<http://linux.yaroslavl.ru/docs/prog/gnu_make_3-79_russian_manual.html#SEC9> - GNU Make. Программа управления компиляцией.

**Практика Makefile**:

Make-файл (makefile) - описывает зависимости между файлами вашей программы, и содержит 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.

Make-файл состоит из "правил" (rules) следующего вида:

*цель* ... : *пререквизит* ...

*команда*

...

...

* **Цель (target)** - имя файла, который генерируется в процессе работы утилиты
* **Пререквизит (prerequisite)** - файл, который используется как исходные данные для порождения цели. Очень часто цель зависит сразу от нескольких файлов.
* **Команда** - действие, выполняемое утилитой make. В правиле может содержаться несколько команд - каждая на своей собственной строке. Важное замечание: строки, содержащие команды обязательно должны начинаться с символа табуляции! Это - "грабли", на которые наступают многие начинающие пользователи.
* **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.
* Мы разбили длинные строки на две части с помощью символа обратной косой черты, за которым следует перевод строки.

clean :

rm edit main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

* 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)**.
* По умолчанию, make начинает свою работу с первой встреченной цели (кроме целей, чье имя начинается с символа `.'). Эта цель будет **являться главной целью по умолчанию (default goal). Главная цель (goal)** - это цель, которую стремится достичь make в качестве результата своей работы.

***Упрощение make-файла с помощью переменных***

* **Переменные (variables)** позволяют, один раз определив текстовую строку, затем использовать ее многократно в нужных местах.
* Обычной практикой при построении make-файлов является использование переменной с именем objects, OBJECTS, objs, OBJS, obj, или OBJ, которая содержит список всех объектных файлов программы.

objects = main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

* Далее, мы можем использовать значение этой переменной с помощью записи `$**(objects)**'

***[Неявные правила упрощают make-файл](http://linux.yaroslavl.ru/docs/prog/gnu_make_3-79_russian_manual.html" \l "TOC9)***

* На самом деле, нет необходимости явного указания команд компиляции отдельно для каждого из исходных файлов. Утилита make сама может "догадаться" об использовании нужных команд, поскольку у нее имеется **неявное правило (implicit rule)** для обновления файлов с расширением `.o' из файлов с расширеним `.c', с помощью команды `cc -c'.
* Например, она бы использовала команду `cc -c main.c -o main.o' для преобразования файла `main.c' в файл `main.o'. Таким образом, можно убрать явное указание команд компиляции из правил, описывающих построение объектных файлов.
* Когда файл с расширением `.c' автоматически используется подобным образом, он также автоматически добавляется в список пререквизитов "своего" объектного файла. Таким образом, мы можем убрать файлы с расширением `.c' из списков пререквизитов объектных файлов.

objects = main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

edit : $(objects)

cc -o edit $(objects)

main.o : defs.h

kbd.o : defs.h command.h

command.o : defs.h command.h

display.o : defs.h buffer.h

.PHONY : clean

clean :

-rm edit $(objects)

***Другой стиль написания make-файлов***

* Альтернативный стиль написания make-файлов - записи группируются по их пререквизитам, а не по их целям. Вот как может выглядеть подобный make-файл:

objects = main.o kbd.o command.o display.o \

insert.o search.o files.o utils.o

edit : $(objects)

cc -o edit $(objects)

$(objects) : defs.h

kbd.o command.o files.o : command.h

display.o insert.o search.o files.o : buffer.h

* Здесь, заголовочный файл `defs.h' объявляется пререквизитом для всех объектных файлов программы. Файлы `command.h' и `buffer.h' являются пререквизитами для перечисленных объектных файлов.

***Правило для очистки каталога***

.PHONY : clean

clean :

-rm edit $(objects)

* .PHONY - предотвратит возможную путаницу если, в каталоге будет находится файл с именем `clean', а также позволит make продолжить работу, даже если команда rm завершится с ошибкой.
* Подобное правило(clean) не следует помещать в начало make-файла, поскольку мы не хотим, чтобы оно запускалось "по умолчанию"!
* Поскольку clean не является пререквизитом цели edit, это правило не будет выполняться, если вызывать `make' без аргументов. Для запуска данного правила, нужно будет набрать `make clean'.

***Из чего состоят make-файлы (5 видов конструкций):***

* **Явное правило (explicit rule)** описывает, когда и каким образом следует обновлять файлы, называемые целями правила. В этом правиле перечисляются файлы, от которых зависит цель правила (так называемые пререквизиты), а также могут быть заданы команды, которые следует использовать для создания или обновления цели.
* **Неявное правило (implicit rule)** описывает, когда и каким образом нужно обновлять некоторую группу файлов, имена которых подходят под определенный шаблон. Пример, OBJ = $(FILES\_C:%.c=%.o). % - знак шаблона.

Такое правило описывает, как цель может зависеть от файла со "сходным" именем и задает команды для обновления целей.

* **Определение переменной (variable definition)** - это строка make-файла, в которой переменной присваивается определенное текстовое значение. Далее, это значение может быть "подставлено" в нужном месте текста. В нашем примере make-файла, переменная objects определялась как список объектных файлов программы.
* **Директива** указывает программе make на необходимость произведения некоторого специального действия во время чтения make-файла. Возможны, в частности, следующие действия:
  + Чтение другого make-файла.
  + Решение (на основе значения переменных) об использовании или игнорировании части make-файла.
  + Определение многострочной переменной, состоящей из нескольких строк.
* **Символ `#' - комментарий**. Комментарии не могут находится внутри директивы define и, возможно, внутри команд (поскольку, здесь уже интерпретатор командной строки будет решать - что именно является комментарием).

***Имена make-файлов***

* По умолчанию, make-файл для обработки, пробуют найти файлы с именами (в указанном порядке): `GNUmakefile', `makefile' и `Makefile'.
* Рекомендуем имя `Makefile', потому что при выводе содержимого каталога, файл с таким именем будет находится в начале списка, наряду с такими важными файлами как `README'.
* `GNUmakefile' - не может быть рекомендовано для большинсства make-файлов, т.к. специфичен для GNU make и не будет обрабатываться другими версиями make.
* В том случае, если make не может найти файлов с перечисленными выше именами, то она пробует продолжить работу без использования make-файла.
* Чтобы использовать "нестандартное" имя, можно использовать флаг `-f' или `--file': **make -f <имя\_файла>** или **make --file=<имя\_файла>** - указывают программе make на необходимость использования файла с именем <имя\_файла> в качестве make-файла.
* Вы можете задать обработку сразу нескольких make-файлов, перечислив их в командной строке с помощь нескольких опций `-f' или `--file'. Все указанные таким образом make-файлы логически "объединяются" в том порядке, как они были заданы в командной строке (логически "объединяясь" в один make-файл). При наличии в командной строке опций `-f' или `--file', автоматический поиск make-файлов с именами `GNUmakefile', `makefile' и `Makefile', не производится.

***Подключение других make-файлов***

* include подключает другие makefil’ы в makefile;
* include foo \*.mk $(bar) = эквивалентны = include foo a.mk b.mk c.mk bish bash
* В начале строки могут находится дополнительные пробелы - все они будут проигнорированы. Табуляция в начале строки недопустима - make считает командами. Между словом include и началом списка файлов, а также между именами файлов необходим пробел. Лишние пробелы между именами, а также пробелы после конца директивы, игнорируются.
* Когда make обрабатывает директиву include, она приостанавливает чтение текущего make-файла и поочередно читает каждый файл из списка, указанного в директиве. Когда весь список будет прочитан, make возвращается к обработке make-файла, в котором была встречена директива include.
* Директива include может быть использована как автоматическая генерация пререквизиттов для исходных файлов. Автоматически сгенерированные пререквизиты могут быть помещены в отдельный файл, который, затем, будет включаться в основной make-файл программы. Подобная практика, в целом, выглядит более привлекательной, чем "беспорядочное" добавление новых пререквизитов в конец главного make-файла.
* `-I' и `--include-dir' - поиск по каталогам, директориям;
* Если поиск включаемого make-файла завершился неудачно, make выдает предупреждающее сообщение, которое, не является фатальной ошибкой, поскольку обработка make-файла, содержащего директиву include, еще продолжается. После того, как все включаемые файлы будут прочитаны, make попытается создать или обновить те из них, которые не существуют или устарели. Только после неудачной попытки найти способ создания отсутствующих make-файлов, ситуация будет квалифицирована как фатальная ошибка и make завершит работу.
* Если вы хотите, чтобы make просто игнорировала make-файлы, которые не существуют и не могут быть построены автоматически, используйте директиву -include: **-include имена\_файлов...**
* Работает аналогично директиве include, за исключением того, что отсутствие включаемых make-файлов <имена\_файлов> не вызывает ошибки (и не выдается предупреждающих сообщений). Для совместимости с другими версиями make, директива **-include** имеет второе, дополнительное имя **sinclude**.

***Переменная MAKEFILES***

* Если среди переменных среды (environment variables) имеется переменная с именем MAKEFILES, то ее содержимое интерпретируется как список имен (разделенных пробелами) дополнительных make-файлов, которые должны быть прочитаны перед тем, как начнут обрабатываться "основные" make-файлы.
* Аналогичен директиве include - производится поиск дополнительных make-файлов в разных каталогах. При этом, главная цель не может браться из этих файлов, а отсутствие какого-либо из них не рассматривается как ошибка.
* Нежелательно устанавливать MAKEFILES перед первым вызовом make (на самом "высоком" уровне), чтобы не создавать причудливую "смесь" из основного make-файла и файлов, перечисленных в MAKEFILES.
* Однако, если вы запускаете make без указания конкретного make-файла, дополнительные make-файлы, перечисленные в MAKEFILES могут сделать что-нибудь полезное в помощь встроенным в make неявным правилам, например, задать нужные пути поиска.
* Некоторые пользователи соблазняются возможностью автоматически устанавливать переменную MAKEFILES при входе в систему, и пишут свои make-файлы в расчете на это. Делать это категорически не рекомендуется, поскольку такие make-файлы не будут работать при попытке их использования другими пользователями. Гораздо лучше, явно подключать нужные make-файлы с помощью обычной директивы include.

***"Перекрытие" (overriding) части make-файла***

* Разные команды для обновления одной и той же цели.
* Во "внешнем" make-файле (который включает другие make-файлы) вы можете задать шаблонное правило с произвольным соответствием (match-anything pattern rule), которое будет указывать, что цели, не описанные в данном make-файле, следут поискать в другом make-файле.
* Имеется make-файл с именем `Makefile', который описывает цель `foo' (и др. цели), мы можем написать make-файл с именем `GNUmakefile', который будет содержать следующие строки:

foo:

frobnicate > foo

%: force

@$(MAKE) -f Makefile $@

force: ;

* В таком случае, при выполнении команды `make foo', make считает файл `GNUmakefile', и увидит, что для достижения цели `foo', должна быть выполнена команда `frobnicate > foo'. При выполнении команды `make bar', make увидит, что цель `bar' не описана в make-файле `GNUmakefile', и, поэтому, использует команду из шаблонного правила: `make -f Makefile bar'. Если `Makefile' содержит правило для цели `bar', то эта цель будет обновлена. Аналогично make поступит и с любыми другими целями, не описанными в `GNUmakefile'
* В данном примере, шаблонное правило содержит лишь `**%**', поэтому любая цель подходит под такой шаблон. Пререквизит `**force**' указан лишь для того, чтобы команды из данного правила выполнялись всегда - даже в том случае, если целевой файл уже существует. Правило, описывающее цель `**force**', содержит пустую команду для того, чтобы make не пыталась использовать неявное правило для обновления этой цели (иначе возник бы бесконечный цикл, поскольку для обновления `**force**', make попыталась бы использовать то же самое шаблонное правило).

***Как make читает make-файл***

* Программа GNU make работает по двухпроходной схеме. На первом проходе производится чтение всех make-файлов (в том числе и подключаемых), в ходе которого вся содержащаяся в них информация (переменные и их значения, явные и неявные правила) переводится во внутреннее представление и строится граф зависимостей для всех целей и их пререквизитов. На втором проходе, это внутреннее представление используется для определения того, какие именно цели нуждаются в обновлении и исполняются соответствующие правила.
* **Расширение (expansion) немедленное (immediate)** - производится во время первой фазы работы: это означает, make будет вычислять переменные и функции в момент считывания и "разбора" make-файла.
* **Расширение отложенное (deferred)** - не происходит "немедленно". Расширение отложенной конструкции не происходит до тех пор, пока эта конструкция не встретится позже, уже в "немедленном" контексте, либо она будет расширена на втором проходе.
* Присваивание значения переменным. Определения переменных обрабатываются следующим образом:
* немедленно = отложенно
* немедленно ?= отложенно
* немедленно := немедленно
* немедленно += отложенно или немедленно
* define немедленно

отложенно

endef

* В операторе добавления, `+=', правая часть обрабатывается "немедленно", если переменная была ранее определена как упрощенно вычисляемая (с помощью `:=') и "отложенно" в противном случае.
* Все условные конструкции (во всех формах - ifdef, ifeq, ifndef и ifneq) целиком и полностью обрабатываются "немедленно".
* Правила всегда обрабатываются одинаковым образом, независимо от их формы:
* немедленно : немедленно ; отложенно

отложенно

* То есть, разделы целей и пререквизитов обрабатываются немедленно, а обработка команд, используемых для обновления цели, всегда откладывается. Это общее правило действует для явных правил, шаблонных правил, суффиксных правил, статических шаблонных правил и простом определении пререквизитов.

***Составление правил (rules)***

* Порядок следования правил внутри make-файле не имеет значения. Исключение составляет лишь выбор **главной цели по умолчанию (default goal)** - цели, к которой стремиться make, если вы не задали ее явно. По умолчанию, главной целью становиться цель из первого правила в первом обрабатываемом make-файле.
* Если это правило содержит несколько целей, то только первая из них становится главной целью.
* Исключения: во-первых, главными целями, выбираемыми по умолчанию, не могут стать цели, имя которых начинается с точки (если только они не содержат по крайней мере одного символа `/'). И, во-вторых, из процесса выбора главной цели исключаются шаблонные правила.
* Поэтому, мы обычно пишем make-файлы таким образом, чтобы первое правило описывало процесс сборки готовой программы, или всех программ, описываемых в этом make-файле (часто, для этого используется цель с именем `**all**').

***Синтаксис правил***

* В общем виде, правило выглядит так:

*цели* : *пререквизиты*

*команда*

...

или так:

*цели* : *пререквизиты* ; *команда*

*команда*

...

* **Цели (targets)** - имена файлов, разделенные пробелами. В именах целей могут быть использованы шаблонные символы **`%`**. Для файлов, содержащихся в архиве, может быть использована специальная форма записи: **`a(m)'**, где **a** - это имя архивного файла, а **m** - имя содержащегося в нем файла. Обычно, в правиле содержится только одна цель, однако, иногда имеет смысл задать несколько целей в одном правиле.
* Строки, содержащие команды, должны начинаться с символа табуляции. Первая команда может располагаться либо в строке с пререквизитами (и отделяться от них точкой с запятой), либо в следующей строке после пререквизитов (эта строка должна начинаться с символа табуляции). В обоих случаях, результат будет один и тот же.
* **$** - используется для ссылки на переменные, для использования его в правилах, нужно писать **`$$'**.
* Длинные строки make-файла могут быть разделены на части с помощью символа **'\'**, находящегося в конце строки.
* make никак не ограничивает длину строк make-файла.
* Правило содержит информацию о двух вещах: когда следует считать, что цель "устарела", и каким образом она может быть обновлена при возникновении такой необходимости.
* **Критерий "устаревания"** вычисляется по отношению к пререквизитам, которые представляют из себя имена файлов, разделенные пробелами. В именах пререквизитов могут использоваться шаблонные символы. Пререквизиты также могут быть файлами, находящимися в архивах. Цель считается "устаревшей", если такого файла не существует, либо он "старше", чем какой-либо из пререквизитов (проверяется время последней модефикации файла).
* Смысл здесь в том, что, поскольку целевой файл строится на основе информации из файлов-пререквизитов, то изменение хотя бы одного из них может привести к тому, что содержимое целевого файла уже не будет "правильным".
* **Команды** указывают на то, каким образом следует обновлять цель. Это - просто строки (с некоторыми дополнительными возможностями), исполняемые интерпретатором командной строки (обычно `**sh**').

