Сценарии shell и make-файлы

Общие сведения о shell

- Командный язык shell (в переводе - раковина) фактически есть язык программирования высокого уровня. На этом языке пользователь осуществляет управление компьютером. Обычно, после входа в систему пользователь начинает взаимодействовать с командной оболочкой. Признаком того, что оболочка (shell) готова к приему команд служит выдаваемый ею на экран промптер. В простейшем случае это символ доллара ("$").
- Shell не является необходимым и единственным командным языком. Например, немалой популярностью пользуется язык cshell, есть также kshell, bashell и другие. Может одновременно на одном экземпляре операционной системы работать с разными командными языками.

Общие сведения о shell

- shell - это одна из многих команд UNIX. То есть в набор команд оболочки (интерпретатора) "shell" входит команда "sh" - вызов интерпретатора "shell". Первый "shell" вызывается автоматически при вашем входе в систему и выдает на экран промтер. После этого можно вызывать на выполнение любые команды, в том числе и снова сам "shell", который создаст новую оболочку внутри прежней.
- Так например, если подготовить в редакторе файл "f1":
- `echo Hello!`
- то это будет обычный текстовый файл, содержащий команду "echo", которая при выполнении выдает все написанное правее ее на экран. Можно сделать файл "f1" выполняемым с помощью команды "chmod 755 f1".

Общие сведения о shell

- Но его можно ВЫПОЛНИТЬ, вызвав явно команду (!) "sh" ("shell"):
- **`sh f1`** или
- **`sh < f1`**
- Файл можно выполнить и в текущем экземпляре "shell". Для этого существует специфическая команда "./"
- **`./f1`**
- Не стоит начинать командные файлы с символа "#", хотя естественно начинать его с комментария. Такой командный файл в оболочке C-Shell ("csh") будет интерпретирован как выполняемый в "csh", в результате будет активизирован. интерпретатор "csh".
- Начинать командный sh-файл следует с пустой строки или пустого оператора ":".

Структура команд

- Стержневым элементом языка shell является команда.
- Команды в shell обычно имеют следующий формат:
- **<имя команды> <флаги> <аргумент(ы)>**
- Например:
- **ls -ls /usr/bin**
- Здесь:
- ls - имя команды выдачи содержимого директория,
- -ls - флаги ("-" - признак флагов, l - длинный формат, s - объем файлов в блоках).
- /usr/bin - директорий, для которого выполняется команда.

Группировка команд

- Средства группировки:
- **;** и **<перевод строки>:** определяют последовательное выполнение команд;
- **&:** асинхронное (фоновое) выполнение предшествующей команды;
- **&&** выполнение последующей команды при условии нормального завершения предыдущей, иначе игнорировать;
- **||:** выполнение последующей команды при ненормальном завершении предыдущей, иначе игнорировать.

Группировка команд

- При выполнении команды в асинхронном режиме (после команды стоит один амперсанд) на экран выводится номер процесса, соответствующий выполняемой команде, и система, запустив этот фоновый процесс, вновь выходит на диалог с пользователем.
- Иногда необходимо, чтобы все фоновые процессы завершились, прежде чем будет выполняться какой-то расчет. Для этого служит специальная команда "wait [PID]". Эта команда ждет завершения указанного идентификатором (числом) фонового процесса. Если команда без параметра, то она ждет завершения всех фоновых процессов, дочерних для данного "sh".
- Для группировки команд также могут использоваться фигурные "{}" и круглые "()" скобки.

Перенаправление команд

- Стандартный ввод - "stdin" в ОС UNIX осуществляется с клавиатуры терминала, а стандартный вывод - "stdout" направлен на экран терминала. Стандартный файл диагностических сообщений - "stderr". Команда, которая может работать со стандартным вводом и выводом, называется ФИЛЬТРОМ.
- Пользователь имеет удобные средства перенаправления ввода и вывода на другие файлы (устройства). Символы ">" и ">>" обозначают перенаправление вывода. Пример:
- `ls >f1`
- команда "ls" сформирует список файлов текущего каталога и поместит его в файл "f1" (вместо выдачи на экран). Если файл "f1" до этого существовал, то он будет затерт новым.

Перенаправление команд

- **`pwd >>f1`**
- команда pwd сформирует полное имя текущего каталога и поместит его в конец файла "f1", т.е. ">>" добавляет в файл, если он непустой.
- Символы "<" и "<<" обозначают перенаправление ввода.
- **`wc -l <f1`**
- подсчитает и выдаст на экран число строк в файле f1.
- **`vi f2 <<!`**
- создаст с использованием редактора файл "f2", непосредственно с терминала. Окончание ввода определяется по символу, стоящему правее "<<" (т. е. "!"). То есть ввод будет закончен, когда первым в очередной строке будет "!".

