. Программирование – это такая область, в которой нужно постоянное обучение. Нельзя прочитав только одну книгу останавливаться на достигнутом. Нужно постоянно совершенствоваться и обучаться.

Прежде чем приступить к чтению самой книги, я хочу сделать несколько замечаний. Первое, я буду очень часто использовать в тексте выражение «Язык программирования Delphi». Многие утверждают, что Delphi – это среда разработки, которая использует язык программирования Pascal (Паскаль). В принципе, я не утверждаю, что это ошибка. Но всё же в Delphi от старого Паскаля осталось очень мало, поэтому я считаю, что это не просто среда разработки, это уже целый собственный язык. Это лично моё мнение и ты можешь с ним соглашаться или нет.

Теперь о содержимом книги. Я постарался описать всё так, чтобы было понятно даже человеку, который только недавно познакомился с компьютером. Возможно, продвинутым пользователям большую часть начала книги будет скучно читать, но это только в начале. Практически с самого начала, я начну описывать специфичные вещи, среди которых можно будет найти для себя полезное, даже опытному программисту. Поверь мне, это действительно так и моё утверждение основано на уже сложившейся практике. Это связано с тем, что все книги упускают из виду некоторые очень важные тонкости, которые желательно знать для понимания принципа работы программ. Без этого понимания тяжело двигаться дальше и любые новые технологии будут казаться тяжёлыми и сложными.

Прежде чем ты приступишь к чтению книги, я тебе дам один совет. Книгу желательно читать полностью, от начала и до конца, потому что материал излагается постепенно и некоторые вещи могут быть непонятны если что-то пропустить. Как только ты почувствуешь, что набрал достаточно знаний и способен самостоятельно писать хотя бы простейшие программы, можешь сделать единственный скачок на главу «Дополнительная информация». В ней тебе необходимо прочитать, как отлаживать

приложения, потому что при самостоятельном написании программ всегда появляются ошибки/опечатки. Эта глава рассказывает, как находить такие ошибки. В ней же ты прочитаешь некоторые приёмы по работе с редактором кода, которые тебе могут пригодиться в будущем при программировании собственных приложений, да и при работе с моими примерами.

После прочтения этой главы нужно вернуться на ту главу, на которой ты остановился до этого и продолжить чтения книги без каких-либо скачков. Иначе какой-то важный момент может быть упущен и нагнать потом будет очень тяжело, потому что ты можешь не заметить, что что-то упустил.

Структура книги

В этом месте принято описывать содержание глав книги. Я поступлю так же, чтобы ты мог иметь представление, что я буду описывать в моей книге. К тому же, это поможет тебе легко найти потом интересующую тебя главу, или наоборот, узнать какие главы ты уже хорошо знаешь, и читать не стоит.

Глава 1. «Основные принципы работы компьютера». Эта глава посвящена принципам работы компьютера. В ней я постараюсь рассказать всё, что необходимо знать о том, как компьютер производит расчёты и выполняет различные команды. Эту главу я ещё не видел ни в одной из книг, которые можно купить в магазине (признаюсь, что я не много видел книг). А это же основы, без которых невозможно понимание самого принципа программирования. Конечно же, можно обойтись и без неё, потому что и обезьяну можно научить кидать гранату. Но только с помощью этих знаний, можно понять, что и зачем ты пишешь в своей программе.

В принципе, эту главу можно и опустить, потому что научиться программированию можно и без этого. Но только с пониманием работы железа можно стать настоящим программистом. Поверь мне, эти знания необходимы.

Глава 2. «Машинная математика». В этой главе я постараюсь объяснить тебе основы машинной математики. Это основа программирования. Ты познакомишься с логикой выполнения программ, и сам научишься строить логику будущей программы.

Мы познакомимся с гениальным изобретением всех времён – «блок-схемы». Они очень хорошо помогают начинающим программистам в понимании работы логики компьютера. Конечно же, в будущем ты сможешь научиться писать программы без использования блок-схем, но на начальном этапе это очень удобный инструмент для построения логики будущей программы.

Глава 3. «Оболочка Delphi». В этой главе я опишу процесс установки Delphi 6, расскажу о входящих в поставку утилитах. После этого мы запустим оболочку Delphi 6 и рассмотрим, из чего она состоит. В этой главе будут в основном начальные сведения о Delphi и её могут пропустить те, кто уже знаком с Delphi. Хотя в конце главы я буду говорить о настройках оболочки, поэтому желательно всё же прочесть всем. Возможно, ты что-то и не знал.

Глава 4. «Визуальная модель Delphi». В этой главе я рассказываю про визуальную модель Delphi. На чём построена вся теория программирования в этой оболочке. А так же мы затронем теорию объектно-ориентированного программирования, без понимания которого невозможно движения дальше.

Глава 5. «Основы языка программирования в Delphi». В этой главе мы познакомимся с типами данных Delphi и напишем нашу первую программу. Хотя она будет простой, и в ней не будет содержаться ни строчки кода, я разберу её по косточкам. Мы познакомимся со всеми её внутренностями и узнаем, из чего состоит скелет любой программы на Delphi.

Глава 6. «Работа с компонентами». В этой главе я расскажу все основы работы с компонентами. Я опишу основные свойства, которые можно встретить у большинства объектов. А так же мы познакомимся с событийной моделью Windows, и основными событиями главной формы.

Глава 7. «Палитра компонентов Standard». В этой главе мы познакомимся с закладной *Standard* палитры компонентов. Я распишу все компоненты, для чего они предназначены, и как их использовать. Мы здесь напишем громадное количество примеров с использованием этих компонентов и закрепим всё описанное на практике.

Глава 8. «Учимся программировать». В этой главе я постарался подробно рассказать про циклы, логические операции, работу со строками и многое другое. Это последняя глава, в которой я рассказываю про самые основы программирования. Те, кто уже имеет опыт программирования в Delphi могут пропустить эту главу.

Глава 9. «Создание рабочих приложений». Здесь будет рассказана основа многооконных приложений. Сейчас уже трудно себе представить программу, состоящую только из одного главного окна. Большинство приложений состоит хотя бы из нескольких окон, а некоторые даже из сотен. Здесь же будет описано, как создавать главное меню программы.

Глава 10. «Основные приёмы кodingа». На первый взгляд тут находится сборная солянка. Тут и работа с массивами, файлами, реестром, преобразование данных, структуры и указатели. Всё это собрано под одной крышей потому что я постарался построить книгу так, чтобы ты мог изучать всё последовательно, по мере надобности. Меня просто бесит литература, в которой сначала описывают разные функции и бесполезные примеры (которые в жизни не пригодятся), и только к концу книги находишь что-то действительно полезное. Я даю полезную информацию намного раньше, чтобы изучение программирования не показалось тебе рутинным и скучным.

Глава 11. «Обзор дополнительных компонентов Delphi». После того, как я рассказал об основных приёмах программирования, можно уже рассматривать остальные компоненты Delphi и сразу же писать к ним достаточно полезные в будущем примеры. Если бы я сделал это раньше, то ничего интересного в качестве примеров привести просто не смог бы.

Глава 12. «Графические возможности Delphi». Здесь будет рассказано всё, что касается графики. Я покажу, как можно рисовать встроенными средствами в Delphi различные фигуры и как работать с изображениями разного формата.

Глава 13. «Печать Delphi». Эта глава будет полностью посвящена печати и только печати. Я покажу как выводить на принтер текст и графику, как учитывать разрешение принтера и многое другое.

Глава 14. «Delphi и базы данных». Все знают, что на Delphi очень легко писать базы данных, потому что в него встроены сильнейшие для этого средства. В этой главе тебе предстоит в этом убедиться. Я покажу как работать с локальными базами MS Access и дам множество полезных примеров.