***Использование шаблонных символов (wildcard characters) в именах файлов***

* При использовании шаблонных символов (**wildcard characters**), с помощью одного имени можно задать целую группу файлов.
* В make шаблонными символами являются `\*', `?' и `[...]' (как в оболочке Bourne). Например, шаблон `\*.c' будет соответствовать всем файлам с суффиксом `.c', находящимся в текущей директории.
* `~' или сочетание `~/' - домашний каталог. Например, выражение `~/bin' будет означать `/home/you/bin', строка `~john/bin' будет означать `/home/john/bin'.
* Раскрытие шаблонных имен (замена их конкретным списком файлов, удовлетворяющих шаблону) автоматически производится в именах целей, именах пререквизитов и командах (в командах этим занимается интерпретатор командной строки). В других случаях, раскрытие шаблона производится только при явном запросе с помощью функции **wildcard**.
* Специальное значение шаблонных символов может быть "отключено" с помощью предшествующего им символа '**\**'. Таким образом, строка `foo\\*bar' будет ссылаться на довольно странное имя, состоящее из семи символов - начального `foo', звездочки и `bar'.
* При задании переменной, раскрытия шаблонов не производится. Например, если вы запишите:

**objects = \*.o**

* то значением переменной **objects** будет строка **`\*.o'**. Однако, если вы используете значение переменной **objects** в цели, в пререквизите или в команде, то в момент использования шаблона, будет произведено его расширение. Чтобы присвоить переменной **objects** значение, полученное после расширения шаблона, используйте функцию **wildcard:**

**objects := $(wildcard \*.o)**

***Функция wildcard***

* Шаблонные имена автоматически "расширяются" при обработке правил, где они использованы. В других случаях, например, при присваивании переменной нового значения, или в аргументах функций, такого расширения не производится. Для "принудительного" расширения шаблонных имен в любых нужных местах, предназначена функция **wildcard.**
* **$(wildcard шаблон...)** - будет заменена списком существующих в данный момент файлов, которые удовлетворяет указанному шаблону/шаблонам. Имена файлов отделяются друг от друга пробелами. Если не будет найдено файлов, удовлетворяющих шаблону, функция возвращает пустую строку (отличается от поведения обычных шаблонов в правилах, которые, в таких случаях, остаются в исходном виде, а не игнорируются).
* Одно из возможных применений фукнции **wildcard** - это получение списка исходных файлов, находящихся в текущем каталоге, например: **$(wildcard \*.c)**
* Затем, мы можем превратить список исходных файлов в список объектных файлов, заменив их расширение с `.c' на `.o': **$(patsubst %.c,%.o,$(wildcard \*.c))**
* Для замены текста, использовали функцию **patsubst**.
* Таким образом, make-файл, компилирующий все файлы с исходными текстами на языке Си из текущего каталога, и, затем, компонующий их вместе, может выглядеть так:

objects := $(patsubst %.c,%.o,$(wildcard \*.c))

foo : $(objects)

cc -o foo $(objects)

* В этом make-файле для компиляции исходных текстов используются неявные правила компиляции программ на языке Си, поэтому нет необходимости в явном описании правил компиляции. Оператор `:=' является вариантом "стандартного" оператора `='.

***Поиск пререквизитов по каталогам***

* Значение переменной **VPATH** указывает утилите make список директорий, где следует производить поиск файлов.
* Если файл, не найден в текущем каталоге, make предпримет попытку найти его в каталогах, перечисленных в **VPATH**.
* Имена перечисленных в **VPATH** каталогов, отделяются друг от друга пробелами или двоеточиями. При поиске, make перебирает каталоги в том порядке, как они перечислены в переменной **VPATH**. В операционных системах MS-DOS и MS-Windows для разделения имен директорий вместо символа двоеточия, должен использоваться символ точки с запятой (поскольку символ ':' может быть частью названия каталога, находясь после имени диска).
* **VPATH** **= src:../headers**

указывает, что путь поиска состоит из двух каталогов, `src' и `../headers'. Поиск в этих каталогах производится в указанном порядке.

* При таком значении переменной VPATH, следующее правило,
* **foo.o : foo.c**

будет интерпретировано так, как будто он записано следующим образом:

* **foo.o : src/foo.c**

если предположить, что файл `foo.c' находится не в текущей директории, а в каталоге `src'.

***Директива vpath***

* vpath - аналогична переменной VPATH, но более "избирательна". Эта директива позволяет задать пути поиска именно, для файлов, чье имя подходит под определенный шаблон. Таким образом, можно задать список каталогов поиска для одной группы файлов, и совсем другой - для других файлов.
* Имеется три формы записи директивы vpath:
* **vpath <шаблон> <каталоги>** - задать путь поиска <каталоги> для файлов, чье имя удовлетворяет шаблону <шаблон>. Путь поиска <каталоги> - это список имен директорий, разделенных двоеточиями (точкой запятой в MS-DOS или MS-Windows) или пробелами, подобно списку поиска переменной VPATH.
* **vpath <шаблон>** - очистить ("забыть") пути поиска для шаблона <шаблон>.
* **vpath** - очистить ("забыть") все пути поиска, ранее определенные с помощью директивы vpath
* В директиве vpath, шаблон представляет собой строку, содержащую символ `**%**'. Имя искомого файла должно соответствовать этой шаблонной строке, причем символ `%', как и в шаблонных правилах, может соответствовать любой последовательности символов (в том числе и пустой). Например, шаблону %.h удовлетворяют имена файлов, имеющие расширение .h.
* Если искомого пререквизита нет в текущей директории, а его имя удовлетворяет шаблону, указанному в директиве vpath, предпринимается попытка найти его в каталогах, список которых указан в этой директиве. Поиск проходит аналогично поиску в каталогах, перечисленных в переменной VPATH, и предшествует ему.
* **vpath %.h ../headers**

инструктирует make, производить поиск пререквизитов с расширением `.h' в каталоге `../headers', если они не могут быть найдены в текущей директории.

* Если имя искомого пререквизита подходит сразу под несколько шаблонов, указанных в директивах vpath, make обрабатывает эти директивы поочередно, друг за другом, производя поиск во всех каталогах, перечисленных в каждой из них. Отдельные директивы vpath обрабатываются в том порядке, как они расположены в make-файле; несколько директив с одинаковым шаблоном никак не влияют друг на друга. Например, в случае:

**vpath %.c foo**

**vpath % blish**

**vpath %.c bar**

* поиск файла с раширением `.c' будет происходить в каталоге `foo', затем `blish', и, наконец `bar', а в случае:

**vpath %.c foo:bar**

**vpath % blish**

* поиск такого файла будет производиться в каталоге `foo', затем `bar', и затем `blish'.

***Процедура поиска по каталогам***

* Если файл, являющийся пререквизитом, найден с помощь поиска в каталогах (независимо от типа поиска - "общего" или "избирательного"), найденный путь к этому файлу не всегда будет присутствовать в имени пререквизита, которое передаст вам make. В некоторых случаях найденный путь "отбрасывается" и не используется.
* Для переменной **GPATH** используется такой же синтаксис, что и для **VPATH** (список имен каталогов, разделенных пробелами или двоеточиями). Если "устаревший" целевой файл был найден в результате проведения поиска по каталогам, и найденный путь присутствует в списке **GPATH**, он не будет "отброшен". Далее, цель будет обновлена именно по этому пути.

***Написание команд с учетом поиска по каталогам***

* Тот факт, что пререквизит был найден с помощью поиска по каталогам, никак не влияет на исполняемые команды правила - они будут исполнены именно в том виде, как они записаны в make-файле. Имея это ввиду, следут внимательно отнестись к написанию команд - файлы, которые является пререквизитами, должны браться командами из тех каталогов, где они были найдены программой make.
* Это может быть сделано с использованием **автоматических переменных**.
* `**$^**' - список всех пререквизитов правила с именами каталогов, где эти пререквизиты были найдены;
* `**$@**' - имя цели

foo.o : foo.c

cc -c $(CFLAGS) **$^** -o **$@**

* `**$<**' - содержит лишь первый пререквизит правила.
* `**$\***` - основа имени.

VPATH = src:../headers

foo.o : foo.c defs.h hack.h

cc -c $(CFLAGS) **$<** -o **$@**

***Поиск в каталогах для подключаемых библиотек***

* Поиск в каталогах может производиться специальным образом для файлов, являющихся библиотеками. Эта специфическая особенность вступает в силу для пререквизитов, чье имя имеет специальную форму `-lимя'. (Возможно, вам это покажется странным, поскольку пререквизит, обычно, является именем файла, а файл библиотеки <имя>, как правило, называется `libимя.a', а не `-lимя'.)
* Когда имя пререквизита имеет форму `-lимя', make обрабатывает ее специальным образом, производя поиск файла с именем `libимя.so' сначала в текущей директории, затем в каталогах, перечисленных в подходящих директивах vpath, каталогах из VPATH, и, наконец, в каталогах `/lib', `/usr/lib', и `prefix/lib' (обычно `/usr/local/lib', но версии make для операционных систем MS-DOS/MS-Windows ведут себя так, как если бы prefix был корневым каталогом, где инсталлирован компилятор DJGPP).
* Если такой файл не обнаружен, предпринимается попытка найти файл `libимя.a' (в перечисленных выше каталогах).
* Так, если в вашей системе имеется библиотека `/usr/lib/libcurses.a' (и отсутствует файл `/usr/lib/libcurses.so'), то в следующем примере:

**foo : foo.c -lcurses**

**cc $^ -o $@**

* будет выполнена команда `cc foo.c /usr/lib/libcurses.a -o foo' если `foo' "старше" чем `foo.c' или `/usr/lib/libcurses.a'.
* Хотя, по умолчанию проводится поиск файлов с именами `libимя.so' и `libимя.a', это поведение может быть изменено с помощью переменной **.LIBPATTERNS**. Каждое слово в значении этой переменной рассматривается как шаблонная строка. Встретив пререквизит вида `-lимя', make заменяет символ процента в каждом из шаблонов на <имя> и производит описанную выше процедуру поиска для полученного имени библиотечного файла. Если библиотечный файл не найден, используется следующий шаблон из списка и так далее.
* По умолчанию, значением переменной **.LIBPATTERNS** является строка "**`lib%.so lib%.a'**", которая и обеспечивает описанное выше поведение.
* Присвоив этой переменной пустое значение, вы можете полностью отключить описанный механизм поиска подключаемых библиотек.

***Абстрактные цели (phony targets)***

* **Абстрактная цель (phony target)** - цель, которая не является, на самом деле, именем файла. Это просто имя для некоторой последовательности команд, которую при необходимости может выполнить make.
* Есть по крайней мере два соображения в пользу использования абстрактных целей: их использование позволяет избежать конфликтов с файлами, имеющими такое же имя, а также ускорить работу make.

clean:

rm \*.o temp

* **правило с такой "псевдо-целью"** откажется работать, если в текущем каталоге по какой-нибудь причине окажется файл с именем `clean'. Поскольку в правиле не указано каких-либо пререквизитов, файл `clean' всегда будет считаться "новым" и команды, указанные в правиле никогда не выполнятся. Во избежании подобной проблемы, вы можете прямо указать, что некоторая цель является абстрактной. Для этого используется специальная цель **.PHONY** (сначала должна идти строка, объявляющая clean абстрактной целью, а затем уже следует правило, описывающее эту цель).

.PHONY: clean

clean:

rm \*.o temp

* После этого, вызов `make clean' будет приводить к исполнению нужных команд, независимо от того, существует файл `clean' или нет.
* Поскольку абстрактные цели не являются файлами, которые могут быть обновлены при изменении других файлов, make не предпринимает попыток применить неявные правила для таких целей. В результате, использование абстрактных целей может ускорить обработку make-файла.
* **Рекурсивный вызов make**. В таких случаях, как правило, в make-файле имеется переменная, хранящая список подкаталогов с "подчиненными" проектами, которые должны быть собраны. Далее, один из возможных вариантов - создание правила, где с помощью интерпретатора командной строки организуется цикл, выполняющий поочередную обработка всех подкаталогов, например:

SUBDIRS = foo bar baz

subdirs:

for dir in $(SUBDIRS); do \

$(MAKE) -C $$dir; \

done

* Такому методу, однако, присущи некоторые недостатки. Во-первых, любые ошибки, возникшие при обработке подпроектов, останутся "незамеченными" - при возникновении ошибки в подпроекте make будет продолжать обработку оставшихся подкаталогов "как ни в чем ни бывало". Разумеется, в цикл можно ввести дополнительный код, который будет детектировать ошибочные ситуации и прерывать работу. К сожалению, при запуске make с опцией -k, такое поведение будет нежелательно. Второй недостаток, (возможно, более серьезный) состоит в том, что при таком подходе нельзя задействовать возможность "параллельной" работы make (из-за наличия единственного правила).
* Объявив подкаталоги абстрактными целями (вы должны это сделать, так как подкаталоги проектов обычно уже существуют и иначе они не стали бы обрабатываться) вы можете решить эти проблемы:

SUBDIRS = foo bar baz

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

$(SUBDIRS):

$(MAKE) -C $

foo: baz

* Мы также объявили, что подкаталог `foo' не может быть обработан до тех пор, пока не будет закончена обработка подкаталога `baz'; подобного рода декларация потребуется для случая "параллельной" работы.
* Как правило, абстрактная цель не должна быть пререквизитом какого-либо целевого файла, в противном случае указанные в подобном правиле команды будут исполняться всякий раз при его обработке. Если абстрактная цель не является пререквизитом какого-либо реального файла, команды из правила, где она описана, будут исполняться только в том случае, когда эта цель будет указана в качестве главной.
* Абстрактные цели могут иметь пререквизиты. Например, когда в одном каталоге содержится сразу несколько собираемых программ, удобно хранить их описания в одном make-файле. Так как главной целью по умолчанию становится первая цель из make-файла, в таких случаях, обычно, первым правилом make-файла делают правило с абстрактной целью `all', пререквизитами которой являются все собираемые программы. Например:

all : prog1 prog2 prog3

.PHONY : all

prog1 : prog1.o utils.o

cc -o prog1 prog1.o utils.o

prog2 : prog2.o

cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o

cc -o prog3 prog3.o sort.o utils.o

* Теперь вам достаточно сказать `make', чтобы обновить все три программы, или указать нужные аргументы для обновления конкретных программ (например, `make prog1 prog3').
* Когда одна абстрактная цель является пререквизитом другой абстрактной цели, она работает как своего рода "подпрограмма". В следующем примере, `make cleanall' удалит объектные файлы, diff-файлы, и файл `program':

.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff

rm program

cleanobj :

rm \*.o

cleandiff :

rm \*.diff

***Правила без команд и пререквизитов***

* Если правило не имеет команд и пререквизитов, а целью этого правила является имя несуществующего файла, то каждый раз, при обработке такого правила, make будет считать что его цель нуждается в обновлении. Если эта цель, в свою очередь, является пререквизитом каких-либо правилах, то указанные в них команды всякий раз будут выполняться.

clean: FORCE

rm $(objects)

FORCE:

* Здесь, цель `FORCE' удовлетворяет специальным условиям (не имеет пререквизитов и команд). Цель `clean' зависит от `FORCE', поэтому команды из правила с `clean' вынуждены будут выполняться. В имени `FORCE' нет ничего "необычного", просто оно часто используется для подобных целей.
* Очевидно, что такое использование `FORCE' эквивалентно объявлению цели clean абстрактной, с помощью `.PHONY: clean'.
* Подход с использованием `.PHONY' более понятен и эффективен, однако другие версии make могут не поддерживать `.PHONY'. В силу этой причины, во многих make-файлах используется `FORCE'.

***Использование пустых целей (empty target files) для фиксации событий***

* **Пустая цель (empty target)** является вариантом абстрактной цели - используются для хранения команд, исполняющих действие, выполнение которого вам может иногда потребоваться. В отличие от абстрактных целей, пустая цель действительно может существовать в виде файла. Однако, содержимое такого файла никоим образом не используется, и, зачастую, он просто пуст.
* Назначение подобной цели - запомнить (с помощью времени последней модификации), когда последний раз исполнялись указанные в правиле команды. Это делается при помощи включения в список команд, содержащихся в правиле, команды touch, обновляющей эту цель.
* Пустая цель должна иметь какие-нибудь пререквизиты (иначе в ней нет смысла). Когда вы запрашиваете обновление этой цели, команды из ее правила будут выполняться, если какой-либо из пререквизитов "новее", чем цель. Другими словами, команды будут выполняться если какой-либо из пререквизитов был обновлен со времени последнего обновления цели.

print: foo.c bar.c

lpr -p $?

touch print

* С таким правилом, `make print' приведет к выполнению команды lpr, если какой-нибудь из исходных файлов был изменен с момента последнего вызова `make print'. Автоматическая переменная `**$?**' использована для того, чтобы печатать только те исходные файлы, которые были изменены.

***Имена специальных целей***

* .PHONY - пререквизиты специальной цели .PHONY объявляются абстрактными целями. При необходимости обновления таких целей, make будет выполнять команды "безусловно", независимо от того, существует ли файл с таким именем, и времени, когда он был модифицирован.
* .SUFFIXES - пререквизиты специальной цели .SUFFIXES представляют собой список суффиксов (расширений) имен файлов, которые будут использоваться при поиске суффиксных правил.
* .DEFAULT - команды, определенные для цели .DEFAULT, будут использованы со всеми целями make-файла, для которых не найдено ни явных, ни неявных правил. Команды, определенные для .DEFAULT, будут использованы для всех пререквизитов, не являющихся целями каких-либо правил.
* .PRECIOUS - цели, перечисленные в качестве пререквизитов .PRECIOUS, подвергаются специальной обработке. В том случае, если make будет принудительно завершена или прервана во время исполнения команд для их обновления, эти цели не будут удалены. Также, если цель является "промежуточным" файлов, он не будет, как обычно, удаляться после того, как необходимость в нем отпала. В последнем случае, эта специальная цель работает подобно .SECONDARY. В качестве пререквизита .PRECIOUS может быть указан шаблон имени (например, `%.o'), что позволит сохранять все промежуточные файлы с именами, удовлетворяющими этому шаблону.
* .INTERMEDIATE - пререквизиты цели .INTERMEDIATE рассматриваются как промежуточные файлы. .INTERMEDIATE без списка пререквизитов не производит никакого эффекта.
* .SECONDARY - цели, указанные в качестве пререквизитов .SECONDARY рассматриваются как промежуточные файлы, за исключением того, что они никогда не удаляются автоматически. .SECONDARY без указания пререквизитов, помечает таким образом все цели, перечисленные в make-файле.
* .DELETE\_ON\_ERROR - при наличии в make-файле цели с именем .DELETE\_ON\_ERROR, make будет удалять цель правила если она была модифицирована, а обновляющая ее команда завершилась с ненулевым кодом возврата; аналогично, цель будет удаляться при прерывании работы make.
* .IGNORE - make будет игнорировать ошибки при выполнении команд, обновляющих цели, перечисленные в качестве пререквизитов .IGNORE. Команды, указываемые для .IGNORE, не имеют значения. Использование .IGNORE без списка пререквизитов, означает необходимость игнорирования ошибок во всех командах, исполняемых для обновления любой цели make-файла. Такое использование `.IGNORE' поддерживается только по историческим соображениям для обеспечения совместимости. Этот прием не слишком полезен, поскольку воздействует на любую команду make-файла; вместо этого, мы рекомендуем использовать более "избирательный" метод, позволяющий игнорировать ошибки в конкретных командах.
* .SILENT - если вы указали некоторые цели в качестве пререквизитов .SILENT, то в процессе обновления этих целей, make не будет печатать выполняемые при этом команды. Указываемые для .SILENT команды не имеют значения. В случае использования .SILENT без списка пререквизитов, будет отключена печать всех исполняемых команд. Такое использование `.SILENT' поддерживается только по историческим причинам, для обеспечения совместимости. Мы рекомендуем использовать более избирательный метод для подавления печати отдельных команд. Временно подавить печать исполняемых команд можно, запуская make с опциями `-s' или `--silent'.
* .EXPORT\_ALL\_VARIABLES - будучи просто упомянутой в качестве цели, указывает make на необходимость, по умолчанию, экспортировать все переменные для возможности их использования в дочернем процессе.
* .NOTPARALLEL - при наличии в make-файле цели .NOTPARALLEL, данный экземпляр make будет работать "последовательно" (даже при наличии опции `-j'). Рекурсивно запущенные экземпляры make по-прежнему могут работать "параллельно" (если только их make-файлы не содержат такой же специальной цели). Любые пререквизиты данной цели игнорируются.
* Любой суффикс, определенный для суффиксных правил, а также "сцепление" двух суффиксов (например, такое как `.c.o'), находясь на месте цели, рассматриваются специальным образом. Такие цели представляют из себя суффиксные правила - устаревший, однако, по-прежнему, широко распространенный способ задания шаблонных правил. На практике, суффиксы обычно начинаются с символа `.', поэтому такие специальные цели также начинаются с `.'.