Перенаправление команд

- Можно сочетать перенаправления. Так
- **`wc -l <f3 >f4`** и **`wc -l >f4 <f3`** выполняются одинаково: подсчитывается число строк файла "f3" и результат помещается в файл "f4".
- Средство, объединяющее стандартный выход одной команды со стандартным входом другой, называется КОНВЕЙЕРОМ и обозначается вертикальной чертой "|".
- **`ls | wc -l`**
- список файлов текущего каталога будет направлен на вход команды "wc", которая на экран выведет число строк каталога.
- Конвейером можно объединять и более двух команд, когда все они, возможно кроме первой и последней - фильтры:

Перенаправление команд

- `cat f1| grep -h result | sort | cat -b > f2`
- Данный конвейер из файла "f1" ("cat") выберет все строки, содержащие слово "result" ("grep"), отсортирует ("sort") полученные строки, а затем пронумерует ("cat -b") и выведет результат в файл "f2".
- Поскольку устройства в ОС UNIX представлены специальными файлами, их можно использовать при перенаправлениях. Специальные файлы находятся в каталоге "/dev". Например, "lp" - печать; "console" - консоль; "ttyi" - i-ый терминал; "null" - фиктивный (пустой) файл (устройство).
- Тогда, например,
- `ls > /dev/lp`
- выведет содержимое текущего каталога на печать, а

Перенаправление команд

- `f1 < /dev/null`
- обнулит файл "f1".
- Стандартные файлы имеют номера: 0 - stdin, 1 - stdout и 2 - stderr. Если не желательно иметь на экране сообщение об ошибке, можно перенаправить его с экрана в указанный файл (или вообще "выбросить", перенаправив в файл "пустого устройства" - /dev/null).
- Можно указать не только какой из стандартных файлов перенаправлять, но и в какой стандартный файл осуществить перенаправление.
- cat f1 f2 2>>ff 1>&2
- Здесь "stderr" (при отсутствии какого-либо из файлов f1, f2) перенаправляется (в режиме добавления) в файл "ff", а стандартный выход перенаправляется на "stderr", которым к этому моменту является файл "ff".

Перенаправление команд

- Конструкция "1>&2" - означает, что кроме номера стандартного файла, в который перенаправить, необходимо впереди ставить "&"; вся конструкция пишется без пробелов.
- Есть и еще варианты перенаправлений:
- **<-** закрывает стандартный ввод.
- **>-** закрывает стандартный вывод.

Командные файлы

- Для того, чтобы текстовый файл можно было использовать как команду, существует несколько возможностей.
- Пусть с помощью редактора создан файл с именем "cmd", содержащий одну строку следующего вида:
- `date; pwd; ls`
- Можно вызвать shell как команду, обозначаемую "sh", и передать ей файл "cmd", как аргумент или как перенаправленный вход:
- `$sh cmd`
- Или
- `$sh <cmd`

Командные файлы

- В результате выполнения любой из этих команд будет выдана дата, затем имя текущего каталога, а потом содержимое каталога.
- Более интересный и удобный вариант работы с командным файлом - это превратить его в выполняемый, т.е. просто сделать его командой, что достигается изменением кода защиты. Для этого надо разрешить выполнение этого файла.
- Например,
- `chmod 711 cmd`
- сделает код защиты "rwx__x__x". Тогда простой вызов
- cmd приведет к выполнению тех же трех команд.

Командные файлы

- Таким образом, выполняемыми файлами могут быть не только файлы, полученные в результате компиляции и сборки, но и файлы, написанные на языке shell. Их выполнение происходит в режиме интерпретации с помощью shell-интерпретатора.
- На языке shell можно писать командные файлы и с помощью команды "chmod" делать их выполняемыми. После этого они ни чем не отличаются от прочих команд ОС UNIX.

shell-переменные

- Имя shell-переменной - это начинающаяся с буквы последовательность букв, цифр и подчеркиваний.
- Значение shell-переменной - строка символов.
- То, что в shell всего два типа данных: строка символов и текстовый файл, с одной стороны, позволяет легко вовлекать в программирование конечных пользователей, никогда ранее программированием не занимавшихся, а с другой стороны, вызывает некий внутренний протест у многих программистов, привыкших к существенно большему разнообразию и большей гибкости языковых средств.
- Для присваивания значений переменным может использоваться оператор присваивания "=".