Глава 15. «Создание отчётности». Здесь я покажу, как можно экспортировать данные из твоих таблиц в Excel и подготавливать к печати документы любой сложности.

Глава 16. «Работа с DBF, Paradox и XML базами данных». В этой главе будет рассказано, как работать с другими таблицами, отличными от Access. Здесь будет описана технология доступа к данным через BDE и dbExpress.

Глава 17. «Потоки». Windows – многозадачная система и позволяет писать многопоточные приложения, в которых операции выполняются параллельно. В этой главе ты познакомишься с многопоточностью и напишешь примеры.

Глава 18. «Динамические библиотеки». В этой главе будет рассказано всё необходимое, что касается динамических библиотек. Ты увидишь, как создавать библиотеки с математическими процедурами и функциями, как хранить окна в библиотеках и увидишь реальные примеры их использования.

Глава 19. «Разработка собственных компонентов». В этой главе пойдёт рассказ о том, как создавать свои VCL компоненты, как устанавливать чужие разработки в Delphi и как работать с пакетами компонентов.

Глава 20. «Мультимедиа». Эта глава полностью посвящена принципам программирования звука и видео. Я покажу в ней, как создавать приложения для работы со звуком с использованием встроенных в Delphi компонентов и без них.

Глава 21. «Графика OpenGL». Есть две достаточно перспективные разработки для профессиональной работы с компьютерной графикой – OpenGL и DirectX. Я решил достаточно подробно описать в этой книге только OpenGL, а по DirectX ты сможешь найти немного начальной информации на компакт диске в директории «Документация».

Глава 22. «OLE, COM, ActiveX». В этой главе будут описаны основные принципы технологий OLE, COM и ActiveX. Все эти термины взаимосвязаны и должны описываться вместе. Я не очень люблю эти технологии, но описать обязан, потому что иногда мне приходится работать с ними и возможно, что и ты когда-нибудь столкнёшься с этой технологией.

Глава 23. «Буфер обмена». Кнопки «Копировать» и «Вставить» есть практически в любом полноценном приложении. Я думаю, что ты тоже захочешь вставить такую возможность в свои программы. В этой главе я дам максимум полезной теоретической и практической информации, чтобы ты смог сделать свои программы более привлекательными, добавив возможность переноса данных между приложениями.

Глава 24. «Дополнительная информация». Эта глава единственная, которую ты можешь прочитать вне очереди. Как только ты почувствуешь, что твоих знаний достаточно для написания собственных небольших приложений, то ты можешь перескочить на эту главу. Здесь будут описаны некоторые приёмы работы с оболочкой Delphi, которые смогут тебе помочь при разработке собственных приложений, а так же принципы тестирования и отладки твоих программ.

Глава 25. «Сплошная практика». Напоследок я опишу несколько интересных программ, чтобы ты мог увидеть некоторые приёмы программирования, которые могут пригодиться тебе в будущем. Эту главу можно рассматривать как дополнительный материал ко всему сказанному выше.

Глава 1. Основные принципы работы компьютера

1.1 Основы работы персонального компьютера.

Прежде чем программировать компьютер, мы должны понять, как он работает. Как говорил какой-то полководец: «Нужно хорошо изучить своего врага!!!». Возможно, это говорил и не полководец, но это не важно :). Кодинг – это постоянная борьба с машиной. Нужно заставлять её делать то, что тебе нужно. Поэтому любой программист просто обязан знать его внутренности.

Компьютер состоит из следующих основных компонентов: процессор, память, видеокарта, винчестер (жесткий диск) и различные разъёмы для подключения дополнительных устройств. Все эти компоненты связаны между собой с помощью шлейфов и шин.

Вся информация в компьютере хранится на винчестере. Когда ты запускаешь какую-нибудь программу, то она сначала загружается в память и только потом процессор начинает выполнять содержащиеся в ней инструкции. Чем больше программа, тем дольше она загружается.

Результат работы программы выводится на экран через видеокарту. На любой видеокарте есть чип памяти, в котором отображается всё содержимое экрана. Когда тебе нужно вывести что-то на экран, ты просто копируешь эти данные в видеопамять, и видеокарта автоматически выводит его содержимое на монитор.

Это всё, что необходимо знать о работе компьютера. Пока я описать только общие черты, а в следующих разделах я опишу необходимые вещи более подробно. В основном нас будет интересовать работа процессора, поэтому ему я уделю особое внимание. Остальное пока достаточно знать в общих чертах.

1.2 Двоичная система работы процессора.

Компьютеры изобрели достаточно давно. В те времена электроникой даже и не пахло. Первые компьютеры были ламповыми и занимали очень много места. Для того, чтобы управлять такой машиной нужно было очень много обслуживающего персонала.

Уже тогда был заложен принцип работы компьютера, который действует до сих пор. А именно, данные передаются с помощью какого-то сигнала (для нас не имеет значения какого, потому что мы не электронщики) методом «есть сигнал или нет» или по-другому «включён или выключен». Так появился «бит» bit. Бит это единица информации, которая может принимать значение или 0, или 1, т.е. «включён или выключен». Восемь бит объединяются в байт, т.е. один байт равен 8 битам. Почему именно 8? Да потому что первые компьютеры были восьмью разрядными и могли работать одновременно только с 8-ю битами, например, 010000111.

Немного позже ты узнаешь, что в один байт можно записать любое число до 255. Но это очень мало, поэтому чаще используют более крупные градации:

1. Два байта = слово.
2. Два слова = двойное слово.

Итак, компьютер стал работать в двоичной системе исчисления. Но как же тогда записать число 135, если у нас единица информации может быть только или 0 или 1. Просто в двоичной системе. Давай разберёмся, как это работает.

Для начала вспомним, как работает наша десятичная система исчисления, к которой мы привыкли. Для этого рассмотрим число 519578246. Я специально выбрал такое число, чтобы оно состояло из восьми разрядом. Теперь запишем его, как на рисунке ниже:

Номер разряда	8	7	6	5	4	3	2	1	0
	5	1	9	5	7	8	2	4	6

Как видишь, я пронумеровал разряды, начиная с нуля до восьми, и справа налево. Теперь представь себе, что это не целое число, а просто набор разрядов. 5, 1, 9, 5, 7, 8, 2, 4 и 6. Как из этих разрядов получить целое число? Наверно некоторые скажут, что надо просто записать их подряд. А если я спрошу, почему? Вот тут появляется математика. Нужно каждый разряд умножить на 10 (степень исчисления) возведённую в степень номера разряда. Непонятно? Попробую оформить в виде формулы:

$$\text{Разряд}_0 * (10^{\text{Номерразряда}}) + \text{Разряд}_1 * (10^{\text{Номерразряда}}) + \dots$$

Давай посчитаем по этой формуле, начиная с нулевого разряда. Получается, что 6 нужно умножить на 10 в нулевой степени $6*10^0=6$. Потом прибавить $4*10^1=40$ (итого уже 46). Потом $2*10^2=200$ (итого 246). Потом $8*10^3=8000$ (итого 8246) и так далее. В итоге получится число 519578246.

А теперь рассмотрим двоичную систему. Здесь каждый разряд может быть или 0 или 1 (2 состояния). Кстати, в десятичной системе у нас каждый разряд мог быть от 0 до 9, то есть десять состояний. Давай рассмотрим следующий байт - 010000111. Запиши его на листке бумаги так, как показано на рисунке ниже.