***Несколько правил с одной целью***

* Возможно, простейший путь указать, что все объектные файлы должны быть перекомпилированы при изменении `config.h' - это написать:

objects = foo.o bar.o

foo.o : defs.h

bar.o : defs.h test.h

$(objects) : config.h

* Подобная запись хороша тем, что может быть легко добавлена в make-файл или удалена из него, не затрагивая "основные" правила, используемые для генерации объектных файлов. Это удобно при необходимости "срочно" добавить в make-файл еще несколько пререквизитов.
* Другой возможный прием заключается в том, чтобы передавать список дополнительных пререквизитов в переменной, значение которой устанавливать в командной строке при вызове make.

extradeps=

$(objects) : $(extradeps)

* вызов **`make extradeps=foo.h'** будет добавлять `foo.h' в список пререквизитов каждого из объектных файлов. При обычном вызове `make', этого делаться не будет.
* Если ни одно из правил, описывающее цель, не имеет команд, make попытается применить к этой цели неявные правила.

***Статические шаблонные правила (static pattern rules)***

* **Статические шаблонные правила (static pattern rules)** - это правила с несколькими целями, и возможностью автоматически создавать список пререквизитов для каждой цели, используя ее имя. Это - механизм более общий, чем обычные правила с несколькими целями, потому что их цели не должны иметь идентичные пререквизиты. Их пререквизиты должны быть похожими, но необязательно идентичными.
* Для статических шаблонных правил используется следующий синтаксис:

*цели* ...: *шаблон-цели*: *шаблоны-пререквизитов* ...

*команды*

...

objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c

$(CC) -c $(CFLAGS) $< -o $@

* В списке **целей** перечисляются **цели**, к которым будет применяться данное правило - при задании имен целей также могут использоваться шаблонные символы.
* **Шаблон-цели** и **шаблоны-пререквизитов** описывают, как вычислять список пререквизитов для каждой цели. Каждая цель статического шаблонного правила сопоставляется с **шаблоном-цели**, для получения части имени **цели**, называемой основой. Далее, полученная основа имени подставляется в каждый из шаблонов-пререквизитов для получения имен пререквизитов (по одному имени из каждого шаблона-пререквизита).
* Обычно, в каждом шаблоне содержится по одному символу `%'. Когда цель сопоставляется с шаблоном-цели, символ `%' может соответствовать любой части имени цели; именно эта часть будет являться основой. Прочие части имени цели должны в точности совпадать с шаблоном. Например, цель `foo.o' удовлетворяет шаблону `%.o' и ее основой будет `foo'. Цели же `foo.c' и `foo.out' не будут удовлетворять этому шаблону.
* Имена пререквизитов для каждой цели генерируются путем подстановки основы вместо символа `%' в каждом из шаблонов пререквизитов. Например, из одного шаблона пререквизита `%.c' и основы `foo', будет получено имя пререквизита `foo.c'. Шаблоны пререквизитов, не содержащие символа `%' также вполне допустимы, в этом случае, указанный пререквизит будет одинаков для всех целей.
* Каждая перечисленная в правиле **цель**, должна удовлетворять **шаблону цели**, иначе будет выдано соответствующее предупреждение. Если у вас имеется список файлов, лишь некоторые из которых удовлетворяют шаблону, вы можете удалить неподходящие имена с помощью функции **filte**r:

files = foo.elc bar.o lose.o

$(filter %.o,$(files)): %.o: %.c

$(CC) -c $(CFLAGS) $< -o $@

$(filter %.elc,$(files)): %.elc: %.el

emacs -f batch-byte-compile $<

* В этом примере, результатом **`$(filter %.o,$(files))'** является `**bar.o lose.o'**, и первое статическое правило вызывает компиляцию этих объектных файлов из соответствующих им исходных файлов. Результатом выражения **`$(filter %.elc,$(files))'** является **`foo.elc'**, и этот файл получается из **`foo.el'**.
* Следующий пример иллюстрирует использование автоматической переменной **$\*** в статическом шаблонном правиле:

bigoutput littleoutput : %output : text.g

generate text.g -$\* > $@

* При запуске команды generate, ссылка на **$\*** будет заменена соответствующей основой имени - строкой `big' или `little'.

***Написание команд***

* Определенные в правиле команды, представляют собой текстовые строки с командами для интерпретатора командной строки. Команды эти исполняются последовательно, одна за другой. Каждая строка, содержащая команду, должна начинаться с символа табуляции. Первая команда также может быть расположена в строке правила, содержащей список целей и пререквизитов - в таком случае она отделяется от списка пререквизитов символом точки с запятой и не требует наличия символа табуляции в ее начале. Среди строк, содержащих команды, могут присутствовать пустые строки и строки, содержащие лишь комментарии - все они будут проигнорированы. (Но имейте ввиду, что "пустая" строка, начинающаяся с символа табуляции, на самом деле не является пустой! Она рассматривается как строка, содержащая пустую команду; смотрите раздел Пустые команды.)
* Несмотря на то, что пользователи могут пользоваться разными интерпретаторами командной строки, для интерпретации команд из make-файла всегда используется оболочка `/bin/sh'(если только в make-файле явно не задается использование какой-либо другого интерпретатора).
* Тип используемой в системе оболочки определяется исходя из того, может ли в командных строках использоваться комментарий и используемый для этого синтаксис. В оболочке `/bin/sh', комментарий начинается с символа `#' и продолжается до конца строки. Символ `#' может располагаться и не в начале строки.

***Отображение исполняемых команд (command echoing)***

* Обычно, make печатает каждую командную строку перед тем, как она будет выполнена - **механизм эхо (echoing)**, поскольку он создает впечатление, что это вы сами набираете исполняемые команды.
* Если строка, содержащая команду, начинается с символа `@', печать этой команды не производится. Символ `@' удаляется из строки с командой перед тем, как она передается для обработки интерпретатору командной строки. Такой прием часто используется для команд, назначением которых является вывод некоторого сообщения, например, команд echo, используемых для отображения хода обработки make-файла:

**@echo About to make distribution files**

* Когда make вызывается с опцией **`-n'** или **`--just-print'**, происходит только лишь отображение команд, без их реального выполнения. Это единственный случай, когда команды, начинающиеся с символа **`@'**, также будут напечатаны. Используя эти опции, можно увидеть, какие команды make считает необходимым выполнить, без того, чтобы реально их выполнять.
* Опции **`-s'** и **`--silent'** отключают всякое отображение команд, как если бы все команды начинались с символа **`@'**. Использование в make-файле правила со специальной целью .SILENT без указания пререквизитов, имеет аналогичный эффект. Использование специальной цели .SILENT является устаревшей практикой, взамен которой мы рекомедуем пользоваться более гибким механизмом - символом `@'.

***Исполнение команд***

* Последовательность команд, обновляющих цель, исполняется путем вызова отдельного экземпляра интерпретатора командной строки для каждой из строк make-файла, содержащих команды.
* Это, в частности, означает, что **(обратите внимание!)** такие команды, как cd, влияющие на переменные среды процесса, не окажут никакого влияния на следующие за ними команды. (2) Если вы хотите, чтобы команда cd повлияла на следующую за ней команду, поместите обе команды на одну строку make-файла, отделив друг от друга с помощью точки с запятой. В таком случае, make будет рассматривать их как единую команду и "вместе" передаст их интерпретатору командной строки для последовательного выполнения.

foo : bar/lose

cd bar; gobble lose > ../foo

* Для повышения удобочитаемости, вы можете разбить длинные строки с командами на несколько частей с помощью символа '\'. Его нужно поместить в конец каждой строки-фрагмента, за исключением последней. Перед вызовом интерпретатора командной строки, подобная последовательность строк будет скомбинирована в одну строку, путем удаления конечных символов '\'.

foo : bar/lose

cd bar; \

gobble lose > ../foo

* Имя программы, являющейся интерпретатором командной строки, берется из переменной SHELL. По умолчанию, используется программа `/bin/sh'.

***Ошибки при исполнении команд***

* После завершения очередной команды, make проверяет полученный от нее код возврата. В случае успешного ее завершения, выполняется следующая командная строка; после выполнения последней команды, обработка правила считается завершенной.
* Если во время выполнения команды произошла ошибка (от нее был получен ненулевой код возврата), make прекращает обработку текущего правила, и, возможно, прерывает работу.
* В определенных ситуациях, ошибка при выполнении некоторой команды не является проблемой - команда mkdir, дабы быть уверенным в существовании каталога. Если такая директория уже существует, команда mkdir сообщит об ошибке, но, скорее всего, вы захотите, чтобы make в таком случае продолжила работу, не обращая внимания на ошибку.
* Для того, чтобы проигнорировать ошибки в команде, поместите в начало строки (после символа табуляции), где она описана, символ `-'. Перед тем, как эта команда будет передана интерпретатору командной строки, символ `-' будет из нее удален.

clean:

-rm -f \*.o

* обработка make-файла не будет прервана даже в том случае, если команда rm не сможет удалить файл.
* При запуске make с опцией **`-i**' или **`--ignore-errors'**, будут игнорироваться ошибки во всех командах, любого из правил. Такой же эффект достигается при использовании специальной цели **.IGNORE** (устаревшая практика) в правиле, не имеющем пререквизитов.
* Когда произошедшая ошибка игнорируется (вследствие использования `-' или опции `-i'), ошибочное завершение команды обрабатывается аналогично нормальному завершению, за исключением того, что печатается полученный от команды код возврата и выдается сообщение, что ошибка была проигнорирована.
* При возникновении ошибки, насчет которой make не имеет инструкций о необходимости ее игнорирования, считается что текущая цель не может быть корректно обновлена. По этой причине, любые другие цели, прямо или косвенно зависящие от нее, также не могут быть достигнуты- и команды, обновляющие эти цели, исполняться не будут, поскольку не будут выполняться их предварительные условия.
* Обычно, в такой ситуации make прекращает работу, возвращая ненулевой результат. Однако, если была указана опция **`-k'** или **`--keep-going'**, make продолжает обработку других пререквизитов оставшихся целей, при необходимости обновляя их. Например, после неудачной компиляции объектного файла, `make -k' продолжит работу, компилируя оставшиеся объектные файлы, хотя уже заранее известно, что их компоновка закончится неудачей. По окончанию работы make, ненулевой код возврата будет указывать на произошедшую ошибку.
* Мотивы подобного поведения make просты. Обычно, запуская make, вы хотите получить свежую версию указанной цели. Как только make понимает, что это невозможно, она сразу же может сообщить о происшедшей ошибке. Напротив, задание опции `-k' означает, что ваша цель - наиболее полно протестировать процесс сборки программы, по возможности обнаружив максимальное количество проблем. После устранения всех найденных проблем, можно будет заново повторить процесс компиляции.
* Обычно, если при аварийном завершении команды, целевой файл все же обновился, то он, скорее всего просто поврежден и непригоден для дальнейшего использования. В любом случае, он, как минимум, обновился некорректно. Однако, поскольку время модификации файла все-таки изменилось, при следующем запуске, make уже не будет обновлять этот файл, считая, что имеется его "свежая" версия. Похожая ситуация возникает, когда исполняемая команда аварийно завершается из-за получения сигнала. Пожалуй, правильнее всего будет удалять целевой файл, если обновляющая его команда завершилась с ошибкой.
* Утилита make будет поступать подобным образом при наличии в make-файле специальной цели **.DELETE\_ON\_ERROR**. Практически во всех случаях подобное поведение является наилучшей стратегией, однако, по умолчанию, make этого не делает (из "исторических" сображений). Для того, чтобы make автоматически удаляла некорректно построенные целевые файлы, вы должны явно этого потребовать.

***Рекурсивный вызов make***

* При рекурсивном использовании, программа make сама выступает в качестве одной из команд make-файла. Подобная техника полезна, когда вы хотите иметь отдельные make-файлы для различных подсистем, составляющих большую систему. Предположим, у вас имеется подкаталог `subdir', содержащий свой собственный make-файл, и вы хотите, чтобы make-файл из "объемлющего" каталога запускал make в этом подкаталоге. Вы можете сделать это, написав:

subsystem:

cd subdir && $(MAKE)

или:

subsystem:

$(MAKE) -C subdir

* Для удобства, GNU make записывает имя текущего рабочего каталога в переменную CURDIR. В случае, если make была запущена с параметром -C, эта переменная будет содержать имя нового (установленного с помощью параметра -C) каталога, а не "оригинального" рабочего каталога. Переменная CURDIR имеет "приоритет" такой же, как если бы она была установлена внутри make-файла (по умолчанию, переменная среды CURDIR не будет "перекрывать" ее значение). Если вы сами запишете в переменную CURDIR новое значение, это никак не повлияет на работу make.

***Как работает переменная MAKE***

* При рекурсивном использовании make, вместо "прямого" указания имени команды (`make'), всегда следует использовать переменную MAKE, как показано в следующем примере:

subsystem:

cd subdir && $(MAKE)

* Эта переменная содержит имя исполняемого файла, запущенного в ответ на команду make. Если, например, этот файл назывался **`/bin/make'**, то в приведенном выше примере будет вызвана команда **`cd subdir && /bin/make'**. Таким образом, при каждом рекурсивном вызове, будет использована та же самая программа make, которая была вызвана для make-файла "верхнего" уровня.
* Командные строки с переменной **MAKE** выполняются обычными образом, не обращая внимания на опции, отключающие выполнение команд. Для передачи параметров от make (опции **`-t', `-n' и `-q'**) "высшего" уровня на "нижние" уровни используется обычный механизм - переменная **MAKEFLAGS**. Таким образом, все ваши запросы на обновление даты модификации целевых файлов или печать исполняемых команд, будут переданы "вниз", во все подсистемы.

***Связь с make "нижнего уровня" (sub-make) через переменные***

* Значения переменных, определенных в make "верхнего уровня", могут быть переданы в "порожденные" make через среду (как переменные среды), при явном на то указании. Эти переменные будут определены и в "порожденных" make, однако их значение может быть "перекрыто" другим значением, устанавливаемым в самом make-файле "нижнего уровня" (если только не использовать опцию `-e');
* Чтобы "передать вниз" или экспортировать переменную, make добавляет переменную с таким же именем и значением в набор переменных среды перед выполнением каждой команды. В свою очередь, "порожденные" копии make, будут использовать значения переменных среды для инициализации своих внутренних таблиц переменных.
* За исключением случаев явного указания, make экспортирует только те переменные, которые "изначально" были определены в среде, либо те из них, которые были определены с помощью командной строки (и только в том случае, если имя переменной состоит только из букв, цифр и символов подчеркивания, поскольку прочие имена могут "не работать" с некоторыми версиями командных интерпретаторов).
* Специальные переменные SHELL и MAKEFLAGS экспортируются всегда (за исключением случаев, когда вы "принудительно" запретили их экспорт). Переменная MAKEFILES экспортируется в том случае, если вы присвоили ей какое-либо значение.
* Переменные, определенные с помощью командной строки, автоматически передаются "вниз", поскольку make помещает их (наряду с другими параметрами) в специальную переменную MAKEFLAGS.
* Переменные, которые по умолчанию созданы самой make, не передаются "вниз". Такие переменные каждая make "нижнего уровня", при необходимости, создаст самостоятельно.
* Для экспорта указанной переменной в make "нижнего уровня", используется директива **export.** Для запрета экспорта переменной, используется директива **unexport.**
* Для удобства, вы можете одновременно определить переменную и указать на необходимость ее экспортирования. Это делается с помощью записи:
* **export переменная = значение** - что эквивалентно: **переменная = значение**
* **export переменная** - или: **export переменная := значение** - что эквивалентно: **переменная := значение**
* Вы можете заметить, что директивы **export и unexport** работают в make таким же образом, как и подобные директивы командного интерпретатора sh.
* Если вы хотите, чтобы, по умолчанию, все переменные экспортировались, используйте директиву **export** без аргументов.
* Эта конструкция говорит о том, что все переменные, которые не были явно указаны в директивах export и unexport, должны быть экспортированы. Любые переменные, перечисленные в директиве unexport, по-прежнему не будут экспортироваться. При использовании директивы export без параметров, переменные, чьи имена содержат не только алфавитно-цифровые символы и подчеркивания, экспортированы не будут. Для экспорта таких переменных надо явно указать их в директиве export.
* Старые версии GNU make, по умолчанию, экспортируют все переменные (как если бы была использована директива export без параметров). Если ваш make-файл рассчитан на такое поведение и вы хотите, чтобы он оставался "совместим" со старыми версиями make, то вместо директивы export без параметров, можно использовать правило со специальной целью **.EXPORT\_ALL\_VARIABLES**. Старые версии make просто проигнорируют такое правило, в то время как использование директивы export вызвало бы синтаксическую ошибку.
* Аналогично, вы можете использовать директиву unexport без параметров для того, чтобы, по умолчанию, не экспортировать переменные. Поскольку именно так, по умолчанию, и ведет себя make, необходимость в директиве unexport без параметров может возникнуть только в случае, если ранее где-то была использована директива export без параметров (возможно, в каком-нибудь из включаемых make-файлов). Вы не можете использовать директивы export и unexport без параметров для того, чтобы экспортировать переменные для одних команд и не экспортировать для других. Сработает самая "последняя" из перечисленных в make-файле директив export или unexport, которая и будет определять поведение make на все время обработки make-файла.
* Специальная переменная **MAKELEVEL** используется для отражения "уровня вложенности" make. Ее значение меняется при переходе "с уровня на уровень". Значением этой переменной является строка с десятичным числом, показывающим "уровень вложенности" данной копии make. Для make самого "верхнего" уровня, ее значением является `0'. Далее, во "вложенной" копии make ее значением будет `1', следующая "вложенная" копия make получит значение `2' и так далее. Значение этой переменной увеличивается в тот момент, когда make устанавливает переменные среды для запуска очередной команды.
* В основном, переменная **MAKELEVEL** применяется в условных директивах; с ее использованием вы можете написать make-файл, который будет вести себя по-разному в зависимости от того, был ли он запущен непосредственно вами, либо исполнялся рекурсивно вызванной копией make.
* Для передачи во "вложенные" копии make дополнительного списка make-файлов, которые нужно интерпретировать, вы можете использовать переменную **MAKEFILES**. Значением этой переменной является список имен make-файлов, разделенных пробелами. Будучи определенной в make-файле "высшего уровня", эта переменная будет передаваться "вниз" через переменные среды и будет работать как список make-файлов, которые должны быть прочтены "порожденными" копиями make перед чтением основного make-файла.