shell-переменные

- **var_1=13 - "13" #- это не число, а строка из двух цифр.**
- **var_2="ОС UNIX" #- здесь двойные кавычки (" ") необходимы, так как в строке есть пробел.**
- Обратим внимание на то, что, как переменная, так и ее значение должны быть записаны без пробелов относительно символа "=".
- Можно присвоить значение переменной и с помощью команды "read", которая обеспечивает прием значения переменной с клавиатуры в диалоговом режиме. Обычно команде "read" предшествует команда "echo", которая позволяет предварительно выдать сообщение на экран. Например:

shell-переменные

- `echo -n "Введите трехзначное число: "`
- `read x`
- При выполнении этого фрагмента командного файла, после вывода на экран сообщения
- **`Введите трехзначное число:`**
- интерпретатор остановится и будет ждать ввода значения с клавиатуры. Если вы ввели, скажем, "753" то это и станет значением переменной "x".

Параметры

- В командный файл могут быть переданы параметры. В shell используются позиционные параметры (т.е. существенна очередность их следования). В командном файле соответствующие параметрам переменные (аналогично shell-переменным) начинаются с символа "$", а далее следует одна из цифр от 0 до 9:
- Пусть сценарий "examp-1" вызывается с параметрами "cock" и "tail". Эти параметры попадают в новую среду под стандартными именами "1" и "2". В (стандартной) переменной с именем "0" будет храниться имя вызванного расчета.
- При обращении к параметрам перед цифрой ставится символ доллара "$" (как и при обращении к переменным):

Параметры

- **$0** соответствует имени данного командного файла;
- **$1** первый по порядку параметр;
- **$2** второй параметр и т.д.
- Поскольку число переменных, в которые могут передаваться параметры, ограничено одной цифрой, ("0" имеет особый смысл), то для передачи большего числа параметров используется команда "shift".
- Своеобразный подход к параметрам дает команда "set".
- Например, фрагмент сценария
- `set a b c`
- `echo первый=$1 второй=$2 третий=$3`
- выдаст на экран
- `первый=a второй=b третий=c`

Параметры

- то есть команда "set" устанавливает значения параметров. Например, команда "date" выдает на экран текущую дату, скажем, "Mon May 01 12:15:10 2012", состоящую из пяти слов, тогда
- `set `date`echo $1 $3 $5`
- выдаст на экран
- `Mon 01 2012`
- Команда "set" позволяет также осуществлять контроль выполнения программы, например:
- set –v: на терминал выводятся строки, читаемые shell. Set +v: отменяет предыдущий режим.
- Set –x: на терминал выводятся команды перед выполнением. Set +x: отменяет предыдущий режим.
- Команда "set" без параметров выводит на терминал состояние программной среды.

Программные структуры

- Как во всяком языке программирования в тексте на языке shell могут быть комментарии. Для этого используется символ "#". Все, что находится в строке (в командном файле) правее этого символа, воспринимается интерпретатором как комментарий. Например,
- # Это комментарий.
- ## И это.
- Как во всяком процедурном языке программирования в языке shell есть операторы. Ряд операторов позволяет управлять последовательностью выполнения команд. В таких операторах часто необходима проверка условия, которая и определяет направление продолжения вычислений.

Команда test ("[ ]")

- Команда test проверяет выполнение некоторого условия. С использованием этой (встроенной) команды формируются операторы выбора и цикла языка shell.
- Два возможных формата команды:
- **`test условие`** или
- **`[ условие ]`**
- shell будет распознавать эту команду по открывающей скобке "[", как слову(!), соответствующему команде "test". Между скобками и содержащимся в них условием обязательно должны быть пробелы.
- Пробелы должны быть и между значениями и символом сравнения или операции (как, кстати, и в команде "expr"). Не путать с противоположным требованием для присваивания значений переменным.