Номер разряда	8	7	6	5	4	3	2	1	0
Биты	0	1	0	0	0	0	1	1	1

Здесь действует та же самая формула, только нужно возводить в степень не 10, а двойку. Опять же произведём расчёт, начиная с нулевого разряда, т.е. справа налево. Получается, что первую 1 мы должны умножить на 2 в нулевой степени ($1*2^0=1$). Следующую единицу нужно умножить на 2^1 получается 2 (итого $2+1=3$) и т.д. Вот как это будет выглядеть полностью:

$$(1*2^0)+(1*2^1)+(1*2^2)+(0*2^3)+(0*2^4)+(0*2^5)+(0*2^6)+(1*2^7)+(0*2^8)=135.$$

Вот так, оказывается, выглядит в двоичной системе число 135. Давай теперь научимся пересчитывать числа из десятичной системы в двоичную систему. Для этого нужно число 135 разделить на 2. Получается 67 и остаток 1 (запомним 1). Теперь 67 делим на 2, получается 33 и остаток 1 (теперь две единицы, т.е. 11). Теперь 33 делим на 2, получаем 16 и остаток 1 (теперь три единицы, 111). Теперь 16 делим на 2, получаем 8 и остаток 0 (всего 0111). Теперь $8/2=4$ и остаток 0 (00111). $4/2=2$ и остаток 0 (000111). Теперь $2/2=1$ и остаток 0 (итого 0000111). 1 на два не делится, значит, просто дописываем её 10000111. Получилось первоначальное число.

Вот так происходит преобразование чисел. В двоичную систему исчисления. Таким же образом можно перевести число в любую систему (двоичная, восьмеричная,

шестнадцатеричная и т.д). Для более полного закрепления материала я решил привести таблицу, в которой показаны соответствия десятичных чисел двоичным. Попробуй сам перевести пару чисел туда и обратно.

Десятичное	Двоичное
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

Таблица 1. Таблица соответствия десятичных и двоичных чисел

В компьютере принято вести расчёт в двоичной или шестнадцатеричной системе. Вторая вошла в обиход, когда компьютеры стали 16-и разрядными.

Шестнадцатеричная система выглядит немного по-другому. Каждый разряд уже содержит не 2 состояния (как в двоичной) или десять (как в десятичной), а шестнадцать. Поэтому один разряд может принимать значения от 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Буква «A» соответствует 10, «B» соответствует 11 и т. д. Например, число 1A в шестнадцатеричной, равно 26 в десятичной. Почему? Да по всё то же формуле. Только здесь нужно возводить 16 в степень номера разряда. «A» - это десять, нужно умножить на $16^0 = 10$. 1 – первый разряд нужно умножить на $16^1 = 16$. $10 + 16 = 26$.

Десятичное	Двоичное	Шестнадцатеричное
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14

Таблица 2. Таблица соответствия десятичных, двоичных и шестнадцатеричных чисел



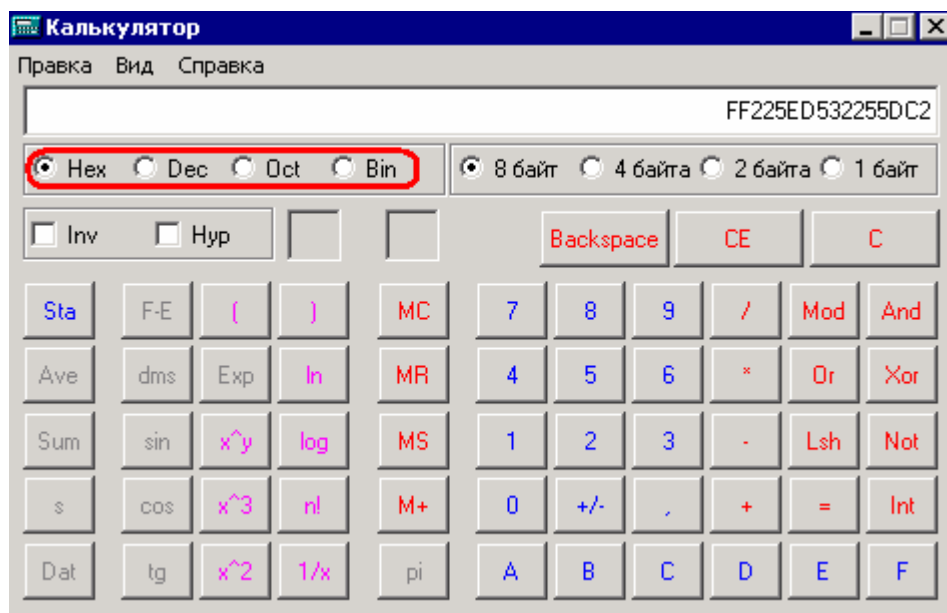
На протяжении всей книги мы будем иногда встречаться с шестнадцатеричной системой исчисления (без этого никуда не денешься), поэтому, когда нужно будет показать, что число шестнадцатеричное, я буду ставить перед ним знак решётки #, например, #13. В других языках, например Assembler или C++ принято ставить в конце числа букву h, например, 13h. Но эта книга о Delphi, поэтому я буду писать так, как принято в этой среде разработки, чтобы потом не возникало никаких проблем.

Но это всё целые числа. С числами с плавающей точкой совершенно другая история. Если заранее предусмотрено, что число может быть отрицательным, то его длина сокращается ровно на один бит. Если неотрицательное целое число может быть 8-ми битным, то число со знаком будет 7-и битным. Первый бит будет означать знак. Если первый бит равен 1, то число отрицательное, иначе положительное.

В дробных числах один байт может быть отведён для целой части и один для дробной. Никогда не смешивают целую и дробную часть в одно целое. За счёт этого, дробные числа всегда будут занимать больше памяти, и операции с ними проходят намного дольше.

На первый взгляд перевод чисел очень сложный, но вручную им пользоваться не обязательно. Человек уже давно придумал для себя хорошего помощника - калькулятор. С его помощью без проблем можно перевести число в любую систему.

Запусти встроенный в Windows калькулятор (Пуск -> Программы -> Стандартные -> Калькулятор). Теперь выбери из меню «Вид» пункт «Инженерный». На рисунке ниже показано окно, которое ты должен увидеть:



Внешний вид калькулятора

Для перевода числа в другую систему, просто набери его и потом выбери нужную систему исчисления. На рисунке я обвёл красным цветом кнопки переключения системы исчисления:

- Нех – шестнадцатеричная.
- Дес – десятичная.
- Oct – восьмеричная.
- Bin – двоичная.

Возникает вопрос – зачем я тогда так долго рассказывал о преобразованиях, когда так легко воспользоваться калькулятором? Ответ прост – НАДО. Поверь мне. Если ты будешь понимать, как происходит преобразование, то тебе потом легче будет работать с этими числами.



Никогда не полагайся только на технику. Всегда полезно знать, как и зачем она что-то делает. Если ты разберёшься с шестнадцатеричным представлением данных, то сможешь простые преобразования делать в уме. Ну а если ты ещё и собираешься стать хакером, то тебе просто необходимо научиться хорошо оперировать разными системами исчисления.

1.3 Машинный язык.

Данные на диске также хранятся в двоичном виде. Даже текстовые файлы на диске выглядят в виде нулей и единиц. Точно так же выглядит и любая программа, только её называют машинным кодом. Давай с ним познакомимся немного поближе.

Любая программа представляет собой последовательность команд. Эти команды называются *процессорными инструкциями*. По этим инструкциям процессор определяет, что и как ему нужно делать. Когда ты запускаешь программу, компьютер загружает её машинный код в память и начинает выполнять. Наша задача, как программистов написать эти инструкции, чтобы компьютер понял, что мы от него хотим.