***Передача опций в make "нижнего уровня"***

* Такие опции как **`-s' и `-k'** автоматически передаются в "порожденные" make с помощью переменной **MAKEFLAGS**. Эта автоматически устанавливаемая переменная содержит имена всех опций, переданных программе make в командной строке. Например, при вызове **`make -ks'**, переменная **MAKEFLAGS** получит значение **`ks'**.
* Далее, каждая "порожденная" копия make получит значение переменной **MAKEFLAGS** через переменные среды и интерпретирует ее содержимое как набор опций для своей работы (аналогично тому, как если бы эти опции были переданы через командную строку).
* Аналогично, переменные, определенные с помощью командной строки, передаются в "порожденные" make через переменную **MAKEFLAGS**. Слова (из переменной **MAKEFLAGS**), содержащие символ **`='**, make рассматривает как определение переменной (аналогично тому, как если бы она были определена с помощью командной строки).
* Опции **`-C', `-f', `-o', и `-W'** не записываются в переменную **MAKEFLAGS** и, соответственно, не передаются в "порожденные" копии make.
* Опция `-j' обрабатывается специальным образом. Если в этой опции вы задали некоторое числовое значение `N', то при наличии в вашей операционной системе соответствующих возможностей (присутствующих в большинстве UNIX системах; в других системах, обычно, отсутствуют), make "верхнего уровня" и "подчиненные" make взаимодействуют между собой, контролируя общее число запущенных во всех копиях make заданий, не допуская ситуацию, когда оно превысит `N'. Обратите внимание, что задания, помеченные как рекурсивно исполняемые, при подсчете общего количества заданий не учитываются (иначе, может получиться, что у нас запущено `N' "порожденных" копий make и не осталось свободных слотов заданий для выполнения "реальной" работы!)
* Если ваша операционная система не поддерживает нужный механизм межпрограммного взаимодействия, то, в переменную **MAKEFLAGS** всегда записывается **`-j 1'.** Это делается для того, чтобы случайно не превысить максимальное число одновременной запускаемых заданий из-за возможного рекурсивного запуска make. Опция `-j' без числовых аргументов передается "вниз" без изменений (поскольку она означает запуск максимального возможного числа заданий).
* Если вы не хотите передавать "вниз" другие опции, вы должны соответствующим образом изменить значение **MAKEFLAGS,** например:

subsystem:

cd subdir && $(MAKE) MAKEFLAGS=

* На самом деле, определения переменных, заданные в командной строке, помещаются в переменную **MAKEOVERRIDES**, а **MAKEFLAGS** содержит ссылку на эту переменную. Если вы хотите передать опции в make "нижнего уровня", но не хотите передавать определения переменных, заданные в командной строке, вы можете записать в переменную **MAKEOVERRIDES** пустое значение, например: **MAKEOVERRIDES =**
* Обычно, в этом нет особого смысла, однако, некоторые системы имеют небольшой и фиксированный лимит размера операционной среды, который легко может быть превышен при записи в переменную MAKEFLAGS такого большого количества информации. Эта проблема может проявляться в виде сообщения об ошибке `Arg list too long' (список аргументов слишком велик). (Для строгого соответствия стандарту POSIX.2, изменение MAKEOVERRIDES не влияет на MAKEFLAGS при наличии в make-файле специальной цели `.POSIX'. Для вас, скорее всего, это и неважно.)
* По "историческим" соображениям (для обеспечения совместимости) существует похожая переменная **MFLAGS**. Она содержит такое же значение, как и **MAKEFLAGS**, но в это значение не попадают определения переменных, заданных в командной строке, и, если это значение не пусто, оно всегда начинается с дефиса (значение переменной **MAKEFLAGS** начинается с дефиса только в том случае, если первая опция не имеет однобуквенного варианта названия, например `--warn-undefined-variables'). Традиционно, MFLAGS используется исключительно для рекурсивного вызова make, наподобие:

subsystem:

cd subdir && $(MAKE) $(MFLAGS)

* Переменная MAKEFLAGS делает подобную технику ненужной. Используйте эту методику в том случае, если вы хотите сделать ваш make-файл "совместимым" со старыми вариантами make; она будет нормально работать и с современными версиями make.
* Переменная MAKEFLAGS может оказаться полезной и в том случае, если вы хотите, чтобы некоторые опции, такие как `-k', использовались при каждом запуске make. Поместите нужное значение в переменную среды MAKEFLAGS. Вы также может установить значение MAKEFLAGS в make-файле, задав, таким образом, опции, которые должны использоваться для этого make-файла. (Обратите внимание, что вы не можете подобным образом использовать переменную MFLAGS. Значение этой переменной устанавливается только для совместимости; самостоятельное присваивание этой переменной другого значения никак не будет интерпретироваться make.)
* При интерпретации значения MAKEFLAGS (полученного как из операционной среды, так и из make-файла), значение этой переменной сначала предваряется дефисом (если оно еще не начинается с дефиса). Далее, это значение рассматривается как разделенные пробелами слова, являющиеся именами опций. Эти опции интерпретируются так, как если бы они были заданы в командной строке (за исключением того, что опции `-C', `-f', `-h', `-o', `-W' и их версии с длинными именами, игнорируются, а неверные опции не вызывают ошибки).
* Используя MAKEFLAGS как переменную среды будьте внимательны и не помещайте в нее никаких опций, которые оказывают "глобальное влияние" на работу make. Так, например, помещение опций `-t', `-n' или `-q' в переменные среды будет иметь "катастрофические" последствия и приведет к "удивительным" и неприятным эффектам.

***Опция `--print-directory'***

* При многократном рекурсивном вызове make, могут оказаться полезными опции **`-w' и `--print-directory'**, заставляющие make печатать имя текущего каталога, когда утилита начинает и заканчивает работу в нем.
* Например, при запуске **`make -w'** в директории **`/u/gnu/make'**, будет выведена следующая строка перед тем, как make начнет что-либо делать:

make: Entering directory `/u/gnu/make'.

* а строка ниже будет выведена перед тем, как работа будет закончена.

make: Leaving directory `/u/gnu/make'.

* Можно не указывать эти опции самостоятельно, поскольку печать каталогов автоматически включаются при наличии опции **`-C'**, а также в "порожденных" копиях make. Печать каталогов не будет включаться автоматически при наличии опции **`-s'** (подавляющей вывод информации) или опции **`--no-print-directory'** (явно запрещающей печать каталогов).

***Именованные командные последовательности (canned command sequences)***

* Когда одна и та же последовательность команд может быть использована для обновления разных целей, с помощью директивы **define** можно определить ее как **именованную командную последовательность** и, далее, использовать ее во всех правилах с такими целями. **Именованная командная последовательность** на самом деле является переменной, поэтому ее имя не должно конфликтовать с именами других переменных.

define run-yacc

yacc $(firstword $^)

mv y.tab.c $@

endef

* Здесь, run-yacc является именем определяемой переменной; endef обозначает конец определения; остальные строки являются командами. Ссылки на переменные и функции внутри директивы define не "раскрываются"; символы `$', скобки, имена переменных и прочее, становятся частью значения определяемой вами переменной.
* Первая команда запустит программу Yacc для первого пререквизита правила, в котором использована данная командная последовательность. Выходная информация программы Yacc всегда будет помещаться в файл `y.tab.c'. Вторая команда даст выходному файлу имя целевого файла правила.
* Переменные, определенные с помощью define являются рекурсивно вычисляемыми, поэтому все ссылки на переменные, находящиеся внутри директивы define, будут при этом вычислены:

foo.c : foo.y

$(run-yacc)

* при вычислении значения переменной run-yacc, вместо `$^' будет подставлено имя `foo.y', и имя `foo.c' вместо `$@'.
* На практике, в данном конкретном правиле нет необходимости, поскольку make имеет соответствующие неявные правила, имеющие аналогичный эффект.
* При выполнении команды, каждая строка именованной командной последовательности рассматривается так, как если бы она являлась отдельной строкой правила и ей предшествовал бы символ табуляции. Так же, каждая строка будет выполняться своей отдельной копией интерпретатора командной строки. В начале каждой командной строки именованной последовательности могут использоваться специальные символы `@', `-', и `+':

define frobnicate

@echo "frobnicating target $@"

frob-step-1 $< -o $@-step-1

frob-step-2 $@-step-1 -o $@

endef

* make не будет отображать первую командную строку, однако напечает следующие две строки с командами.
* В то же время, специальный символ в начале строки, ссылающейся на именованную командную последовательность, будет применен к каждой строке этой последовательности. Так, при обработке правила:

frob.out: frob.in

@$(frobnicate)

* ни одна команда отображена не будет.

***Пустые команды (empty commands)***

* Иногда, возникает потребность задать команду, которая, на самом деле, не выполняет никаких действий. Такая команда задается с помощью командной строки, не содержащей ничего, кроме пробела.

target: ;

* определяется пустую команду для цели `target'. Можно также использовать отдельную командную строку, начинающуюся с символа табуляции, однако это может вызвать путаницу, поскольку такая строка выглядит как пустая.
* Правило с пустой командой может предовратить применение для указанной цели команд, определенных в неявных правилах (или правилах со специальной целью **.DEFAULT**.
* Вы можете попытаться использовать пустые команды для тех целей, которые на самом деле не являются файлами, а служат лишь для того, чтобы были обновлены все их пререквизиты. Однако, это не слишком хорошая идея, поскольку при наличии реального файла, имя которого совпадает с именем цели, пререквизиты могут и не быть обновлены в нужный момент. В таких ситуациях лучше пользоваться абстрактными целями.

***Использование переменных (variables)***

* **Переменная (variable)** представляет собой имя, определенное в make-файле для представления строки текста, называемой **значением** переменной. В некоторых других версиях make, переменные называются **макросами (macros).**
* Переменные и функции во всех частях make-файла вычисляются при его чтении. Исключение составляют команды, передаваемые интерпретатора командной строки, правые части определений (с помощью символа `=') переменных, а также определения переменных с помощью директивы define.
* Переменная может представлять собой что угодно, например, список файлов, набор передаваемых компилятору опций, имя запускаемой программы, список каталогов с исходными файлами, директорию для выходных файлов и так далее.
* Именем переменной может быть любая последовательность символов, не содержащая **`:', `#', `='** и начальных или конечных пробелов. Лучше избегать использования имен переменных, содержащих символы, отличные от букв, цифр и символа подчеркивания. Во-первых, такие имена в будущем могут получить какое-либо специальное значение, и, во-вторых, не все интерпретаторы командной строки смогут передать (через переменные среды) такие переменные "порожденным" копиям make.
* Имена переменных чувствительны к регистру. Таким образом, имена `foo', `FOO', и `Foo' будут ссылаться на разные переменные.
* Традиционно, имена переменных записывались с использованием букв верхнего регистра. Рекомендуем пользоваться нижним регистром для всех переменных, используемых для "внутренних нужд" make-файла. Верхний же регистр оставить для переменных, влияющих на работу неявных правил или содержащих параметры, которые пользователь может переопределять.
* Некоторые переменные имеют имена, состоящие лишь из одного или нескольких символов пунктуации - автоматические переменные; они используются специальным образом.

***Обращение к переменным***

* Для подстановки значения переменной, напишите знак доллара, за которым следует имя переменной, заключенное в круглые или фигурные скобки: обе записи **`$(foo)'** и **`${foo}'** представляют собой ссылку на переменную **foo**. Из-за подобного специального значения символа **`$'**, для получения знака доллара в имени файла или команде нужно использовать запись **`$$'**.

***Две разновидности (flavors) переменных***

* **Рекурсивно вычисляемые (recursively expanded) переменные** - определяются с помощью **`='** или директивы **define**. Значение этой переменной запоминается точно в том виде, как вы его указали; если оно содержит ссылки на другие переменные, то эти ссылки будут вычислены (заменены своими текстовыми значениями) только в момент вычисления значения самой переменной (когда будет вычисляться какая-то другая строка, где использована эта переменная). Этот процесс называется **рекурсивным вычислением (recursive expansion).**
* Эта разновидность переменных - единственная, поддерживаемая другими версиями make. Она имеет свои достоинства и недостатки.

CFLAGS = $(include\_dirs) -O

include\_dirs = -Ifoo -Ibar

* ссылки на переменную `CFLAGS' будут "раскрываться" в текст `-Ifoo -Ibar -O'. С другой стороны, серьезный недостаток заключается в том, что вы не можете ничего "добавить" к переменной, наподобие: **CFLAGS = $(CFLAGS) -O**
* поскольку это вызовет бесконечный цикл при попытке вычислить ее значение. (make распознает ситуацию зацикливания и сообщает об ошибке.)
* Другой недостаток - все функции, на которые они ссылаются будут вычисляться заново при каждой "подстановке" этой переменной. Работа make при этом замедляется. Также результат выполнения функций wildcard и shell становится труднопредсказуемым, поскольку сложно в точности сказать, когда и сколько раз эти функции будут выполнены.
* Подобных недостатков лишены **упрощенно вычисляемые переменные.**
* **Упрощенно вычисляемые (simply expanded) переменные** определяются с помощью **`:='**. Значение такой переменной вычисляется (с расширением всех ссылок на другие переменные и вычислением функций) только в момент присваивания ей нового значения. После определения переменной, ее значение представляет собой обычный текст, уже не содержащий ссылок на другие переменные. Таким образом,

x := foo

y := $(x) bar

x := later

* эквивалентно

y := foo bar

x := later

* При ссылке на упрощенно вычисляемую переменную делается простая подстановка ее значения (без каких-либо дополнительных вычислений).
* Этот пример также демонстрирует использование переменной MAKELEVEL, которая изменяется во время переходов "с уровня на уровень" при рекурсивном использовании make.

ifeq (0,${MAKELEVEL})

cur-dir := $(shell pwd)

whoami := $(shell whoami)

host-type := $(shell arch)

MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}

endif

* Преимущество использования **`:='** заключается в том, что типичная команда `спуска в подкаталог' будет выглядеть как:

${subdirs}:

${MAKE} cur-dir=${cur-dir}/$@ -C $@ all

* В общем случае, упрощенно вычисляемые переменные позволяют сделать процесс программирования сложных make-файлов более предсказуемым, поскольку они работают аналогично обычным переменным в большинстве языков программирования. Они позволяют переопределять переменные, используя их собственные значения (возможно, обработанные какими-либо функциями) и использовать функции подстановки гораздо более эффективным образом.
* Упрощенно вычисляемые переменные можно использовать для добавления начальных пробелов в значения переменных. Начальные пробелы удаляются из значения переменной, однако, что вы можете сохранить начальный пробел, "защитив" его с помощью ссылки на переменную, например:

nullstring :=

space := $(nullstring) # end of the line

* Здесь, значением переменной space будет в точности один пробел. Комментарий `# конец строки' включен только для большей ясности. Если вы помещаете пробел в конец значения переменной, хорошей идеей будет помещение соответствующего комментария в конец строки, поясняющего ваши намерения. И наоборот, если для вас нежелательно наличие пробелов в конце значения переменной, вы не должны помещать каких-либо комментариев в конец строки после пробелов:

dir := /foo/bar # directory to put the frobs in

* Здесь, значением переменной dir будет строка `/foo/bar ' (с четырьмя пробелами в конце), чего вы, скорее всего, совсем не хотели. (Представьте себе, что случится со строкой, наподобие `$(dir)/file' в этом случае!)
* GNU make поддерживает еще один оператор присваивания **`?='**. Он называется оператором условного присваивания, поскольку срабатывает лишь в том случае, когда переменная еще не была определена. Например, выражение:

FOO ?= bar эквивалентно:

ifeq ($(origin FOO), undefined)

FOO = bar

endif

* Обратите внимание на то, что переменная, содержащая "пустое" значение, все равно считается определенной, и оператор **`?='** не будет присваивать ей нового значения.