Команда test ("[ ]")

- В shell используются условия различных "типов".
- Условия проверки файлов:
- **-f file** файл "file" является обычным файлом;
- **-d file** файл "file" - каталог;
- **-c file** файл "file" - специальный файл;
- **-r file** имеется разрешение на чтение файла "file";
- **-w file** имеется разрешение на запись в файл "file";
- **-s file** файл "file" не пустой.
- Пример. В первом случае получим подтверждение (код завершения "0"), а во втором - опровержение (код завершения "1"). "specific" - имя существующего файла.
- `[ -f specific ] ; echo $?`
- `[ -d specific ] ; echo $?`

Команда test ("[ ]")

- Условия проверки строк:
- str1 = str2 строки "str1" и "str2" совпадают;
- str1 != str2 строки "str1" и "str2" не совпадают;
- -n str1 строка "str1" существует (непустая);
- -z str1 строка "str1" не существует (пустая).
- Условия сравнения целых чисел:
- x -eq y "x" равно "y",
- x -ne y "x" неравно "y",
- x -gt y "x" больше "y",
- x -ge y "x" больше или равно "y",
- x -lt y "x" меньше "y",
- x -le y "x" меньше или равно "y".

Команда test ("[ ]")

- То есть в данном случае команда "test" воспринимает строки символов как целые числа. Поэтому во всех остальных случаях "нулевому" значению соответствует пустая строка. В данном же случае, если надо обнулить переменную, скажем, "x", то это достигается присваиванием "x=0".
- Сложные условия:
- Реализуются с помощью типовых логических операций:
- **!** (not) инвертирует значение кода завершения.
- **-o** (or) соответствует логическому "ИЛИ".
- **-a** (and) соответствует логическому "И".

Условный оператор "if"

- В общем случае оператор "if" имеет структуру
- **if условие**
  - **then список**
  - **[elif условие**
  - **then список]**
  - **[else список]**
- **fi**
- Здесь "elif" (сокращенный вариант от "else if ") может быть использован наряду с полным, т.е. допускается вложение произвольного числа операторов "if" (как и других операторов).
- Конструкции
- **[elif условие then список]** и
- **[else список]**

Условный оператор "if"

- не являются обязательными (в данном случае для указания на необязательность конструкций использованы квадратные скобки - не путать с квадратными скобками команды "test"!).
- Самая усеченная структура этого оператора
- **`if условие`**
- **`then список`**
- **`fi`**
- Если выполнено условие (как правило это ком получен код завершения "0"), то выполняется "список", иначе он пропускается.
- Обратите внимание, что структура обязательно завершается служебным словом "fi". Число "fi", естественно, всегда должно соответствовать числу "if".

Условный оператор "if"

- Пример.
- Пусть написан сценарий "if-1"
- `if [ $1 -gt $2 ]`
  - `then pwd`
  - `else echo $0 : Hello!`
- `fi`
- Тогда вызов сценария
- `if-1 12 11`
- даст
- `/home/gun/SHELL`
- `if-1 12 13`
- даст
- `if-1 : Hello!`

Оператор вызова ("case")

- Оператор выбора "case" имеет структуру:
- `case строка in`
- `шаблон) список команд;;`
- `шаблон) список команд;;`
- `...esac`
- Здесь "case" "in" и "esac" - служебные слова. "Строка" (это может быть и один символ) сравнивается с "шаблоном". Затем выполняется "список команд" выбранной строки. Непривычным будет служебное слово "esac", но оно необходимо для завершения структуры.

Оператор вызова ("case")

- Пример.
- `#### case-1: Структура "case".`
- `echo -n " А какую оценку получил на экзамене?: «`
- `read zcase $z in`
- `5) echo Молодец !  ;;`
- `4) echo Все равно молодец !  ;;`
- `3) echo Все равно !  ;;`
- `2) echo Все !  ;;`
- `*) echo  !  ;;`
- `esac`

Оператор вызова ("case")

- Непривычно выглядят в конце строк выбора ";;", но написать здесь ";" было бы ошибкой. Для каждой альтернативы может быть выполнено несколько команд. Если эти команды будут записаны в одну строку, то символ ";" будет использоваться как разделитель команд.
- Обычно последняя строка выбора имеет шаблон "*", что в структуре "case" означает "любое значение". Эта строка выбирается, если не произошло совпадение значения переменной (здесь $z) ни с одним из ранее записанных шаблонов, ограниченных скобкой ")". Значения просматриваются в порядке записи.

Оператор цикла с перечислением ("for")

- Оператор цикла "for" имеет структуру:
- **`for имя [in список значений]`**
  - **`do  список команд`**
- **`done`**
- где "for" - служебное слово определяющее тип цикла, "do" и "done" - служебные слова, выделяющие тело цикла. Фрагмент "in список значений" может отсутствовать.
- Пусть команда "lsort" представлена командным файлом
- **`for i in f1 f2 f3`**
  - **`do`**
  - **`proc-sort $i`**
- **`done`**

Оператор цикла с перечислением ("for")

- В этом примере имя "i" играет роль параметра цикла. Это имя можно рассматривать как shell-переменную, которой последовательно присваиваются перечисленные значения (i=f1, i=f2, i=f3), и выполняется в цикле команда "procsort".
- Часто используется форма "for i in *", означающая "для всех файлов текущего каталога".