Реальная программа, которую выполняет компьютер, представляет собой последовательность единиц и нулей. Такую последовательность называют машинным языком. Но человек не способен эффективно думать единицами и нулями. Для нас легче воспринимается осмысленный текст, а не сумасшедшие числа в двоичной системе измерения, с которой мы не привыкли работать. Например, команда складывания двух регистров выглядит так: #03C3. Нам это мало о чём говорит, и запомнить такую команду очень тяжело. На много проще написать «сложить число1+ число2».

Первое время программисты писали в машинных кодах, пока кому-то не пришла в голову идея: «Почему бы не писать текст программы на понятном языке, а потом заставлять компьютер переводить этот текст в машинный код?». Идея действительно заслуживала внимания. Так появился первый компилятор – программа, которая переводила текст программ в машинный код.

Вот тут, я думаю надо сделать паузу и рассказать тебе небольшую историю языков программирования. Она достаточно интересна и поучительна. Ну а потом мы продолжим изучения принципов работы компьютера и познакомимся с содержимым процессора и его работой.

1.4 История языков программирования.

Как мы уже выяснили, компьютер - примитивное существо, которое мыслит нулями и единицами, из которых складываются числа. Так что все, что может делать процессор, так это оперировать этими числами. Так и программы - это тоже числа, которые воспринимаются процессором как команды к выполнению каких-то действий.

Мы также выяснили, что первые программисты писали программы в машинных кодах. Тогда еще не было компиляторов и приходилось все писать числами. Ты даже представить себе не можешь, какой это адский труд. Постоянно держать в памяти таблицу машинных кодов - это тебе не таблица умножения. Например, тебе понятно число 8BC3. Нет? А это простая команда копирования между двумя ячейками регистров. Это просто пример, потому что тогда регистры были другие и процессоры были на много проще.

Со временем компьютер стал уметь. Он все так же оперировал числами, но делал это намного быстрее. Но программист - это человек, а не железка и ему очень тяжело создавать логику в числах. Намного легче работать с привычными словами. Например, все ту же команду удобней записать словами типа "скопировать ebx в eax". Но что делать, если компьютер не понимает слов, а только числа? Выход есть - написать такую программу, которая будет превращать текст в машинные коды. Пусть компьютер сам создает байт-код. Такую программу называли компилятором. А язык, на котором писался текст программы, называли языком программирования.

```
MainUnit.pas.131: InitViewInit;
00527F7C 8BC3          mov eax,ebx
00527F7E E84D010000    call TSborDataForm.InitViewInit
MainUnit.pas.132: InitProgramm;
00527F83 8BC3          mov eax,ebx
00527F85 E8B6030000    call TSborDataForm.InitProgramm
MainUnit.pas.133: if lDeviceNum>=0 then
00527F8A 83BB1004000000 cmp dword ptr [ebx+$00000410],$00
00527F91 7C26          jl +$26
MainUnit.pas.135: ScanPortThr := TScanPortThread.Create(true);
00527F93 B101          mov cl,$01
00527F95 B201          mov dl,$01
00527F97 A1D4A45200    mov eax,[ $0052a4d4]
00527F9C E8EB94EFFF    call TThread.Create
00527FA1 8BF0          mov esi,eax
00527FA3 89B314040000 mov [ebx+$00000414],esi
MainUnit.pas.136: ScanPortThr.lDeviceNum:=lDeviceNum;
00527FA9 8B8310040000 mov eax,[ebx+$00000410]
00527FAF 89464C        mov [esi+$4c],eax
MainUnit.pas.137: ScanPortThr.Resume;
00527FB2 8BC6          mov eax,esi
00527FB4 E81F98EFFF    call TThread.Resume
MainUnit.pas.141: try
00527FB9 33C0          xor eax,eax
00527FBB 55            push ebp
00527FBC 680B805200    push $0052800b
00527FC1 64FF30        push dword ptr fs:[eax]
```

Программа в машинных кодах и Assembler

И вот был написан первый компилятор. Эту программу называли Assembler, что переводится, как "сборщик". Писать на нем практически так же, как и в машинных кодах, только теперь уже использовались не числа, а понятные человеку слова. Например, все та же команда копирования регистров теперь выглядела так: "mov eax, ebx". То есть цифры заменились на понятные слова.

Вроде все прекрасно и удобно, но почему-то среди программистов возникли споры и разногласия. Кто-то воспринял новый метод с удовольствием. А кто-то говорил, что машинные коды лучше. Любители языка Assembler хвалили компилятор за то, что

программировать стало проще и быстрее, а противники утверждали, что программа, написанная в кодах, работает быстрее. Говорят, что эти споры доходили до драк и иногда лучшие друзья становились врагами. А в принципе, и те и другие были правы. На языке Assembler действительно программу писать легче и быстрее, а в машинных кодах программа работала быстрее.

Тогда никто не мог себе представить, чем же все может закончиться. Но время показало свое. С помощью Assembler программы писались быстрее, а это один из основных факторов успеха любой программы на рынке. Люди начинают пользоваться тем продуктом, который выходит на рынок первым. Даже если более поздний вариант лучше, человека трудно переубедить перейти на другую версию. Здесь играет большую роль фактор привычки. К тому же, к тому моменту, когда программист напишет свою первую версию в машинных кодах, программист на языке Assembler выпустит уже пару новых версий своего шедевра.

Вот так и получилось, что те, кто программировал на языке Assembler превратились в убегающих вперед, а те, кто программировал в машинных кодах превратился в вечно догоняющих. В конце концов, первые убежали на столько, что вторые не смогли догнать, и вынуждены были или перейти на Assembler или отойти от программирования на совсем.

Вот тут начался бум. Языки программирования стали появляться один за другим. Так появились C, ADA, FoxPro, Fortran, Basic, Pascal и другие. Некоторые из них были предназначены только для детей, а некоторые и для профессиональных программистов. И тут споры перенесли в другую плоскость - какой язык лучше. И этот спор длится уже около 30 лет и конца ему не видно. Некоторые говорили, что это Pascal, другие утверждали что C, ну а кое-кто утверждал что это Visual Basic. Этот спор разделился на две части:

1. Какой язык самый лучший?
2. Что лучше - язык высокого уровня или низкого?

Первый спор не может закончиться до сих пор. Каждый пытается доказать, что его язык программирования самый могучий, удобный и создаёт самый быстрый код. Мне кажется, что этот спор не закончится никогда. В принципе, меня это устраивает, потому что это своеобразная конкуренция. Благодаря ей происходит развитие и мы летим вперед.

Так все же, какой язык лучше? На этот вопрос я дам ответ, но только немного позже.

Наиболее интересным был спор: "Что лучше - язык высокого уровня или низкого?". Язык низкого уровня это тот, который наиболее приближен к командам процессора, то есть Assembler. К языкам высокого уровня относят C, Pascal, Basic и др. Этот спор проходил в той же манере, как и спор между любителями Assembler и любителями программирования в машинных кодах. Только теперь приверженцы Assembler утверждали, что их код самый быстрый, а любители языков высокого уровня утверждали, что они напишут программу быстрее, чем самый лучший программист на языке Assembler.

Спор продолжался достаточно долгое время. И опять победила скорость разработки и удобство языка программирования. Любителям Assembler пришлось отступить, потому что теперь они превратились в «догоняющих», и не смогли угнаться за языками высокого уровня.

Конечно же, нельзя сказать, что машинные коды и Assembler на совсем ушли из нашей жизни. Они используются до сих пор, но в очень ограниченном количестве. Язык Assembler используется только в качестве вставок для языков высокого уровня, а машинные коды используются для написания того, чего нельзя сделать компилятором (да и для написания самого компилятора они нужны). Ушедшие технологии живут, и будут жить, но рядовой программист очень редко встречается с ними.