***Ссылка с заменой (substitution reference)***

* При использовании ссылки с заменой (substitution reference), вместо нее подставляется значение переменной. Имеет форму **`$(переменная:a=b)' (или `${переменная:a=b}')**. Это значит, что должно быть взято значение переменной, и каждая найденная в нем цепочка символов **a**, находящаяся в конце слова, должна быть заменена на цепочку символов **b**.
* "находящаяся в конце слова" - за последовательностью символов **a** должен следовать пробел, либо она должна находиться в конце строки (со значением переменной); только в этих случаях она будет заменена на последовательность **b**. Все прочие цепочки **a** (не удовлетворяющие указанным условиям) будут оставлены без изменений.

foo := a.o b.o c.o

bar := $(foo:.o=.c)

* переменная `bar' получит значение `a.c b.c c.c'.
* На самом деле, ссылка с заменой представляет собой "укороченый" вариант использования функции **patsubst**. Для поддержания совместимости с другими версиями make, мы поддерживаем оба этих механизма - функцию **patsubst** и **ссылки с заменой**.
* Другой вариант ссылки с заменой соизмерим по "мощности" с использование функции **patsubst**. Он имеет аналогичную форму **`$(переменная:a=b)'**, но теперь в строке **a** должен присутствовать одиночный символ `**%**'. Этот вариант ссылки с заменой эквивалентен **`$(patsubst a,b,$(переменная))'**.

foo := a.o b.o c.o

bar := $(foo:%.o=%.c)

* переменная `bar' получит значение `a.c b.c c.c'.

***Вычисляемые имена переменных (computed variable names)***

* Вычисляемые имена переменных - достаточно сложная концепция, полезная для программирования весьма "нетривиальных" make-файлов. В большинстве случаев, у вас не возникнет потребности в изучении этого механизма - достаточно лишь знать, что знак доллара внутри имени переменной может привести к весьма "странным" результатам. Однако, если вы хотите досконально изучить работу make или же действительно заинтересованы в изучении этого механизма, то читайте дальше.
* Внутри имени переменной может находится ссылка на другую переменную. Это называется **вычисляемым именем переменной (computed variable name)** или **вложенной ссылкой (nested variable reference)**.
* Число уровней вложенности может быть неограниченным.

x = y

y = z

z = u

a := $($($(x)))

* Здесь самая внутренняя ссылка `$(x)' заменяется на `y', так что строка `$($(x))' расширяется в `$(y)', которая, в свою очередь, расширяется в `z'; теперь мы получаем ссылку `$(z)', которая заменяется на `u'.
* Ссылки на вычисляемые имена переменных, в свою очередь, сами могут находится внутри имен переменных.

x = $(y)

y = z

z = Hello

a := $($(x))

* переменная a получит значение `Hello': строка `$($(x))' преобразуется в строку `$($(y))', которая преобразуется в строку `$(z)' которая, в свою очередь, имеет значение `Hello'.
* Вложенные ссылки могут содержать ссылки с заменой, а также вызовы функций. Вот пример, в котором используется функция subst:

x = variable1

variable2 := Hello

y = $(subst 1,2,$(x))

z = y

a := $($($(z)))

* Здесь, переменная a получает значение `Hello'. Выражение `$($($(z)))' преобразуется в строку `$($(y))', которая, затем, преобразуется в `$($(subst 1,2,$(x)))'. Далее, из переменной x берется ее текущее значение (`variable1'), которое, с помощью подстановки, заменяется на `variable2'. После этого, выражение выглядит как `$(variable2)', и его значением является строка `Hello'.
* Вычисляемое имя переменной совсем необязательно должно состоять из единственной ссылки на переменную. Оно вполне может включать в себя несколько ссылок на переменные, а также обычные текстовые строки. В следующем примере:

a\_dirs := dira dirb

1\_dirs := dir1 dir2

a\_files := filea fileb

1\_files := file1 file2

ifeq "$(use\_a)" "yes"

a1 := a

else

a1 := 1

endif

ifeq "$(use\_dirs)" "yes"

df := dirs

else

df := files

endif

dirs := $($(a1)\_$(df))

* переменная dirs получит значение одной из переменных (a\_dirs 1\_dirs, a\_files или 1\_files) в зависимости от значений переменных use\_a и use\_dirs.
* Вычисляемые имена переменных также могут использоваться в ссылках с заменой. В следующем примере:

a\_objects := a.o b.o c.o

1\_objects := 1.o 2.o 3.o

sources := $($(a1)\_objects:.o=.c)

* переменная sources получит значение `a.c b.c c.c' либо `1.c 2.c 3.c', в зависимости от значения переменной a1.
* Единственно ограничение на подобное использование вложенных ссылок состоит в том, что они не могут определять часть имени вызываемой функции. Оно вызвано тем, что распознание имен функций происходит до того, как производится вычисление вложенных ссылок.

ifdef do\_sort

func := sort

else

func := strip

endif

bar := a d b g q c

foo := $($(func) $(bar))

* в переменную `foo', вместо ожидаемого значения (результата применения функций sort или strip к аргументу `a d b g q c'), будет записана строка `sort a d b g q c' или `strip a d b g q c'. Возможно, в будущем подобное ограничение будет снято, если это окажется полезным.
* Вычисляемые имена переменных могут использоваться в левой части оператора присваивания и в директиве define, например:

dir = foo

$(dir)\_sources := $(wildcard $(dir)/\*.c)

define $(dir)\_print

lpr $($(dir)\_sources)

endef

* В данном примере определяются переменные `dir', `foo\_sources', и `foo\_print'.
* Обратите внимание, что вложенные ссылки на переменные и рекурсивно вычисляемые переменные - это разные концепции, хотя при написании "нетривиальных" make-файлов, они часто используются совместно.

***Как переменные получают свои значения***

* При запуске make, в командной строке вы можете задать значение переменной, которое "перекроет" значение, устанавливаемое в make-файле.
* Вы можете задать значение переменной внутри make-файла с помощью оператора присваивания или с помощью директивы define.
* Переменные среды (environment variables) автоматически становятся переменными make.
* Несколько переменных, называемых автоматическими, "автоматически" получают новое значение при обработке каждого правила. Каждая из таких переменных предназначена для какого-то конкретного применения.
* Некоторые переменные имеют "фиксированное" начальное значение (раздел Используемые в неявных правилах переменные).

***Установка значения переменной***

* objects = main.o foo.o bar.o utils.o
* определяется переменная с именем objects. Пробелы "вокруг" имени переменной и после `=', игнорируются.
* Переменные, определенные с помощью **`='**, являются **рекурсивно вычисляемыми (recursively expanded)** переменными. Переменные, определенные с помощью **`:='**, являются **упрощенно вычисляемыми (simply expanded)** переменными; их определения могут содержать ссылки на другие переменные, которые будут вычислены до того, как будет сделано определение (иными словами, до того, как в переменную будет записано новое значение).
* Имена переменных могут содержать ссылки на другие переменные и функции, которые будут вычислены (во время чтения make-файла) для получения действительного имени переменной.
* Длина строки не имеет ограничений (за исключением доступного программе объема памяти). Можно разбить на несколько отдельных строк с помощью символа '\', за которым следует символ перевода строки. Такое разбиение будет способствовать повышению удобочитаемости make-файла.
* Большинство переменных, значение которых вы нигде не устанавливали, рассматриваются как содержащие пустую строку. Некоторые переменные имеют заранее определенные непустые начальные значения (которые, разумеется, вы можете изменить обычным образом - Используемые в неявных правилах переменные). Некоторые специальные переменные автоматически получают новое значение при обработке каждого правила - автоматические переменные.
* Если вы хотите, чтобы переменная получила новое значение только в том случае, если ей еще не было присвоено какого-либо значения, вместо **`='** используйте оператор **`?='**. Следующие два фрагмента make-файла производят одинаковый эффект:

FOO ?= bar

и

ifeq ($(origin FOO), undefined)

FOO = bar

endif

***Добавление текста к переменной***

objects = main.o foo.o bar.o utils.o

objects += another.o

* К значению переменной objects добавляется текст `another.o'. Так, переменная objects получит значение `main.o foo.o bar.o utils.o another.o'.
* Примерный аналог этого фрагмента без использования **`+='** может выглядеть так:

objects = main.o foo.o bar.o utils.o

objects := $(objects) another.o

* Однако, аналогия здесь не полная, поскольку работа оператора **`+='** отличается некоторыми деталями, которые проявляются при работе с более сложными значениями переменных.
* Если рассматриваемая переменная до сих пор не была определена, оператор **`+='** ведет себя как обычный **`='**, определяя **рекурсивно вычисляемую переменную**.
* Если же переменная была ранее определена, поведение оператора **`+='** будет зависеть от ее **разновидности.**
* Когда вы что-либо добавляете с помощью **`+='** к значению уже определенной переменной, make, по существу, действует так, как если бы этот дополнительный текст был включен в первоначальное определение переменной. Предположим, что переменная была ранее определена с помощью **`:='** и является, вследствие этого, **упрощенно вычисляемой переменной**. Тогда, при добавлении к ней нового текста, оператор **`+='** предварительно его "расширит" (аналогично тому, как это делает оператор **`:='**).

variable := value

variable += more

* эквивалентен следующему:

variable := value

variable := $(variable) more

* С другой стороны, при использовании оператора **`+='** с **рекурсивно вычисляемой** переменной, make работает немного по-другому. Когда вы определяете подобную переменную, make не сразу вычисляет ссылки на функции и другие переменные, присутствующие в устанавливаемом для нее значении. Вместо этого, make запоминает указанный вами текст в "оригинальном" виде, так что все присутствующие в нем ссылки на переменные и функции остаются и могут быть вычислены позднее, когда переменная будет "раскрываться". Таким образом, при использовании оператора **`+='** с **рекурсивно вычисляемой** переменной, указанный вами текст добавляется к "нераскрытому" значению, которое хранится внутри переменной.

variable = value

variable += more

* примерно эквивалентен:

temp = value

variable = $(temp) more

* (разумеется, на самом деле, никакой переменной temp не создается). Важность этого становится понятной при рассмотрении более сложных случаев, когда "старое" значение переменной содержит ссылки на другие переменные. Рассмотрим типичный пример:

CFLAGS = $(includes) -O

...

CFLAGS += -pg # enable profiling

* В первой строке этого примера определяется переменная **CFLAGS**, которая содержит ссылку на другую переменную с именем **includes**. (переменная **CFLAGS** используется в неявных правилах для компиляции программ на языке Си) Использование оператора **`='** определяет переменную **CFLAGS** как **рекурсивно вычисляемую**, и, значит, выражение **`$(includes) -O'** не будет вычисляться в момент определения переменной. Таким образом, переменная **includes** совсем необязана к этому времени быть определена. Достаточно, чтобы она была определена к тому моменту, когда будет вычисляться значение переменной **CFLAGS**. Если мы попробуем обойтись без оператора **`+='**, придется сделать что-нибудь, наподобие:

CFLAGS := $(CFLAGS) -pg # enable profiling

* К сожалению, этот фрагмент работает не совсем так, как хотелось бы. Из-за использования оператора **`:='** переменная **CFLAGS** становится **упрощенно вычисляемой**; это означает, что make "раскроет" выражение **`$(CFLAGS) -pg'** перед тем, как присвоить его переменной. Если переменная **includes** еще не была определена, в качестве результата "раскрытия" мы получим строку **` -O -pg'**, и последующее определение **includes** уже не сможет повлиять на этот результат. Напротив, при использовании **`+='** в переменную **CFLAGS** запишется нераскрытое значение **`$(includes) -O -pg'**. Таким образом, мы сохраняем ссылку на переменную **includes**. Если переменная **includes** будет определена где-нибудь позднее, то ссылка на **`$(CFLAGS)'** будет использовать ее значение.

***Директива override***

* Если переменная была установлена при помощи командной строки, то "обычное" присваивание ей нового значения внутри make-файла игнорируется. Если вы все-таки хотите присвоить подобной переменной новое значение, нужно использовать директиву **override**, выглядящую следующим образом:

override переменная = значение

* или

override переменная := значение

* При добавлении текста к переменной, определенной через командную строку, используйте конструкцию: **override переменная += добавляемый-текст**
* Директива override была придумана, чтобы дать возможность изменять или дополнять передаваемые пользователем в командной строке значения.
* Предположим, вы хотите, чтобы компилятор Си всегда запускался с опцией `-g' и, в то же время, хотели бы дать пользователю возможность самостоятельно указать необходимые опции компиляции: **override CFLAGS += -g**

***Многострочные переменные***

* Другой способ установки значения переменной - использование директивы define. Эта директива имеет несколько необычный синтаксис, позволяющий включать в ее значение символы перевода строки. С ее помощью удобно определять **именованные командные последовательности**.
* На первой строке находится только название директивы **define**, за которым следует **имя переменной**. Значение переменной указывается в следующих строках. Меткой конца значения переменной служит строка, содержащая единственное слово **endef**. За исключением синтаксических различий, директива **define** работает аналогично оператору **`='**, создавая **рекурсивно вычисляемую** переменную. Имя этой переменной может содержать функции и ссылки на другие переменные, которые будут "вычислены" в момент чтения директивы define для нахождения действительного имени определяемой переменной.

define two-lines

echo foo

echo $(bar)

endef

* При использовании обычного оператора присваивания, значение переменной не может содержать символов перевода строки. При использовании же директивы define, символы перевода строки (за исключением символа, находящегося перед строкой с endef) становятся частью значения переменной.
* Предыдущий пример функционально подобен:

two-lines = echo foo; echo $(bar)

* поскольку две команды, разделенные точкой с запятой, работают во многом также, как и две отдельные команды. Заметьте, однако, что для команд, расположенных в двух отдельных строках, make будет вызывать командный интерпретатор дважды, запуская каждую команду в своей отдельной копии интерпретатора.
* Для переменных, определенных при помощи **define**, также может использоваться директива **override**. Как обычно, при ее использовании, определение переменной внутри make-файла будет иметь "приоритет" перед определением этой же переменной из командной строки:

override define two-lines

foo

$(bar)

endef

***Переменные из операционного окружения (environment)***

* Переменные в make могут "приходить" из программного окружения (environment), в котором make была запущена. Каждая переменная среды (environment variable), видимая для make, преобразуется в соответствующую переменную make с таким же именем и значением. Однако, явное определение такой же переменной внутри make-файла или через командную строку, "перекроет" значение, полученное из операционной среды. (При наличии опции **`-e',** значения из переменных среды будут иметь "приоритет" перед значениями, определенными в make-файле. Но мы не рекомендуем использовать такую практику.)
* Таким образом, установив, например, переменную среды CFLAGS, вы заставите большинство make-файлов запускать компилятор Си с указанными вами опциями. Такая методика относительно безопасна для переменных со стандартными или общепринятыми значениями, поскольку вряд ли make-файлы будут использовать такие переменные для каких-либо других целей. (Разумеется, стопроцентной гарантии надежности здесь дать нельзя; например, некоторые make-файлы самостоятельно устанавливают переменную CFLAGS и, таким образом, не зависят от значения соответствующей переменной среды.)
* При рекурсивном вызове make, переменные, определенные на "верхних уровнях" могут быть переданы на "нижние уровни" через операционное окружение. По умолчанию, через операционную среду будут передаваться только переменные, которые были "первоначально" в ней определены, а также переменные, определенные с помощью командной строки. Для передачи через операционную среду любых других переменных, следует использовать директиву **export**.
* Использовать переменные среды для других целей мы не рекомендуем. Плохо, если поведение make-файлов будет зависеть от значения (неподконтрольных им) переменных среды; это может привести к тому, что один и тот же make-файл у разных пользователей будет работать по-разному, выдавая разные результаты. Это противоречило бы самой идее make-файлов.

***Целе-зависимые (target-specific) значения переменных***

* Значения переменных в make обычно являются глобальными; они одинаковы - в каком бы месте make-файла они ни вычислялись. Одно из исключений - автоматические переменные.
* Другое исключение - это **целе-зависимые значения переменных (target-specific variable values)**. С их помощью вы можете задать для одной и той же переменной разные значения в зависимости от того, какую цель в данный момент обновляет make. Так же, как и автоматические переменные, эти значения доступны только в контексте выполняемых для обновления цели команд (а также других целе-зависимых операторов присваивания).
* Целе-зависимые значения переменных устанавливаются с помощью конструкции вида: **цель ... : присваивание-переменной** - или - **цель ... : override присваивание-переменной.**
* При указании сразу нескольких целей, для каждой из них создается свое отдельное целе-зависимое значение.
* Присваивание-переменной может быть любой допустимой формой оператора присваивания; рекурсивной (`='), статической (`:='), дополняющей (`+='), или условной (`?=').
* Все переменные, участвующие в присваивании-переменной, вычисляются "в контексте" указанных целей: таким образом, все ранее определенные для этих целей целе-зависимые переменные будут здесь доступны. Обратите внимание, что целе-зависимые значения переменных не обязательно должны принадлежать к одной разновидности (рекурсивно вычисляемые или упрощенно вычисляемые).
* Целе-зависимые переменные имеют такой же приоритет как и любые другие переменные make-файла; переменные, определенные через командную строку (или "взятые" из переменных среды при наличии опции **`-e'**) будут иметь перед ними "приоритет". По-прежнему, использование директивы override позволит целе-зависимой переменной избежать "перекрытия".
* Одна из особенностей целе-зависимых переменных заключается - ее значение также будет "видно" для всех пререквизитов данной цели (конечно, если для пререквизитов не определены свои собственные целе-зависимые переменные).

prog : CFLAGS = -g

prog : prog.o foo.o bar.o

* переменная CFLAGS будет иметь значение `-g' во всех командах, выполняемых для цели `prog', а также (обратите внимание!) для целей `prog.o', `foo.o', `bar.o', и всех их пререквизитов.