Оператор цикла с истинным условием ("while")

- Структура "while" предпочтительнее тогда, когда неизвестен заранее точный список значений параметров или этот список должен быть получен в результате вычислений в цикле.
- Оператор цикла "while" имеет структуру:
- **`while условие`**
  - **`do список команд`**
- **`done`**
- где "while" - служебное слово определяющее тип цикла с истинным условием. Список команд в теле цикла (между "do" и "done") повторяется до тех пор, пока сохраняется истинность условия или цикл не будет прерван изнутри специальными командами ("break", "continue" или "exit"). При первом входе в цикл условие должно выполняться.

Оператор цикла с истинным условием ("while")

- `#### print-50: Структура "while"`
- `n=0`
- `while [ $n -lt 50 ]`
- `do n=`expr $n + 1``
- `cat file-22 > /dev/lp`
- `done`
- Обратим внимание на то, что переменной "n" вначале присваивается значение 0, а не пустая строка, так как команда "expr" работает с shell-переменными как с целыми числами, а не как со строками.
- `n=`expr $n + 1``
- т.е. при каждом выполнении значение "n" увеличивается на 1.

Оператор цикла с истинным условием ("while")

- Расчет "pr-br" приведен для иллюстрации бесконечного цикла и использования команды "break", которая обеспечивает прекращение цикла.
- `#### pr-br: Структура "while" с "break"`
- `n=0`
- `while true`
- `do`
  - `if [ $n -lt 50 ]`
  - `then n=`expr $n + 1``
  - `else break`
  - `fi`
- `cat file-22 > /dev/lp`
- `done`

Оператор цикла с истинным условием ("while")

- Команда "break [n]" позволяет выходить из цикла. Если "n" отсутствует, то это эквивалентно "break 1". "n" указывает число вложенных циклов, из которых надо выйти: "break 3" - выход из трех вложенных циклов.
- Команда "continue [n]" прекращает выполнение текущего цикла и возвращает на НАЧАЛО цикла. Она также может быть с параметром. Например, "continue 2" означает выход на начало второго (если считать из глубины) вложенного цикла.
- Команда "exit [n]" позволяет выйти вообще из процедуры с кодом возврата "0" или "n" (если параметр "n" указан). Эта команда может использоваться не только в циклах. Она может быть полезна при отладке, чтобы прекратить выполнение (текущего) сценария в заданной точке.

Оператор цикла с ложным условием ("until")

- Оператор цикла "until" имеет структуру:
- **`until условие`**
- **`do  список команд`**
- **`done`**
- где "until" - служебное слово, определяющее тип цикла с ложным условием. Список команд в теле цикла (между "do" и "done") повторяется до тех пор, пока сохраняется ложность условия или цикл не будет прерван изнутри специальными командами ("break", "continue" или "exit"). При первом входе в цикл условие не должно выполняться.
- Отличие от оператора "while" в том, что условие цикла проверяется на ложность ПОСЛЕ каждого (в том числе и первого!) выполнения команд тела цикла.

Оператор цикла с ложным условием ("until")

- Пример.
- `until false`
- `do  read x`
  - `if    [ $x = 5 ]`
  - `then echo enough ;`
  - `break`
  - `else echo some more`
  - `fi`
- `done`
- Здесь программа с бесконечным циклом ждет ввода слов (повторяя на экране фразу "some more"), пока не будет введено "5". После этого выдается "enough" и команда "break" прекращает выполнение цикла.

Пустой оператор

- Пустой оператор имеет формат
- :
- Ничего не делает. Возвращает значение "0". Применяется, например, в конструкции "while :" для организации бесконечного цикла или ставить в начале командного файла, чтобы гарантировать, что файл не будет принят за выполняемый файл для "csh".

Функции в shell

- Функция позволяет подготовить список команд shell для последующего выполнения. Описание функции:
- **имя**()
- {
- **список команд**
- }
- после чего обращение к функции происходит по имени. При выполнении функции не создается нового процесса. Она выполняется в среде соответствующего процесса. Аргументы функции становятся ее позиционными параметрами; имя функции - ее нулевой параметр. Прервать выполнение функции можно оператором "return [n]", где (необязательное) "n" - код возврата.