Следующей ступенью стало объектно-ориентированное программирование. Язык С превратился в C++, Pascal превратился в Object Pascal и так держать. И снова борьба. И снова скорость разработки против быстроты кода. Опять споры, драки и оскорбления.

Война длилась несколько лет. Сколько времени было потрачено в спорах, сколько волос было вырвано на голове в процессе доказательств крутизны именно его кода. А результат - победила скорость и удобство разработки, т.е. объектно-ориентированное программирование (ООП).

Последней крупной революцией происходящей в программировании я считаю переход на визуальное программирование. Этот переход происходит прямо на наших глазах. Визуальность дает нам еще более удобные средства разработки для более быстрого написания кода, но проигрывает ООП по скорости работы. Вот многие начинающие и стоят на перекрестке, какой язык выбрать.

Лидеров в визуальных языках является Borland, а приверженцем ООП остается Microsoft. Конечно же, Билл Гейтс пытается встроить в свои языки визуальность, но она примитивна по сравнению с такими гигантами, как Delphi, Kylix или C++ Builder. Это связано с изначальной дырой MFC, которая не может работать визуально. Нужна глобальная переработка кода, которую почему-то не хотят делать. Вот народ и стоит на двух атомных бомбах и ожидает взрыва одной из них. Как ты думаешь, какая бомба рванет? Что победит - скорость разработки или скорость кода? Я не буду отвечать на этот вопрос. История говорит сама за себя, а мы подождем подтверждения этому.

Я считаю, что прогресс не будет стоять на месте и переход на новые технологии программирования рано или поздно состоится. Поэтому я уже перешел на Delphi. Если ты хочешь успеть за прогрессом, то ты тоже обязан вступить в партию любителей Borland. Выбирай любой из его компиляторов, и ты не ошибешься. Для тебя есть все, что угодно Delphi, JBuilder, Kylix или C++ Builder. Как видишь у Бормана есть визуальные варианты всех языков, и они действительно лучшие.

Я уже сказал, что самая лучшая технология - визуальность. Твоя среда разработки просто обязана быть визуальной, потому что за этим наше будущее. Если ты хочешь получить визуальность + мощь разработки, то твоя среда от Borland. С помощью языков этой фирмы можно сделать абсолютно все. Так что с этим мы покончили. Вердикт окончательный и обжалованию не подлежит.

Мне осталось только ответить на вопрос: "Какой язык программирования лучше?". Я уже несколько лет пытаюсь ответить на этот вопрос, но окончательного решения вынести не могу. Даже у того же Visual C++ от Microsoft есть свои плюсы. Как это не странно, но положительные стороны есть у всех. Вопрос остается только за тем, что ты будешь писать? Я могу дать примерно такую градацию:

1. Если ты будешь писать базы данных, программы общего значения или утилиты, то твой язык Delphi или C++ Builder.

2. Если это игры, то желательно Visual C++ или Watcome C плюс знание Assembler. Но это не значит, что нельзя использовать Delphi или C++ Builder. В этих средах ты потеряешь не намного больше в скорости работы, поэтому на большинстве игр можно не обращать внимания на эту потерю.

3. Если это будут драйверы и работа с железом, то тут критичен размер файла, а значит твой язык чистый C или Assembler.

И все же большую массу программ занимают утилиты и базы данных. А тут визуальность необходима, если ты хочешь оказаться впереди. Визуальные языки будут жить и за ними будущее. И на протяжении всей этой книги я буду тебе рассказывать про самый лучший (это на мой взгляд, и он может отличаться от других) - Delphi.

1.5 Исполнение машинных инструкций.

Прежде чем переходить дальше, я должен познакомить тебя с несколькими понятиями:

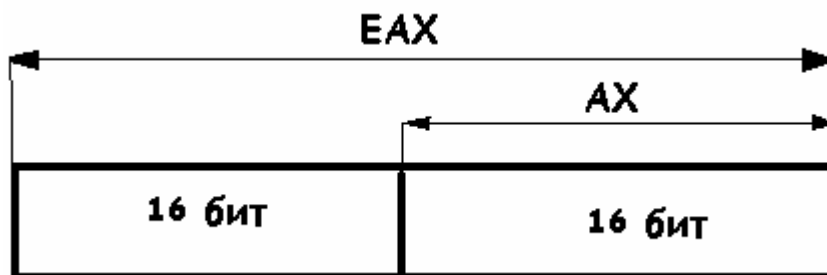
Сегмент – это просто область памяти. Раньше, когда операционные системы (ОС) были 16 битными, процессор не мог работать с памятью размером более 64 килобайт (это максимум, что можно записать в два байта). Поэтому память делилась на сегменты по размеру и по назначению. На данный момент мы используем 32-ю ОС, которая может адресовать до 4 Гбайт оперативной памяти. Поэтому можно сказать, что память стала сплошной. Но деление по назначению всё-таки осталось. Существуют следующие сегменты памяти:

- **Сегмент кода** – в эту область памяти загружается машинный код, который будет потом выполняться процессором.
- **Сегмент данных** – это область памяти для хранения данных.
- **Сегмент стека** – область памяти для хранения временных (локальных) данных и адресов возврата из процедур.

Каждой запущенной программе отводится свой сегмент кода, данных и стека. Поэтому данные одной программы не могут пересекаться с данными или кодом другой программы, если конечно же не произошёл сбой.

Регистр – ячейка памяти в процессоре. Размер ячеек зависит от его разрядности. В 32-х разрядных процессорах ячейки 32-битные. Мы будем говорить о 32-х разрядных процессорах, а значит и о 32-х битных регистрах. Таких ячеек там несколько и каждая из них предназначена для определённых целей. Один регистр состоит из двух ячеек, значит в 32-битном процессоре регистр равен $2 \times 32 = 64$ бит.

Когда компьютер был ещё 16 битным, все регистры были тоже 16-и битными. С появлением 32-й платформы размер ячейки регистра тоже увеличился, до 32 бит. Но для совместимости со старыми программами они как бы делятся на две части (те же две ячейки). Первая – это тот старый регистр, а вторая – дополнительные 32 бит. На словах не совсем понятно, поэтому давай посмотрим на рисунок.



На этом рисунке показан регистр EAX. Полная его длина – 32 бита, но младшая половина – это регистр AX (16 битный вариант регистра). То есть, если мы попросим процессор показать нам содержимое регистра AX, то мы увидим половину регистра EAX. Иногда это очень даже удобно, особенно когда тебе надо прочитать только половину числа из регистра.

Теперь реальный пример. Допустим, в регистре EAX находится шестнадцатеричное число #21CD52B, тогда в регистре AX будет находиться последние 16 бит, а именно D52B.

Сейчас я начну описание регистров, и если его имя начинается с буквы «Е», то это значит, что он 32-битный и для него существует и 16-и битный вариант без буквы «Е».

Сегментные регистры - CS, DS, SS и ES. (есть ещё, но нас пока интересуют только эти):

- **Регистр CS** - регистр сегмента кода, в нём хранится начальный адрес сегмента кода.
- **Регистр DS** - регистр сегмента данных, в нём хранится начальный адрес сегмента данных. В этом сегменте располагаются глобальные переменные.
- **Регистр SS** - регистр сегмента стека, в нём хранится начальный адрес сегмента стека. Здесь располагаются локальные переменные.
- **Регистр ES**. Для использования дополнительного сегментного регистра.

Разницу между глобальными и локальными переменными мы рассмотрим чуть позже, когда будем изучать сам язык программирования. Сейчас я ещё скажу только то, что в сегменте сохраняются и переменные, которые передаются в процедуры и адрес возврата из процедуры (куда нужно вернуться по окончании выполнения кода процедуры).