***Шаблонно-зависимые (pattern-specific) значения переменных***

* Эти переменные считается определенными для всех целей, подходящих под указанный шаблон.
* Шаблонно-зависимые переменные устанавливаются с помощью конструкции вида: шаблон ... : присваивание-переменной - или - **шаблон ... : override присваивание-переменной**.
* где шаблон представляет собой шаблон с символом **'%'**. Подобно случаю задания целе-зависимых переменных, при указании сразу нескольких шаблонов, для каждого из них создается отдельный экземпляр шаблонного-зависимого значения. Присваивание-переменной может быть любой допустимой формой оператора присваивания. Как обычно, переменные, определенные через командную строку, будут иметь "приоритет" если только не использовать директиву override.

%.o : CFLAGS = -O

* переменной CFLAGS будет присвоено значение `-O' при обработке всех целей, удовлетворяющих шаблону %.o.

***Условные части (conditional parts) make-файла***

* **Условная конструкция (conditional)** заставляет make обрабатывать или игнорировать часть make-файла в зависимости от значения некоторых переменных. В качестве условия может использоваться сравнение двух переменных или сравнение переменной с константной строкой.

***Пример условной конструкции***

* Условная конструкция инструктирует make использовать разные наборы библиотек в зависимости от того, имеет ли переменная CC значение `gcc' или нет. Условная конструкция работает, управляя тем, какая из двух командных строк будет использоваться в качестве команды правила. В результате, при запуске make с параметром `CC=gcc' произойдет не только изменение используемого компилятора, но и изменение набора библиотек, с которыми будет компоноваться собираемая программа.

libs\_for\_gcc = -lgnu

normal\_libs =

foo: $(objects)

ifeq ($(CC),gcc)

$(CC) -o foo $(objects) $(libs\_for\_gcc)

else

$(CC) -o foo $(objects) $(normal\_libs)

endif

* В этой условной конструкции используется три директивы: ifeq, else и endif.
* Директива ifeq определяет начало условной конструкции и указывает само условие. Она имеет два параметра, разделенных запятой и заключенных в скобки. Эти параметры вычисляются и, затем, сравниваются. Если два параметра совпадают, строки, следующие за ifeq, обрабатываются; в противном случае они игнорируются.
* При использовании директивы else, следующие за ней строки должны быть обработаны, если условие (из директивы ifeq) не выполняется. Наличие директивы else в условной конструкции не является обязательным.
* Директива endif завершает условную конструкцию. Следующие за этой директивой строки, относятся уже к "безусловной" части make-файла. Наличие директивы endif является обязательным.
* Как видно из предыдущего примера, условные конструкции работают на "текстовом" уровне: отдельные строки рассматриваются как часть make-файла либо игнорируются, в зависимости от проверяемого условия. Поэтому, более крупные синтаксические элементы (например, правила) могут пересекать "границы" условной конструкции.
* Аналогичного результата можно добиться, заключив оператор присваивания переменной нужного значения в условную конструкцию, и, затем, использовать эту переменную "безусловно":

libs\_for\_gcc = -lgnu

normal\_libs =

ifeq ($(CC),gcc)

libs=$(libs\_for\_gcc)

else

libs=$(normal\_libs)

endif

foo: $(objects)

$(CC) -o foo $(objects) $(libs)

***Синтаксис условных конструкций***

* Синтаксис "сложной" условной конструкции выглядит так:

*условная-директива*

*фрагмент-для-выполненного-условия*

else

*фрагмент-для-невыполненного-условия*

endif

* **Условная-директива** имеет одинаковый синтаксис как в простой, так и в сложной условной конструкции. Имеется четыре разных директивы, проверяющих разные условия (такие же для **ifneq**). Вот они:

ifeq (параметр1, параметр2)

ifeq 'параметр1' 'параметр2'

ifeq "параметр1" "параметр2"

ifeq "параметр1" 'параметр2'

ifeq 'параметр1' "параметр2"

* Значения параметров **параметр1 и параметр2** вычисляются и сравниваются между собой. При их совпадении используется фрагмент **фрагмент-для-выполненного-условия**; иначе используется фрагмент **фрагмент-для-невыполненного-условия** (если он имеется). Часто возникает необходимость проверить - содержит ли переменная какое-нибудь "непустое" значение. Трудность состоит в том, что после разного рода преобразований и вычислений функций, значение, выглядящее как "пустое", будет, на самом деле, состоять из пробелов и, таким образом, не будет считаться "пустым".
* Для того, чтобы избежать интерпретации пробелов как непустых значений, можно воспользоваться функцией **strip**:

*ifeq ($(strip $(foo)),)*

*фрагмент-для-случая-пустого-значения*

*endif*

* фрагмент фрагмент-для-случая-пустого-значения будет использован даже в том случае, если значение переменной $(foo) будет состоять из пробелов.
* Переменные, которые еще не определены, рассматриваются как имеющие пустые значения. Обратите внимание, что директива **ifdef** просто проверяет - имеет ли переменная непустое значение. Она никогда не пытается вычислить это значение и проверить - не пустое ли оно. Как следствие, проверка с помощью **ifdef** (есть также **ifndef**) будет возвращать "истину" для всех определений, кроме определений, подобных:

**foo =**.

* Для проверки "вычисленного" значения переменной, используйте конструкцию ifeq ($(foo),). В следующем примере:

bar =

foo = $(bar)

ifdef foo

frobozz = yes

else

frobozz = no

endif

* переменная `frobozz' получит значение `yes', а в примере:

foo =

ifdef foo

frobozz = yes

else

frobozz = no

endif

* переменная `frobozz' будет установлена в `no'.
* В начале строки с условной директивой могут находится пробелы (которые игнорируются) но не символ табуляции. (При наличии символа табуляции такая строка рассматривалась бы как команда правила.) За этим маленьким исключением, пробелы и символы табуляции могут свободно использоваться в любом месте строки с условной директивой (только, разумеется, не внутри имени самой директивы и не внутри аргументов). В конце строки может располагаться комментарий, начало которого обозначается символом `#'.
* Две другие директивы, используемые в условных конструкциях - это директивы **else и endif**. Каждая из этих директив записывается в одно слово и не имеет параметров. В начале строк с этими директивами, могут находится дополнительные пробелы (которые будут игнорированы), а в конце строк - пробелы и символы табуляции (которые также будут игнорированы). В конце таких строк может располагаться комментарий, начало которого обозначается символом `#'.
* Условные конструкции влияют на то, какие строки make-файла в действительности будет использовать make. При выполнении указаного условия, make будет рассматривать строки фрагмент-для-выполненного-условия как часть make-файла; при невыполнении условия, эти строки будут игнорироваться. Вследствие этого, синтаксические единицы make-файла (такие как правила) вполне могут пересекать границы условной конструкции.
* Условия, указанные в условных конструкциях, вычисляются в момент чтения make-файла. Как следствие, в качестве условий не может использоваться проверка **автоматических переменных**, поскольку они являются неопределенными до момента запуска команд правила.
* Во избежании неприятных конфузов, не разрешается "начинать" условную конструкцию в одном make-файле и "заканчивать" ее в другом. Однако, внутри условной конструкции вы можете использовать директиву include (при условии, что включаемый make-файл не будет пытаться "завершить" эту условную конструкцию).

***Проверка опций запуска make в условных конструкциях***

* Вы можете написать условную конструкцию, проверяющую наличие определенных опций (например, `-t'), указанных при запуске make. Это можно сделать, используя переменную **MAKEFLAGS** совместно с функцией **findstring**. Необходимость в этом может возникнуть, например, в ситуации, когда одного лишь использования команды **touch** недостаточно для обновления файла.
* Функция **findstring** определяет, входит ли одна строка в другую в качестве подстроки. Если, например, вы хотите проверить наличие опции **`-t'**, используйте **`t'** как первую строку (первый параметр функции), а переменную **MAKEFLAGS** - как другую строку (второй параметр функцции).
* В следующем примере, в качестве завершающего шага пометки архивного файла как "обновленного", используется команда **`ranlib -t':**

archive.a: ...

ifneq (,$(findstring t,$(MAKEFLAGS)))

+touch archive.a

+ranlib -t archive.a

else

ranlib archive.a

endif

* Префикс `+' помечает командные строки как "рекурсивные". Эти команды будут выполняться даже при наличии опции `-t'.

***Функции преобразования текста***

* Вызов функции внешне напоминает ссылку на переменную. Он выглядит так: $(функция параметры) или так ${функция параметры}
* В make имеется некоторое количество "встроенных" функций. С помощью встроенной функции call вы можете определить свою собственную функцию.
* Параметры являются параметрами функции. От имени функции они отделяются одним или несколькими пробелами, или символами табуляции. При наличии нескольких параметров, они отделяются друг от друга запятыми. Такие пробелы и запятые не рассматриваются как часть значения параметра.
* Разделители, которые вы использовали для обозначения вызова функции (круглые или фигурные скобки), могут появляться в аргументах только "попарно"; другие символы разделителей могут появляться и поодиночке. Если аргументы сами, в свою очередь, содержат ссылки на другие функции или переменные, для всех ссылок рекомендуется использовать один и тот же вид разделителей; то есть, например, писать `$(subst a,b,$(x))', а не `$(subst a,b,${x})'. Такая запись является не только более ясной, но и более "простой" для make (только один вид разделителей используется при поиске конца ссылки).
* Перед тем, как аргумент будет передан для обработки в функцию, вычисляются все содержащиеся в нем ссылки на переменные и функции. Обработка аргументов производится в том порядке, как они перечислены.
* Запятые и непарные скобки не могут "явным" образом появляться в аргументах функции; нельзя также "явным" образом указать наличии ведущих пробелов в первом параметре функции. Однако, эти символы могут быть вставлены в аргумент с помощью ссылки на переменные. Это можно сделать, определив, например, переменные comma и space, значениями которых будут, соответственно, одиночные символы запятой и пробела.
* Далее, значения этих переменных могут быть подставлены в любое место аргументов, где требуется наличие соответствующего символа. Например:

comma:= ,

empty:=

space:= $(empty) $(empty)

foo:= a b c

bar:= $(subst $(space),$(comma),$(foo))

# bar is now `a,b,c'.

* Здесь, с помощью функции subst, каждый символ пробела, содержащийся в переменной foo, заменяется на символ запятой.

***Функции анализа и подстановки строк***

* **$(subst заменяемый\_фрагмент,замена,текст)** - производит текстовую замену в *тексте*: каждое вхождение подстроки *заменяемый\_фрагмент* заменяется на фрагмент *замена*. Результат подставляется в место вызова функции:

$(subst ee,EE,feet on the street)

результатом будет строка `fEEt on the strEEt'

* **$(patsubst шаблон,замена,текст)** - находит в тексте разделенные пробелом слова, удовлетворяющие *шаблону* и заменяет их на строку *замена*.

$(patsubst %.c,%.o,x.c.c bar.c)

результатом выражения будет строка `x.c.o bar.o'.

**Ссылка с заменой** является упрощенным способом получения эффекта, аналогичного использованию функции **patsubst**. Выражение: $(переменная:шаблон=замена) эквивалентно $(patsubst шаблон,замена,$(переменная))

Еще одна упрощенная форма записи имеется для распространенного способа использования функции **patsubst**: замены суффиксов в именах файлов.

objects = foo.o bar.o baz.o

$(objects:.o=.c) вместо "обобщенной" формы записи - $(patsubst %.o,%.c,$(objects))

* **$(strip строка)** - удаляет начальные и конечные пробелы из строки, а также заменяет все внутренние последовательности пробельных символов на одиночные пробелы. Так, результатом выражения `$(strip a b c )' будет строка `a b c'.

Функция **strip** весьма полезна в условных конструкциях. Например, при использовании директив **ifeq и ifneq** для сравнения с пустой строкой `', обычно желательно, чтобы строка, целиком состоящая из пробельных символов, рассматривалась как пустая. Так, например, следующий фрагмент make-файла не всегда будет работать желаемым образом:

.PHONY: all

ifneq "$(needs\_made)" ""

all: $(needs\_made)

else

all:;@echo 'Nothing to make!'

endif

Заменив в директиве ifneq ссылку на переменную `$(needs\_made)' вызовом функции `$(strip $(needs\_made))', мы получим более надежно работающую конструкцию.

* **$(findstring фрагмент,строка)** - производит поиск фрагмента в строке. В случае успеха (фрагмент найден) возвращает значение фрагмент; в противном случае, возвращается пустая строка. Эту функцию можно использовать в условных конструкциях для проверки наличия в рассматриваемой строке определенной подстроки.

$(findstring a,a b c)

$(findstring a,b c)

Результатами будут, соответственно, строки `a' и `' (пустая строка).

* **$(filter шаблон...,текст)** - удаляет из текста все разделенные пробелами слова, которые не удовлетворяют ни одному из указанных шаблонов и возвращает только слова, подходящие под шаблоны.

Функция filter может быть использована для отделения друг от друга строк (например, имен файлов) разных "типов":

sources := foo.c bar.c baz.s ugh.h

foo: $(sources)

cc $(filter %.c %.s,$(sources)) -o foo

объявляется, что цель `foo' зависит от файлов `foo.c', `bar.c', `baz.s' и `ugh.h', однако, при вызове компилятора, ему будут переданы только файлы `foo.c', `bar.c' и `baz.s'.

* **$(filter-out шаблон...,текст)** - удаляет из текста все разделенные пробелами слова, которые соответствуют какому-либо из перечисленных шаблонов, возвращая только слова, не соответствующие ни одному из шаблонов.

objects=main1.o foo.o main2.o bar.o

mains=main1.o main2.o

$(filter-out $(mains),$(objects))

возвратит список объектных файлов, не входящих в `mains'.

* **$(sort список) -** отсортировывает слова из списка в лексикографическом порядке, удаляя дубликаты (повторяющейся слова). Результатом является список слов, разделенных одиночными пробелами.

$(sort foo bar lose) - результатом выражения будет строка `bar foo lose'.

* Реалистичный пример использования функций subst и patsubst. Значение переменной VPATH представляет собой список имен каталогов, разделенных двоеточиями, например **`src:../headers'**. Сперва используем функцию **subst** для замены символов двоеточия на пробелы: $(subst :, ,$(VPATH))
* Полученный результат будет выглядеть как `**src ../headers'**. Далее, с помощью функции **patsubst**, преобразуем каждое из имен каталогов в соответствующую опцию `-I' компилятора.
* Полученное значение можно добавить к содержимому переменной CFLAGS, которая автоматически передается компилятору:

override CFLAGS += $(patsubst %,-I%,$(subst :, ,$(VPATH)))

* Как результат, к первоначальному значению переменной CFLAGS добавляется строка `-Isrc -I../headers'. Директива override была использована для того, чтобы изменить значение переменной CFLAGS даже в том случае, если она была задана с помощью командной строки.

***Функции для обработки имен файлов***

* **$(dir имена...) -** из каждого имени файла, перечисленного в именах, выделяет имя каталога, где этот файл расположен. Именем каталога считается часть имени до последнего встреченного символа `/' (включая и этот символ). Если имя файла не содержит символов `/', его именем каталога считается строка `./'.

$(dir src/foo.c hacks) - результатом будет строка `src/ ./'.

* **$(notdir имена...) -** из каждого имени файла, перечисленного в именах, удаляет имя каталога, где он находится. Имена файлов, не содержащие символов `/', остаются без изменений. В противном случае (при наличии символов `/'), из имени файла удаляется все, что расположено до последнего встреченного символа `/' (включая и сам этот символ). Это означает, что имя файла, оканчивающееся символом `/' преобразуется в пустую и строку, и, таким образом, количество имен файлов на выходе функции может не совпадать с количеством имен файлов, переданных ей на вход:

$(notdir src/foo.c hacks) - результатом будет строка `foo.c hacks'.

* **$(suffix имена...) -** из каждого имени файла, перечисленного в именах, выделяется его суффикс.

$(suffix src/foo.c src-1.0/bar.c hacks) - результатом будет строка `.c .c'

* **$(basename имена...) -** из каждого имени файла, перечисленного в именах, выделяет так называемое "базовое имя" (basename) - все то, что не относится к суффиксу.

$(basename src/foo.c src-1.0/bar hacks) - результатом будет строка `src/foo src-1.0/bar hacks'.

* **$(addsuffix суффикс,имена...) -** аргумент имена рассматривается как последовательность разделенных пробелами имен; аргумент суффикс рассматривается как строка. Результатом работы этой функции является список имен (разделенных одиночными символами пробелов), каждое из которых получено из соответствующего "исходного" имени, в конец которого добавлен суффикс суффикс.

$(addsuffix .c,foo bar) - результатом будет строка `foo.c bar.c'.

* **$(addprefix префикс,имена...) -** аргумент имена рассматривается как последовательность разделенных пробелами имен; аргумент префикс рассматривается как строка. Результатом этой функции является список имен (разделенных одиночными символами пробелов), каждое из которых получено из соответствующего "исходного" имени, в начало которого добавлен префикс префикс.

$(addprefix src/,foo bar) - результатом будет строка `src/foo src/bar'.

* **$(join список1,список2) -** "пословно" объединяет оба аргумента: первые два слова (по одному из каждого аргумента) объединяются в первое слово результата; вторые два слова (второе слово каждого из аргументов) объединяются во второе слово результата и т.д. Таким образом, n-ное слово результата строится из n-ного слова каждого из аргументов. Если один из аргументов содержит большее количество слов чем другой, "избыточные" слова копируются в результат без изменений.

Например, результатом выражения: **`$(join a b,.c .o)'** будет строка **`a.c b.o'.** "Оригинальные" пробелы между словами списка не сохраняются - они заменяются на одиночный символ пробела. Применив join к результатам работы функций dir и notdir, можно получить "исходный" список файлов.