Утилита make

- Утилита make автоматически определяет какие части большой программы должны быть перекомпилированы, и выполняет необходимые для этого действия.
- В примерах будут фигурировать программы на языке Си, однако, можно использовать make с любым языком программирования для которого имеется компилятор, работающий из командной строки. На самом деле, область применения make не ограничивается только сборкой программ. Вы можете использовать ее для решения любых задач, где одни файлы должны автоматически обновляться при изменении других файлов.

Синтаксис make-файлов

- Перед тем, как использовать make, вы должны создать так называемый **make-файл (makefile)**, который будет описывать зависимости между файлами вашей программы, и содержать команды для обновления этих файлов. Как правило, исполняемый файл программы зависит от объектных файлов, которые, в свою очередь, получаются в результате компиляции соответствующих файлов с исходными текстами.
- После того, как нужный make-файл создан, простой команды :
- `$make`
- будет достаточно для выполнения всех необходимых перекомпиляций если какие-либо из исходных файлов программы были изменены.

Синтаксис make-файлов

- Используя информацию из make-файла, и, зная время последней модификации файлов, утилита make решает, каких из файлов должны быть обновлены. Для каждого из этих файлов будут выполнены указанные в make-файле команды.
- При вызове make, в командной строке могут быть заданы параметры, указывающие, какие файлы следует перекомпилировать и каким образом это делать.
- Простой make-файл состоит из "правил" (rules) следующего вида:
- *цель* ... : *пререквизит* ... *команда*
- Обычно, **цель (target)** представляет собой имя файла, который генерируется в процессе работы утилиты make.

Синтаксис make-файлов

- Примером могут служить объектные и исполняемый файлы собираемой программы. Цель также может быть именем некоторого действия, которое нужно выполнить (например, `clean' - очистить).
- **Пререквизит (prerequisite)** - это файл, который используется как исходные данные для порождения цели. Очень часто цель зависит сразу от нескольких файлов.
- **Команда** - это действие, выполняемое утилитой make. В правиле может содержаться несколько команд - каждая на свое собственной строке. **Важное замечание:** строки, содержащие команды обязательно должны начинаться с символа табуляции! Это - "грабли", на которые наступают многие начинающие пользователи.

Синтаксис make-файлов

- Обычно, команды находятся в правилах с пререквизитами и служат для создания файла-цели, если какой-нибудь из пререквизитов был модифицирован. Однако, правило, имеющее команды, не обязательно должно иметь пререквизиты. Например, правило с целью `clean' ("очистка"), содержащее команды удаления, может не иметь пререквизитов.
- **Правило (rule)** описывает, когда и каким образом следует обновлять файлы, указанные в нем в качестве цели. Для создания или обновления цели, make исполняет указанные в правиле команды, используя пререквизиты в качестве исходных данных. Правило также может описывать, каким образом должно выполняться некоторое действие.

Синтаксис make-файлов

- Помимо правил, make-файл может содержать и другие конструкции, однако, простой make-файл может состоять и из одних лишь правил.
- Вот пример простого make-файла, в котором описывается, что исполняемый файл edit зависит от объектного файла, который, в свою очередь, зависит от соответствующих исходного файла и заголовочного файлов.

```
edit : main.o
        cc -o edit main.o
main.o : main.c defs.h
        cc -c main.c
clean :
        rm edit main.o
```

Синтаксис make-файлов

- Для повышения удобочитаемости, можно разбить длинные строки на две части с помощью символа обратной косой черты, за которым следует перевод строки. Для того, чтобы с помощью этого make-файла создать исполняемый файл `edit', наберите:
- $make
- Для того, чтобы удалить исполняемый и объектные файлы из директории проекта, наберите:
- make clean
- В примере целями являются объектный файл `main.o', а также исполняемый файл `edit'. К пререквизитам относятся файлы`main.c' и `defs.h'. Каждый объектный файл является одновременно и целью и пререквизитом. Пример команд `cc -c main.c'.

Синтаксис make-файлов

- Цель `clean' является не файлом, а именем действия. Поскольку, при обычной сборке программы это действие не требуется, цель `clean' не является пререквизитом какого-либо из правил. Следовательно, make не будет "трогать" это правило, пока вы специально об этом не попросите. Заметьте, что это правило не только не является пререквизитом, но и само не содержит каких-либо пререквизитов. Таким образом, единственное предназначение данного правила - выполнение указанных в нем команд. Цели, которые являются не файлами, а именами действий называются *абстрактными целями (phony targets)*.