Регистры общего назначения EAX, EBX, ECX и EDX, все эти регистры 32-х битные. В 16-разрядных процессорах, они были 16-разрядными и назывались AX, BX, CX и DX. Они могут использоваться в программе по собственному усмотрению, но в некоторых случаях им отведена определённая роль. В регистр EAX в основном записываются результаты арифметических вычислений, а регистр ECX используется в качестве счётчика. Регистр EAX используется для хранения результатов вычисления.

Очень часто, прежде чем выполнить какую-то команду, процессор загружает необходимые данные в регистры и только после этого выполняет необходимую инструкцию. Но возможны варианты, когда вычисления идут напрямую с памятью.

Регистровые указатели ESP и EBP. ESP - это указатель стека, который обеспечивает его использование в памяти. EBP — обеспечивает доступ к данным и указателям на данные переданные через стек.

Индексные регистры ESI и EDI. Эти регистры используются при сложении и вычитании, а так же для расширенной адресации.



Если ты собираешься писать простенькие утилиты или базы данных, то эти знания ты не будешь использовать. Но если ты хочешь пойти дальше, то желательно не просто знание, но и понимание процесса работы процессора. Поэтому я сейчас попробую на пальцах (а точнее сказать на примере) объяснить процесс его работы.

Итак, сейчас я опишу процесс выполнения программы более подробно. В любом случае, это тебе пригодится.

При старте программы, исполняемый код загружается в сегмент кода. Регистр CS сразу устанавливается в значение, указывающее на начало этого сегмента. Данные программы загружаются в сегмент данных (это константы и любые другие дополнительные данные). На этом же этапе происходит подготовка (инициализация) к работе сегмента стека. Если программе переданы какие-нибудь значения, то они автоматически заносятся в стек.



Сегменты кода содержит код только одной программы. В один сегмент не может быть загружен код двух абсолютно разных программ. Точно так же и сегмент данных, и сегмент стека. При каждом старте новой программы,

операционная система отводит её свои собственные сегменты кода, данных и стека.

После этих подготовительных действий, ОС готова к выполнению кода. Напомню, что регистр CS указывает на начало сегмента кода. Есть ещё один регистр, о котором я ещё не сказал – EIP. Этот регистр указывает на текущую выполняемую команду в сегменте кода.

Процессор последовательно выполняет все команды, находящиеся в сегменте кода. Иногда необходимо перескочить не на следующую точку программы, а совершенно в другое место. В этом случае происходит такой переход, и команды начинают последовательно выполняться, уже начиная с новой точки.

Я уже сказал, что любые операции, вычисления могут производиться с регистрами и с памятью. Например, к значению регистра EAX прибавить значение EBX – это сложение регистров. Можно складывать и значения находящиеся в оперативной памяти. Но надо помнить, что вычисления с регистрами происходит намного быстрее, потому что регистр – это та же оперативная память, только находящаяся в процессоре. Если ты собираешься произвести с одним и тем же числом две операции, то намного эффективнее будет располагать его в регистре.



Глава 2. Машинная математика.

До перестройки, в нашей стране практически не обучали программистов. Большинство программистов были выходцами с кафедр математики, на которых очень часто были какие-то предметы с уклоном в сторону информатики. На этих курсах учили писать блок-схемы – это схема, описывающая логику программы.

Я с блок-схемами познакомился на первом курсе института. Первое впечатление – полное фуфло. Но со временем я понял их достоинства. Возможно, что тебе покажется это слишком просто, но всё же желательно прочитать эту главу полностью. Здесь я расскажу теорию всего процесса программирования. В дальнейшем нам останется только познакомиться с практикой, и мы на коне ☺.

2.1 Основы машинной математики.

На любом языке программирования можно выполнять математические операции любой сложности. Delphi не исключение. Но пока что мы не будем рассматривать все возможности, а остановимся только на основных. Вот основные математические операции языка Delphi:

Математическая операция	Описание
*	Умножить
/	Разделить
Sqr	Квадрат
Sqrt	Квадратный корень
+	Сложение
-	Вычитание
:=	Присвоить значение.

Математические операции

В таблице перечислены основные математические операции языка программирования Delphi. Они выполняются в том же порядке, в котором перечислены. Например, в формуле $2+2*2$ результатом будет 6, потому что сначала выполняется операция умножения, а потом сложения. Если ты хочешь сначала выполнить сложение, а потом вычитание, то как и в простой математике должен использовать скобки: $(2+2)*2=8$. В этом случае результат уже будет совершенно другим.

Для изучения компьютерной математики ты должен знать следующие понятия:

Переменная – это память, в которую можно записывать различные значения. Чаще всего этой памяти присваивается в соответствие имя. Например, я завожу переменную с именем F. Ей я могу присваивать значения, например 5. Для этого мне нужно записать $F:=5$. Знак двоеточие + равно означает операцию «присвоить».

Значения переменных можно копировать из одной в другую. Допустим, что у меня есть ещё одна переменная G. Я могу присвоить ей значение переменной F с помощью простого присваивания $G:=F$. После этого в переменной G у меня тоже будет значение 5.

Переменной можно присваивать результаты каких-то вычислений, например: $F:=10/2$. Это достаточно простой пример. А вот уже целое выражение с использованием переменных:

```
F:=5;  
G:=10;  
F:=G/2;
```

Имя переменной может состоять как из одной буквы, так и из нескольких букв. Например, имя переменной может быть: Str, или MyPeremen. Единственное ограничение – имя должно состоять из английских букв и не должно использовать зарезервированные слова (о зарезервированных словах немного позже). Ты так же можешь в имени использовать числа (желательно в конце), например Str1, Str2, Str3 и так далее.



Мой тебе совет, назначай переменным осмысленные имена. Когда ты начнёшь писать большие программы, тяжело будет разобраться, что означает переменная i или b. Желательно давать более осмысленные имена. В течении всей книги я постараюсь тебя к этому приучить.

Тип переменной – тип значения, которое можно записать в переменную (в память). Очень часто используют термин «*Тип данных*», потому что это действительно тип данных хранящихся в переменной. Он показывает, какого типа информация находится в конкретной переменной. В Delphi принято обязательно указывать типы переменных, чтобы сразу можно было увидеть какую информацию можно туда записать.

Существует несколько основных типов переменных:

Название типа	Описание	Дополнительная информация
Integer	Целое число	Переменная этого типа может принимать в качестве значения любые целые числа, как положительные, так и отрицательные.
Real	Вещественное число	Переменная этого типа может принимать в качестве значения целые и дробные числа со знаком и без.
String	Строка	Переменная этого типа может принимать в качестве значения любые символы и наборы символов.
Boolean	Булево значение	Может принимать значение true или false (истина или ложь). Этот тип очень часто используется для организации логики.

Основные типы данных в Delphi

Это только основные типы. Реально их намного больше. Когда мы перейдём к программированию, я познакомлю тебя с большим количеством типов данных.

Строки – любые символы и наборы символов. В языке Delphi они выделяются одинарными кавычками, например, 'Привет'. Строки так же можно присваивать переменным, как и любое другое значение.

Str – строковая переменная.
Str:='Привет!!!'

На этом основные сведения о машинной математике подошли к концу. Пора применить наши знания на практике.