* **$(word n,текст) -** возвращает n-ное слово текста. Допустимые значения n начинаются с 1. Если значение n превышает количество слов в тексте, результатом работы фунции будет пустая строка.

$(word 2, foo bar baz) - результатом будет строка `bar'.

* **$(wordlist s,e,текст) -** возвращает список слов текста, начиная со слова номер s и заканчивая словом номер e (включительно). Допустимые значения s и e начинаются с 1. Если s превышает количество слов в тексте, возращается пустая строка. Если e превышается количество слов в тексте, возвращаются все слова до конца текста. Если величина s превышает e, также возвращается пустая строка.

$(wordlist 2, 3, foo bar baz) - результатом будет строка `bar baz'.

* **$(words текст) -** возвращает число слов в тексте. Таким образом, последнее слово текста можно получить с помощью выражения **$(word $(words текст),текст)**.
* **$(firstword имена...) -** аргумент имена рассматривается как последовательность имен, разделенных пробелами. Результатом функции является первое имя из списка. Остальные имена игнорируются.

$(firstword foo bar) - результатом будет строка`foo'. Хотя выражение $(firstword текст) и эквивалентно $(word 1,текст), использование функции firstword может повысить удобочитаемость make-файла.

* **$(wildcard шаблон) -** аргумент шаблон является шаблоном имени файла и, обычно, содержит шаблонные символы (такие же как в шаблонах имен файлов интерпретатора командной строки). Результатом функции wildcard является список разделенных пробелами имен существующих в данный момент файлов, удовлетворяющих указанному шаблону.

***Функция foreach***

* При ее использовании, некоторый фрагмент текста используется многократно, каждый раз с "подстановкой" в него нового значения - напоминает команду for командного интерпретатора sh.
* $(foreach переменная,список,текст)
* Сначала, вычисляются значения первых двух аргументов - переменной и списка; последний аргумент, текст, пока не вычисляется. Далее, каждое слово из вычисленного значения аргумента список поочередно подставляется в переменную с (заранее вычисленным) именем переменная и производится "расширение" текста текст. Как правило, текст содержит ссылку на эту переменную, поэтому при каждой новой подстановке получаются разные результаты.
* Затем, полученные таким образом результаты "расширений" текста (их количество равно количеству разделенных пробелами слов в аргументе список) "соединяются" вместе (с вставкой пробела между ними). Полученная таким образом строка и является результатом работы функции foreach.
* В следующем примере, в переменную `files' заносится список всех файлов, находящихся в каталогах, которые перечислены в переменной `dirs':

dirs := a b c d

files := $(foreach dir,$(dirs),$(wildcard $(dir)/\*))

* Здесь, аргументом текст является выражение **`$(wildcard $(dir)/\*)'**. При первой итерации, переменная dir получает значение `a', что производит эффект, аналогичный `$(wildcard a/\*)'; вторая интерация даст результат, аналогичный `$(wildcard b/\*)'; и, наконец, третья итерация даст результат, как от выражения `$(wildcard c/\*)'.
* Таким образом, приведенный выше пример даст результат, аналогичный (за исключением установки переменной `dirs') выражению: **files := $(wildcard a/\* b/\* c/\* d/\*)**
* Когда выражение текст достаточно сложно, вы можете повысить удобочитаемость make-файла, поместив его в отдельную дополнительную переменную:

find\_files = $(wildcard $(dir)/\*)

dirs := a b c d

files := $(foreach dir,$(dirs),$(find\_files))

* Для подобной цели в приведенном выше примере использована переменная find\_files. Для ее определения как рекурсивно вычисляемой переменной, мы использовали обычный оператор `='. Вследствие этого, ее значение (содержащее ссылку на функцию wildcard), будет вычисляться многократно, под управлением функции foreach; с упрощенно вычисляемой переменной этого бы не произошло, поскольку функция wildcard была бы выполнена лишь однажды, во время определения переменной find\_files.
* Работа функции foreach не оказывает "необратимого" влияния на переменную; ее значение и "разновидность" после выполнения функции foreach остаются неизменными (такими же, как и до выполнения этой функции). Другие значения, которые берутся из списка, "действуют" только временно, на период работы функции foreach. Во время работы функции foreach, переменная считается упрощенно вычисляемой. Если до выполнения функции foreach, переменная не была определена, она остается неопределенной и после вызова этой функции.

***Функция if***

* Функция if обеспечивает поддержку для условного вычисления выражений (не путайте с поддерживаемыми GNU make условными конструкциями наподобие ifeq, которые "действуют" на уровне make-файла).
* При вызове функции if, ей передается два или три аргумента:

$(if условие,фрагмент-для-выполненного-условия[,фрагмент-для-невыполненного-условия])

* Сначала из первого аргумента, условия, удаляются начальные и конечные пробелы, затем он вычисляется. Если в результате получается любая непустая строка, то условие считается "истинным". При получении пустой строки, условие считается "ложным".
* Если условие выполняется, вычисляется второй аргумент, фрагмент-для-выполненного-условия, и полученный результат становится результатом вычисления всей функции if.
* Если условие не выполняется, вычисляется третий аргумент, фрагмент-для-невыполненного-условия, и полученный результат становится результатом вычисления всей функции if. При отсутствии третьего аргумента, результатом выполнения функции if становится пустая строка.
* Обратите внимание, что всегда вычисляется только один из фрагментов - либо фрагмент-для-выполненного-условия, либо фрагмент-для-невыполненного-условия. Поэтому, оба из них могут производить какие-либо "побочные" эффекты (например, вызывать функцию shell).

***Функция call***

* С ее помощью вы можете определять свои собственные функции с параметрами. Вы можете запомнить сложное выражение в качестве значения переменной, а затем использовать функцию call для его вычисления с разными параметрами.
* **$(call переменная,параметр,параметр,...)**
* При вычислении этой функции, make помещает каждый из параметров во временные переменные $(1), $(2) и т.д.. Переменная $(0) будет содержать имя переменной. На максимальное число параметров ограничения нет. Нет ограничения и на минимальное число параметров, однако, нет особого смысла в использовании call без параметров.
* Далее, значение переменной вычисляется "в контексте" этих временных переменных. Так, любая ссылка на $(1) в значении переменной будет ссылаться на первый параметр, переданный при вызове call.
* Обратите внимание, что переменная - это имя переменной, а не ссылка на эту переменную. Поэтому, как правило, вам не потребуется использовать `$' или скобки при описании этого аргумента. (Вы можете, однако, использовать внутри имени ссылку на другую переменную, если вы хотите, чтобы имя не было "константным".)
* Если переменная представляет собой имя "встроенной" функции, то вызывается именно она (даже если существует переменная с таким же именем).
* Перед тем, как присвоить значения временным переменным, функция call вычисляет значения всех параметров. Это означает, что значения переменной, содержащие ссылки на встроенные функции, имеющие специальные правила вычисления (наподобие foreach или if), могут работать не так, как вы ожидали.
* Пример, функция "переставляет" свои аргументы в обратном порядке:

reverse = $(2) $(1)

foo = $(call reverse,a,b)

* Переменная foo будет содержать `b a'.
* Определяется функция, которая производит поиск указанной программы в каталогах, перечисленных в PATH:

pathsearch = $(firstword $(wildcard $(addsufix /$(1),$(subst :, ,$(PATH)))))

LS := $(call pathsearch,ls)

* Переменная LS будет содержать /bin/ls или что-нибудь подобное.
* Функция call может быть "вложенной". Каждый рекурсивный вызов получит свои собственные локальные копии $(1) и прочих переменных, которые "замаскируют" своих "тезок" из call более "высокого" уровня.
* map = $(foreach a,$(2),$(call $(1),$(a)))
* Теперь, с помощью функции map, вы можете "за один шаг" вызывать функции, имеющие только один параметр (наподобие origin), сразу для нескольких значений. Так, в следующем примере:
* o = $(call map,origin,o map MAKE)
* переменная o будет содержать нечто вроде `file file default'.
* И последнее предупреждение: будьте осторожны при добавлении пробелов к аргументам функции call. Как и с другими функциями, любые пробелы, содержащиеся во втором и последующих аргументах, сохраняются; это может привести к весьма странным результатам. Надежнее всего, удалять все "дополнительные" пробелы, указывая параметры для функции call.

***Функция origin***

* Функция origin позволяет вам получить некоторую информацию о самой переменной. Точнее, она позволяет узнать, откуда "взялась" рассматриваемая переменная.
* **$(origin переменная)**
* Обратите внимание, что переменная является именем переменной, а не ссылкой на нее. Таким образом, вам, как правило, не придется использовать символ `$' или скобки при написании этого аргумента. (Однако, внутри имени переменной может находится ссылка на другую переменную, если вы хотите, чтобы имя переменной не было "фиксированным".)
* Результатом этой функции будет строка, указывающая на то, каким образом переменная переменная была определена:
* **`undefined'** - переменная не была определена.
* **`default'** - переменная была определена по умолчанию (как, например, переменная CC и ей подобные). Обратите внимание, что если вы переопределили переменную, имеющую значение по умолчанию, функция origin возвратит информацию о более позднем переопределении.
* **`environment' -** переменная была создана из соответствующей переменной среды и опция `-e' не была включена.
* **`environment override'** - переменная была создана из соответствующей переменной среды и опция `-e' была включена.
* **`file'** - переменная была определена внутри make-файла.
* **`command line' -** переменная была определена с помощью командной строки.
* **`override' -** переменная была определена в make-файле с с использованием директивы override.
* **`automatic' -** переменная является автоматической переменной, определяемой во время выполнения команд каждого правила.
* Подобная информация может быть полезна для того, чтобы определить, насколько вы можете "доверять" значению, содержащемуся в рассматриваемой переменной. Предположим, что у вас имеется make-файл `foo', который включает в себя другой make-файл `bar'. Вы хотите, чтобы при запуске команды `make -f bar', переменная bletch была определена в make-файле `bar' даже в том случае, если аналогичная переменная bletch будет содержаться в операционном окружении. Однако, если переменная bletch уже была определена в make-файле `foo' (до подключения make-файла `bar'), вы бы не хотели "переопределять" эту переменную в `bar'. Это можно было бы сделать, используя в файле `foo' директиву override: в этом случае определение переменной, данное в файле `foo' имело бы приоритет перед более поздним ее определением в файле `bar'. К сожалению, директива override также "перекрыла" бы любое определение этой переменное, заданное в командной строке. Решение может выглядеть следующим образом (фрагмент make-файла `bar'):

ifdef bletch

ifeq "$(origin bletch)" "environment"

bletch = barf, gag, etc.

endif

endif

* Здесь, переменная bletch будет переопределена, если она была определена из соответствующей переменной среды.
* Если вы хотите "перекрыть" определение bletch, пришедшее из программного окружения, даже при наличии опции `-e', то можно написать:

ifneq "$(findstring environment,$(origin bletch))" ""

bletch = barf, gag, etc.

endif

* Здесь, переопределение произойдет, если выражение `$(origin bletch)' вернет любую из строк - `environment' или `environment override'.

***Функция shell***

* В отличие от большинства других функций (кроме, пожалуй, функции wildcard), функция shell служит для "общения" make с внешним миром.
* Функция shell работает аналогично символу ``' в большинстве интерпретаторов командной строки: она производит подстановку результата выполнения команды. Это означает, что в качестве аргумента она принимает команду интерпретатора командной строки, а в качестве результата возвращает "выходные данные" этой команды.
* Единственным преобразованием полученного результата, которое выполняет make перед подстановкой его в окружающий текст, является преобразование символов перевода строки (или пар перевод-строки/возврат-каретки) в одиночные пробелы. Также производится удаление "конечных" (находящихся в конце данных) символов перевода строки (или пар перевод-строки/возврат-каретки).
* Команды, указанные в shell, запускаются в момент вычисления этой функции. Как правило, это происходит в момент чтения make-файла. Исключение составляет случай, когда эта функция shell используется в командах правила. В этом случае она будет вычисляться (будут выполняться указанные в ней команды) во время работы команд правила.
* contents := $(shell cat foo)
* В переменную contents записывается содержимое файла `foo' (видоизмененное таким образом, что все символы перевода строки заменены в нем на пробелы).
* files := $(shell echo \*.c)
* В переменную files записывается список файлов, полученных по маске `\*.c'. Скорее всего (если только вы не имеете какой-нибудь очень странный командный интерпретатор), результат будет аналогичен использованию выражения `$(wildcard \*.c)'.

***Функции управления сборкой***

* Эти функции управляют ходом сборки. В основном, они используются для выдачи некоторой информации пользователю make-файла или для завершения работы make при обнаружении каких-либо проблем в "окружающей среде".
* **$(error текст...)**
* Генерирует "фатальную" ошибку с сообщением текст. Обратите внимание, что ошибка генерируется в момент вычисления функции. Соответственно, если вы вызываете эту функцию внутри команд правила или в правой части оператора присваивания для рекурсивной переменной, ошибка будет генерироваться не сразу. Перед генерацией ошибки сообщение "расширяется".

ifdef ERROR1

$(error error is $(ERROR1))

endif

* во время чтения make-файла будет генерироваться фатальная ошибка если была определена переменная ERROR1. Здесь:

ERR = $(error found an error!)

.PHONY: err

err: ; $(ERR)

* фатальная ошибка будет генерироваться при обработке цели err.
* **$(warning текст...) -** работает подобно описанной выше функции error, но, в отличии от нее, не вызывает завершения работы make. Сообщение текст "расширяется" и выводится, после чего обработка make-файла продолжается. Возвращаемое значение этой функции - пустая строка.

***Запуск make***

* Make-файл, описывающий, каким образом следует перекомпилировать программу, может использоваться по-разному. В простейшем случае, он используется для перекомпиляции всех "устаревших" файлов программы. Обычно, make-файлы пишутся таким образом, чтобы при запуске без параметров, make выполняла именно это действие.
* Однако, у вас может возникнуть потребность выполнить какие-нибудь другие действия, например: обновить только некоторые из файлов, использовать другой компилятор или другие опции компиляции. Наконец, у вас может появиться желание просто узнать какие из файлов нуждаются в обновлении без того, чтобы в действительности их обновлять.
* Указывая дополнительные параметры при запуске make, вы сможете выполнить эти и многие другие действия.
* По окончанию работы make, код завершения программы представляет собой одно из трех возможных значений:
* **0** - код завершения равен нулю если работа make завершена успешно.
* **2** - код завершения равен двум если в ходе работы make возникли какие-то ошибки. Суть происшедших ошибок описана в выдаваемых make сообщениях.
* **1** - код завершения равен одному, если при запуске make была указана опция `-q' и make определила, что некоторые цели нуждаются в обновлении.

***Аргументы для задания главной цели (goal)***

* Главная цель (goal) - это цель которую стремится достичь make в результате своей работы. Прочие цели make-файла обновляются только в том случае, если они прямо или косвенно являются пререквизитами главной цели.
* По умолчанию, главной целью становится первая цель make-файла (кроме целей, чьи имена начинаются с точки). Поэтому, обычно, make-файлы пишутся таким образом, чтобы первая цель описывала процесс компиляции программы или набора программ, для сборки которых предназначается make-файл. Если первое правило make-файла описывает сразу несколько целей, только первая из целей (а не все описываемые цели) становится главной целью по умолчанию.
* Вы можете задать make главную цель (или несколько главных целей), указав ее (или их) имя в качестве аргументов. При задании сразу нескольких главных целей, make будет обрабатывать их поочередно, в том порядке, как они были перечислены.
* В качестве главной цели может быть указана любая цель make-файла (за исключением целей, чье имя начинается с символа `-' или содержит символ `=', поскольку такая цель будет "воспринята" как задание опции или определение переменной). Можно задать даже такую цель, которая не описана в make-файле; в этом случае make попытается ее достичь, используя имеющиеся у нее неявные правила.
* Список главных целей, указанных вами в командной строке, make записывает в специальную переменную **MAKECMDGOALS**. Если в командной строке не было задано ни одной цели, эта переменная остается пустой. Обратите внимание, что эта переменная должна использоваться только в особых случаях.
* Следующий пример демонстрирует "надлежащее" использование переменной **MAKECMDGOALS**. Она используется для того, чтобы избежать подключения файлов **`.d'**, когда в командной строке указывается цель clean, так что make не будет пытаться создать эти файлы только для того, чтобы тут же их удалить:

sources = foo.c bar.c

ifneq ($(MAKECMDGOALS),clean)

include $(sources:.c=.d)

endif

* Явное указание главной цели полезно, если вы хотите перекомпилировать только часть программы или только одну из нескольких программ. В таком случае, укажите в качестве главной цели все файлы, которые вы хотели бы обновить. Пусть, например, у вас имеется каталог, содержащий сразу несколько программ и make-файл, который начинается примерно так:

.PHONY: all

all: size nm ld ar as

* Если вы работаете над программой size, удобно запускать `make size' чтобы перекомпилировались только файлы, относящиеся к этой программы.
* Явное задание цели может применяться для построения тех файлов, которые при "обычной" работе не строятся. Это, например, может быть файл с отладочной информацией или специальная тестовая версия программы, для которых в make-файле имеется свое правило, но которые не являются пререквизитами главной цели, выбираемой по умолчанию.
* Явное указание главной цели может использоваться для запуска команд, ассоциированных с абстрактной или пустой целью. Например, во многих make-файлах есть абстрактная цель `clean', которая очищает каталог, удаляя все файлы, кроме файлов с исходными текстами. Естественно, это делается только по вашему требованию `make clean'.