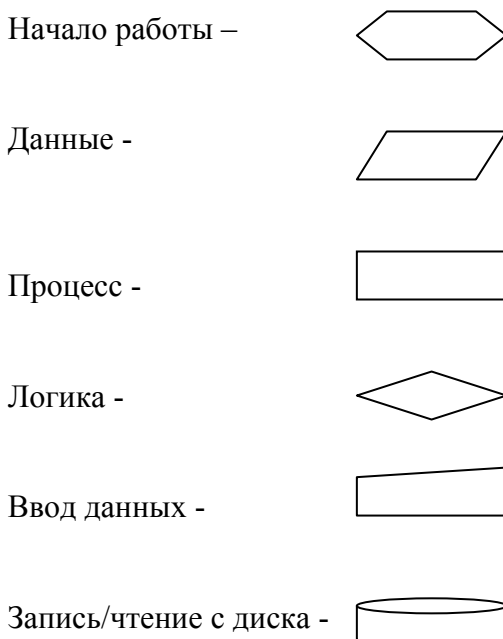
2.2 Блок-схемы.

Давай сразу зададим какой-нибудь простой пример, на котором попробуем расписать логику его решения. Допустим, нам надо получить произведение двух чисел. В человеческой логике мы должны выполнить следующие операции:

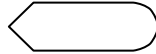
1. *Старт.*
2. *Ввести число.*
3. *Ввести число 2.*
4. *Умножить число 1 на число 2.*
5. *Вывести результат.*

Простейшая и подробная логика, которой оперирует человек. Но машина немного сложнее и в её логике нужно рассуждать немного по-другому. Для отображения машинной логики удобнее перечисления шагов не удобно, поэтому давай знакомится с блок схемами на этом примере.

Блок схемы принято чертить различными квадратами, овалами и прямоугольниками. Я особо не буду придергиваться стандартов, потому что это не особо имеет значения, но некоторых особенностей буду придергиваться. Вот основные типы блоков используемых мной:

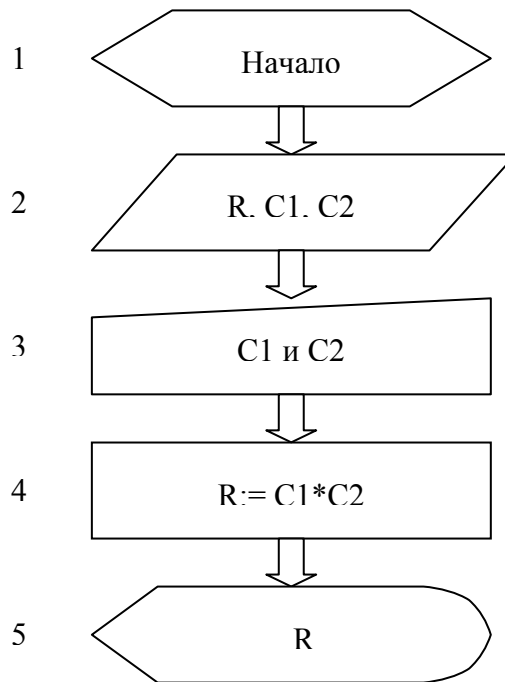


Вывод на экран -



Есть и другие, более извращённые блоки, но я ими не буду пользоваться. Большинство блоков я буду оформлять, как просто прямоугольник, потому что от формы блока суть особо не изменится. Самое главное (на мой взгляд) выделить отдельным видом блока начало блок-схемы и логику. Всё остальное можно оформить однообразно, наглядность от этого пострадает, но не сильно.

Итак, наша первая блок-схема, умножения 2-х чисел будет выглядеть так:



На первый взгляд всё слишком сложно. Но это только первый взгляд. Реально здесь ничего сложного нет, просто очень громоздко и простейшая операция перемножения превращается в несколько операций. Но всё же надо объяснить происходящее поподробнее, чтобы мы могли продвинуться дальше и разобраться с более сложными примерами.

Первый блок – это начало. Если ты соберёшься строить блок-схемы, то обязательно указывай его, чтобы сразу можно было увидеть, начало логики.

Второй блок – перечисляет переменные, которые нам нужны для вычислений. Я использую три переменные R, C1 и C2. В переменную R будет помещён результат вычисления. Переменные C1 и C2 используются для хранения введённых данных. Пока я не указываю тип данных хранящихся в переменных, но подразумеваю, что это будут или целые числа, или вещественные.

Третий блок – здесь показывается, что надо ввести значения переменных C1 и C2.

Четвёртый блок – здесь показывается, на необходимость произвести умножения C1 на C2 и результат записать в переменную R.

Пятый блок – вывод результата на экран.

Это простая блок-схема, в которой нет ничего особенного, и ты даже не можешь увидеть всех её преимуществ. Следующие примеры будут уже использовать логику, поэтому блок-схемы будут более сложными, и ты сможешь ощутить всю прелесть машинной математики.

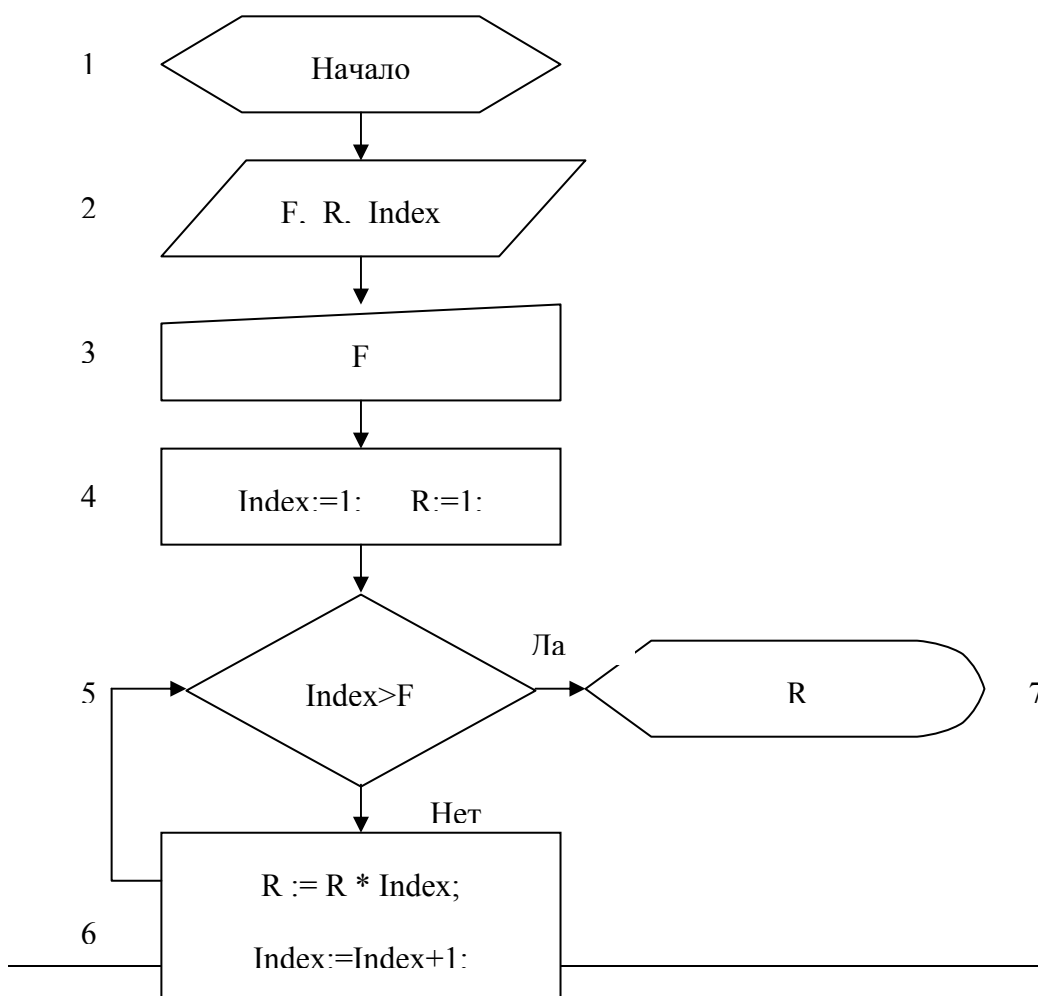
2.3 Машинная логика и циклы.

В любом языке программирования есть куча операций сравнения, которые позволяют реализовать машинную логику. Что понимается под логическими операциями? Это операции сравнения на «равенство», «больше» или «меньше».

Математическая операция	Описание
=	Равно
>	Больше
<	Меньше

Давай рассмотрим простейшую логику компьютера на примере факториала. Факториал – это произведение от 1 до какого-то числа. Например, 5 факториал равен $1*2*3*4*5=120$. Факториал обозначается как знак восклицания «!».

Давай напомним алгоритм в виде блок-схемы:



Представим, что мы не знаем число, факториал которого мы должны вычислить. Это число будет вводить пользователь. Поэтому в общем случае формула будет выглядеть, как число F факториал ($F!$). Так что нам надо перемножить все числа от $1*2*3*4*\dots*F$. Всё это можно сделать последовательно. Умножить $1*1$. Затем проверить, не превысили ли мы число F, если нет, то результат прошлого вычисления $*$ на 2. Снова проверить. Если не превысили F, то снова умножить результат прошлого вычисления на 3. И так далее.

Теперь я постараюсь объяснить каждый блок, чтобы ты понял, как эта блок-схема работает. Это очень важно. Работа блок-схемы очень похожа на то, как мыслит компьютер. Именно так ты должен будешь размышлять, когда начнёшь писать первые программы.

Итак, давай рассмотрим каждый блок в отдельности:

1. Это опять начало.
2. Я объявляю три переменных: F, R и Index. F – здесь я буду хранить число, факториал которого надо вычислить. R – это для результата. Index - здесь я буду держать счётчик вычислений.
3. В этом блоке происходит ввод числа, факториал которого надо вычислить. Если пользователь ввёл 5, то мне нужно вычислить $5!$.
4. Задаю начальные значения переменным. На этом этапе я устанавливаю счётчик и результат равным 1. Почему именно 1? Да потому что мне нужно перемножить все числа начиная от 1 до числа факториала.
5. В этом блоке я проверяю – счётчик (index) больше числа, которое ввёл пользователь? Если «нет», то надо перейти на блок 6. Допустим, пользователь ввёл число 5. Наш счётчик пока равен 1. 1 меньше 5, значит, мы должны перейти на блок 6.
6. Здесь я вычисляю результат $R:=R*Index$. На этом этапе у меня $Index=1$ и $R=1$. Значит, $R:=1*1$. Результат будет 1, значит, в R опять будет единица. Далее, я увеличиваю index на 1 ($Index:=Index+1$), после чего Index становится равным 2 ($Index:=1+1$). И снова возвращаюсь на блок 5. В нём опять происходит проверка $Index>F$. Index сейчас равен 2, а это меньше пяти. Значит, снова идём на блок 6. Здесь опять расчёт $R:=R*Index$ ($R:=1*2$), после чего R становится равным 2. Опять увеличиваем счётчик ($Index:=Index+1$), и Index становится равным 3. Снова на блок 5.

... И так далее.

Я не стал дальше расписывать. Попробуй сам пройти по этой блок-схеме, рассуждая так же, как и я. Очень важно понять, как это работает.

В этой блок-схеме мы задействовали очень интересную и очень удобную вещь, как циклы. Цикл – повторяющееся выполнение какого-то блока. В данном случае мы несколько раз выполняем шестой блок. Если бы мы знали заранее число факториала, то мы могли бы упростить нашу блок-схему, написав формулу типа $R:=1*2*3*4*5$. Но мы не знаем числа, которое введёт пользователь. Поэтому нам приходится делать цикл на каждом этапе которого, вычисляем промежуточный результат.

Попробуй сам написать блок-схему арифметической прогрессии. Её формула достаточно проста и ты без проблем сможешь модифицировать блок-схему. Разница только в том, что арифметической прогрессии рассчитывается сумма чисел от 1 до определённого числа F.

Нарисуй эту блок-схему на бумаге и попробуй мысленно пройти по ней, выполняя каждый блок, как бы это делала машина. Даже если ты думаешь, что блок-схемы слишком

просты, то попробуй написать что-нибудь более сложное. Ты просто обязан научиться мыслить как компьютер. Только так ты сможешь объяснить ему то, что тебе нужно, и он тебя сможет понять.

2.4 Программирование машинной логики.

Подошло время превратить нашу логику, описанную в блок-схеме в настоящую программу. Пока эта программа будет существовать только на бумаге, но со временем мы её сможем превратить в настоящий исполняемый модуль.

Сначала напомним нашу программу на русском языке:

Начало программы.

Переменные:

F, R, Index – это целые числа;

Начало кода

F:=5;

R:=1;

Index:=1;

От 1 до 5 выполнять

Начало цикла

*R:=R*INDEX;*

INDEX:=INDEX+1;

Конец цикла

Вывести на экран переменную R.

Конец кода

Если не обращать внимания на то, что всё написано русским языком, то можно считать, что мы уже написали первую программу. В принципе, программа на любой языке программирования выглядит приблизительно так, как мы это описали. В данном случае, наша программа состоит из следующих блоков:

1. Начало программы
2. Описание переменных.
3. Начало кода (заметь, что описание переменных, это не код программы).
4. Заполнение переменных начальными значениями.
5. Запуск цикла от 1 до 5.
6. Выполнение в цикле расчёта.
7. Вывод результата.

В принципе, ничего нового здесь нет. Я просто описал блок-схему словами, похожими на программный язык. В дальнейшем, когда ты сам начнёшь писать свой собственный код, ты должен будешь действовать так же. Сначала построить алгоритм программы в виде блок-схемы или хотя бы мысленно представить алгоритм работы будущей программы и только потом перенести всё это в компьютер.

Не пугайся, не всё так сложно. Я строил такие блок-схемы только на начальном этапе. Сейчас я уже пишу программы без использования всякой дополнительной логики. Ты тоже сможешь так же, если поймёшь, как мыслит машина и сможешь думать её логикой.

Это то же самое, что разговаривать с англичанином. Мало знать английских слов, нужно ещё и мыслить как он. Я сразу вспоминаю случай, который случился со мной в Испании:

Как-то раз, ко мне подошла девушка и, узнав, что я из России стала просить у меня деревянную ложку в качестве сувенира. У меня не было ложек, я просто не брал их с собой. Поэтому мне пришлось сказать ей No, т. е. «Нет». Я это воспринимал, как у меня НЕТ ложки. Она меня спрашивала, почему НЕТ, воспринимая мой ответ, как будто я отказываюсь дать ей эту чёртову ложку. На английском «NO» - это отрицание, а по нашему «Нет» - это не только отрицание, но и признак отсутствия (например, у меня нет ложки). Если бы я тогда мыслил, как англичанин, то я бы смог ответить: «I don't have». Но этому я научился намного позже, тогда я ещё очень слабо владел английским языком.

Точно так же и машинный язык. В большинстве случаев он схож с человеческим, потому что его создавали люди. Но иногда разница чувствительна, потому что машина не всегда может мыслить, как человек. Поэтому ты должен научиться объяснять свои мысли понятным для машины языком.

Я думаю, что мы готовы перейти к изучению программирования. Возможно, в будущем я улучшу эту главу, если увижу, что не все мои читатели смогли понять, что я пытаюсь сказать. Да и в любом случае, сколько не старайся, основная часть знаний прейдёт только на практике. Только практика сможет закрепить описанные мною здесь знания. И всё же, чем больше ты усвоил из сказанного ранее, тем легче будет дальше.