***Список типичных абстрактных и пустых целей***

* **`all'** - построить все "высокоуровневые" цели make-файла.
* **`clean'** - удалить все файлы, которые, обычно, создаются в результате работы make.
* **`mostlyclean'** - работает аналогично `clean', но может воздержаться от удаления некоторых файлов, которые, как правило, не желательно перекомпилировать. Например, в make-файле, описывающем сборку компилятора GCC, цель `mostlyclean' не удаляет файл `libgcc.a', поскольку его перекомпиляция требуется редко и занимает много времени.
* **`distclean', `realclean', `clobber'** - любая из этих целей может быть определена для удаления большего числа файлов, чем это делает `clean'. Так, например, могли бы удаляться конфигурационные файлы или линки, которые создаются при подготовке к компиляции (не обязательно самим make-файлом).
* **`install'** - скопировать исполняемый файл в каталог, где обычно хранятся исполняемые файлы программ; скопировать дополнительные файлы, используемые исполняемым файлом, в нужные каталоги (где исполняемый файл ожидает их найти).
* **`print'** - печать листингов всех исходных файлов, которые были модифицированы.
* **`tar'** - создает tar-архив с файлами исходных текстов.
* **`shar'** - создать самораспаковывающийся архив (shar-файл) из исходных файлов.
* **`dist'** - создать из исходных файлов дистрибутив. Это может быть tar-файл, shar-файл, сжатые их версии или что-нибудь другое.
* **`TAGS'** - обновить таблицу тегов для этой программы.
* **`check', `test'** - выполнить самотестирование программ, построенных этим make-файлом.

***Вместо исполнения команд***

* Make-файл указывает программе make, как определить - нуждается ли цель в обновлении и каким образом ее следует обновлять. Однако, не всегда вам требуется именно обновление цели. Указывая подходящие опции, можно заставить make выполнять другие действия.
* **`-n', `--just-print', `--dry-run', `--recon' -** "Нет операции". Будут печататься (без реального выполнения) команды, которые бы выполнила make для обновления целей.
* **`-t', `--touch'** - цели помечаются как "обновленные" без реального их изменения. Иначе говоря, make просто "делает вид" что скомпилировала нужные цели, на самом деле не изменяя их.
* **`-q', `--question'** - "Проверка". Делается проверка - нуждаются ли цели в обновлении, но никаких команд не исполняется. Никакой компиляции и никакого вывода сообщений при этом не производится.
* **`-W файл', `--what-if=файл', `--assume-new=файл', `--new-file=файл'** - "Что если?". За каждой опцией `-W' следует имя файла. Для указанных таким образом файлов, make предполагает, что их время модификации равно текущему времени (при этом, реальное время модификации этих файлов не меняется). Совместно используя опции **`-W' и `-n'**, можно увидеть, какие действия предпримет make, если перечисленные файлы действительно будут модифицированы.
* При наличии опции **`-n',** make только печатает команды, без реального их выполнения.
* Эффект опции **`-t'** состоит в том, что make игнорирует команды, указанные в правилах и использует вместо них команду **touch** для всех целей, нуждающихся в обновлении. Печатается также команда **touch**, если только не были указаны опции **`-s' или .SILENT.** В действительности, для увеличения скорости, make не вызывает программу **touch**, а выполняет всю требуемую работу "напрямую".
* При наличии опции **`-q'**, make ничего не печатает и не исполняет никаких команд, а просто возвращает соответствующий код возврата. Нулевой код возврата означает, что цели не нуждаются в обновлении. Код возврата, равный единице, означает, что какие-то из целей нуждаются в обновлении. И, наконец, код возврата, равный двум, означает, что произошла ошибка (таким образом, вы можете отличить ошибочную ситуацию от случая, когда цели нуждаются в обновлении).
* При вызове make, можно указать только одну из трех вышеперечисленных опций - задание сразу нескольких опций считается ошибкой.
* Опции **`-n', `-t' и `-q'** не влияют на командные строки, начинающиеся с **`+'**, а также строки, содержащие **`$(MAKE)'** или **`${MAKE}**'. Обратите внимание, что при наличии вышеперечисленных опций будут запускаться только строки, начинающиеся с **`+'** или содержащие **`$(MAKE)'** или **`${MAKE}'** - другие строки тех же правил не будут запускаться.
* Опцию **`-W'** можно использовать двумя путями:
* При наличии опций **`-n' или `-q'**, вы можете увидеть, какие действия предприняла бы make, если бы вы модифицировали указанные файлы.
* При отсутствии опций **`-n' или `-q'**, опции **`-W'** заставят make вести себя во время выполнения команд так, как если бы указанные файлы были модифицированы (хотя, на самом деле, они и не были модифицированы).
* Другую информацию о make и используемых make-файлах вы можете получить с помощью опций **`-p' и `-v'**.

***Предотвращение перекомпиляции некоторых файлов***

* Иногда, после изменения исходного файла, вам хотелось бы избежать перекомпиляции всех файлов, которые от него зависят. Предположим, что вы добавили макрос или прототип функции в заголовочный файл, от которого зависят многие исходные файлы. Будучи "консервативной", make предполагает что любые изменения, внесенные в заголовочные файлы, требуют перекомпиляции всех файлов, которые от них зависят. Вы, однако, понимаете, что, в данном случае, нет необходимости в подобной перекомпиляции и не хотели бы тратить время, ожидая ее завершения.
* Если вы заранее предвидели эту проблему и еще не внесли изменения в заголовочный файл, вы можете воспользоваться опцией `-t'. Эта опция заставит make не исполняя команд, пометить все цели как "обновленные", изменив время их последней модификации. Поступая так, вам следует придерживаться следующей процедуры:
  + Используйте команду `make' для перекомпиляции всех файлов, которые действительно в этом нуждаются.
  + Внесите изменения в заголовочные файлы.
  + Используйте команду `make -t', чтобы пометить все объектные файлы как "обновленные". При следующем запуске make, внесенные в заголовочные файлы изменения, уже не вызовут перекомпиляции программы.
* Однако, если вы уже внесли изменения в заголовочный файл, в то время как у вас еще имеются другие файлы, нуждающиеся в обновлении, использовать приведенную выше процедуру уже поздно. Вместо этого, вы можете воспользоваться опцией **`-o файл'** дабы указанный файл рассматривался как "старый". В таком случае ни этот файл, ни зависящие от него файлы обновляться не будут. Придерживайтесь следующей процедуры:
  + Перекомпилируйте все исходные файлы, которые нуждаются в компиляции по причинам, не связанным с вашей модификацией заголовочного файла, используя **`make -o имя\_заголовочного\_файла'**. Если вы изменили несколько заголовочных файлов, используйте отдельные опции **`-o'** для каждого из них.
  + Обновите время модификации всех объектных файлов, используя **`make -t'.**

***"Перекрытие" (overriding) переменных***

* Аргумент командной строки, содержащий **`='** определяет значение переменной: запись **`v=x'** означает, что переменная **v** получит значение **x**. Если значение переменной было задано подобным образом, то все "обычные" присваивания этой переменной нового значения внутри make-файла будут игнорироваться; мы говорим, что они будут перекрыты (overridden) аргументом командной строки.
* Чаще всего, данное средство используется для передачи дополнительных опций компилятору. Например, в "правильно" написанном make-файле переменная **CFLAGS** используется при каждом вызове компилятора Си, так что исходный файл **`foo.c'** мог бы компилироваться приблизительно так: **cc -c $(CFLAGS) foo.c**
* Таким образом, значение, хранящееся в **CFLAGS**, будет влиять на любой процесс компиляции. Вполне возможно, в make-файле определяется некоторое "обычное" значение для переменной **CFLAGS**, например: **CFLAGS=-g**
* Каждый раз, запуская make, вы, при желании, можете "перекрыть" это значение переменной **CFLAGS**. Например, при вызове **`make CFLAGS='-g -O''**, каждая компиляция будет осуществляться с помощью **`cc -c -g -O'**. (Этот пример также иллюстрирует, как вы можете включать пробелы и другие специальные символы в значение переменной при ее перекрытии.)
* Переменная **CFLAGS** - лишь одна из многих "стандартных" переменных, существующих только для того, чтобы вы могли изменять их подобным образом. Смотрите раздел **Используемые в неявных правилах переменные**, где приведен полный список таких переменных.
* В вашем make-файле вы можете использовать и свои собственные переменные, давая пользователю возможность влиять на ход работы make-файла путем изменения этих переменных.
* Когда вы "перекрываете" значение переменной с помощью командной строки, вы можете определить, как **рекурсивно вычисляемую**, так и **упрощенно вычисляемую** переменную. В приведенном выше примере создавалась рекурсивно вычисляемая переменная; для определения упрощенно вычисляемой переменной, достаточно вместо **`='** использовать оператор **`:='**. С другой стороны, если указанное в командной строке значение переменной не будет ссылаться на другие переменные или функции, тип создаваемой переменной не имеет значения.
* Имеется один способ, с помощью которого внутри make-файла можно изменить значение переменной, заданное с помощью командной строки. Для этого нужно использовать директиву override, которая выглядит следующим образом: **`override переменная = значение'.**

***Проверка компиляции программы***

* Обычно, при возникновении ошибки во время исполнения какой-либо команды, make немедленно прекращает работу, возвращая ненулевой код возврата. Никаких команд, обновляющих какие-либо цели, после этого более не выполняется. Возникновение ошибки означает, что главная цель не может быть корректно обновлена и make сообщает об этой ситуации сразу же, как только это становится ясным.
* Однако, при компиляции программы, которую вы только что модифицировали, такое поведение make не слишком удобно. Скорее, вам хотелось бы, чтобы make попробовала скомпилировать все модифицированные вами файлы дабы обнаружить как можно большее число ошибок.
* В такой ситуации следует использовать опции **`-k' или `--keep-going'**. При этом make будет продолжать обработку других пререквизитов рассматриваемой цели, обновляя их при необходимости. Только после этого make завершит работу с возвратом ненулевого статуса.
* Например, при возникновении ошибки во время компиляции объектного файла, **`make -k'** продолжит компиляции других объектных файлов, хотя уже и понятно, что скомпоновать готовую программу будет невозможно. В дополнении к этому, **`make -k'** будет пытаться продолжать работу как можно дольше и после того, как станет ясно что make не знает, как можно обновить рассматриваемую цель или пререквизит. В этом случае будет выдано соответствующее сообщение об ошибке, но make попытается продолжить работу (в то время, как без опции **`-k'** такая ошибка считалась бы фатальной).
* Обычно, make исходит из того, что ваша задача - получить новую версию главной цели. Запуск с опцией **`-k'** говорит, что ваша задача - протестировать максимальное количество внесенных вами изменений (все обнаруженные таким образом проблемы можно было исправить "сразу", до следующей попытки компиляции). Вот почему, кстати, при выполнении команды M-x compile редактор Emacs по умолчанию использует опцию `-k'.

***Обзор опций***

* **`-b', `-m'** - оставлены для совместимости с другими версиями make. При работе они игнорируются.
* **`-C каталог', `--directory=каталог'** - перед чтением make-файла перейти в каталог *каталог*. При наличии сразу нескольких опций **`-C'**, каждая из них рассматривается относительно предыдущей:

так, **`-C / -C etc'** эквивалентно **`-C /etc'**. Обычно, это используется при рекурсивном вызове make.

* **`-d'** - в дополнение к основной работе, выводить отладочную информацию. Отладочная информация: какие файлы make считает необходимым обновить, для каких файлов сравнивается их время изменения и каков полученный результат, какие неявные правила рассматриваются в качестве кандидатов на исполнение и какие из них действительно выполняются и так далее. Опция **-d** эквивалентна использованию опции **`--debug=a'** (смотрите ниже).
* **`--debug[=опции]'** - в дополнение к основной работе, выводить отладочную информацию. При выводе отладочной информации можно выбрать нужный ее тип и степень "подробности". При отсутствии аргументов, выбирается "базовый" уровень отладочной информации. Ниже перечислены все возможные аргументы; при "разборе" аргументов учитывается только первый символ названия. При задании нескольких аргументов, они должны разделяться пробелами или запятыми:
  + **a (all)** - выдача всей имеющейся отладочной информации ("максимальный уровень"). Эквивалентно использованию опции **`-d'**.
  + **b (basic)** - "базовый уровень" отладочной информации: печатаются все цели, которые были найдены "устаревшими" и информация об успешности или неуспешности попытки их обновления.
  + **v (verbose)** - следующий уровень после "базового"; дополнительно выдается информация о том, какие make-файлы обрабатываются, какие пререквизиты не нуждаются в обновлении и так далее. Включение этой опции также приводит к выдаче отладочной информации "базового" уровня.
  + **i (implicit)** - выдается информация о процессе поиска подходящих неявных правил для каждой из целей. Включение этой опции также приводит к выдаче отладочной информации "базового" уровня.
  + **j (jobs) -** выдача информации о вызове некоторых команд.
  + **m (makefile) -** по умолчанию, описанная выше отладочная информация не выдается на стадии, когда make пробует обновить make-файлы. Данная опция разрешает выдачу отладочной информации в процессе обновления make-файлов. Обратите внимание, что `all' также включает данную опцию. Эта опция также разрешает выдачу отладочных сообщений "базового" уровня.
* **`-e', `--environment-overrides' -** дает переменным, созданным из соответствующих переменных среды "приоритет" перед переменными, определенными внутри make-файла.
* **`-f файл', `--file=файл', `--makefile=файл'** - указанный файл рассматривается в качестве make-файла.
* **`-h', `--help'** - напоминает список опций, распознаваемых make и завершает работу.
* **`-i', `--ignore-errors'** - игнорировать все ошибки, возникающие в любых командах, исполняемых для обновления файлов. Смотрите раздел Ошибки при исполнении команд.
* **`-I каталог', `--include-dir=каталог'** - указывает каталог для поиска включаемых make-файлов. При наличии нескольких опций `-I', поиск в указанных каталогах производится в том порядке, как они были перечислены.
* **`-j [число\_заданий]', `--jobs[=число\_заданий]'** - указывает количество одновременно выполняемых заданий (команд). При отсутствии аргумента, число одновременно выполняемых заданий не ограничено. При наличии сразу нескольких опций `-j', будет действовать только последняя из перечисленных. Смотрите раздел Параллельное исполнение команд, где подробно описан процесс запуска команд. Обратите внимание, что при работе в операционной системе MS-DOS, эта опция игнорируется.
* **`-k', `--keep-going'** - после возникновения ошибки, продолжить, насколько это возможно, обработку make-файла. Хотя цель, при обновлении которой произошла ошибка, уже не сможет быть корректно обновлена, и, следовательно, не могут быть правильно обновлены и все цели, зависящие от нее, make попробует обработать другие пререквизиты этих целей. Смотрите раздел Проверка компиляции программы.
* **`-l [загрузка]', `--load-average[=загрузка]', `--max-load[=загрузка]'** - указывает, что новые задания (команды) не должны запускаться если уже имеется хотя бы одно запущенная задание и загрузка системы равна или превышает значение загрузка (число с плавающей точкой). При отсутствии аргумента, ограничение на максимальную загрузку снимается. Смотрите раздел Параллельное исполнение команд.
* **`-n', `--just-print', `--dry-run', `--recon'** - печатать команды, которые должны выполняться, но не исполнять их. Смотрите раздел Вместо исполнения команд.
* **`-o файл', `--old-file=файл', `--assume-old=файл' -** не обновлять файл даже если он "старше" своих пререквизитов и при обработке других файлов не принимать в рассчет возможные изменения в этом файле. По существу, этот файл обрабатывается как "очень старый" и его правила игнорируются. Смотрите раздел Предотвращение перекомпиляции некоторых файлов.
* **`-p', `--print-data-base'** - перед началом основной работы, распечатать базу данных (правила и значения переменных), полученную в результате чтения make-файла. Печатается также информация о номере версии (аналогично опции `-v', смотрите ниже). Для того, чтобы просто распечатать базу данных, не обновляя при этом никаких файлов, используйте `make -qp'. Для распечатки базы данных с предопределенными правилами и переменными, используйте `make -p -f /dev/null'. Помимо всего прочего, выводимая информация содержит имя файла и номер строки, где было дано определение правила или переменной. Это может оказаться ценным подспорьем для отладки сложных make-файлов.
* **`-q', `--question'** - "Режим проверки". Никаких команд не выполняется и не печатается никаких сообщений. Вся работа make заключается в возврате соответствующего кода завершения. В случае, если указанная цель не нуждается в обновлении, возвращается нулевой код. В случае, если обновление требуется, возвращается код, равный единице. При возникновении каких-либо ошибок ошибок, возвращается код, равный двум. Смотрите раздел Вместо исполнения команд.
* **`-r', `--no-builtin-rules'** - отключает использование встроенных неявных правил (смотрите раздел Использование неявных правил). Однако, вы по-прежнему можете задать свои собственные неявные правила с помощью шаблонных правил (смотрите раздел Определение и переопределение шаблонных правил). Опция `-r' также очищает используемый по умолчанию список суффиксов для суффиксных правил. (смотрите раздел Устаревшие суффиксные правила). По-прежнему, вы можете использовать специальную цель .SUFFIXES для определения своего собственного списка суффиксов. Далее, эти суффиксы можно будет использовать в своих суффиксных правилах. Обратите внимание, что опция -r воздействует только на правила и никак не влияет на используемые по умолчанию переменные (смотрите раздел Используемые в неявных правилах переменные); смотрите описание опции `-R'.
* **`-R', `--no-builtin-variables'** - отключает использование встроенных переменных, используемых неявными правилами (смотрите раздел Используемые в неявных правилах переменные). Разумеется, вы по-прежнему можете определять свои собственные переменные. Включение опции `-R' автоматически приводит к включению опции `-r' (смотрите выше), поскольку нет смысла в наличии неявных правил без наличия переменных, которые в них используются.
* **`-s', `--silent', `--quiet'** - "Бесшумный режим". Отключается печать исполняемых команд. Смотрите раздел Отображение исполняемых команд.
* **`-S', `--no-keep-going', `--stop'** - отменяет опцию `-k'. Как правило, это может потребоваться только при рекурсивном использовании make, когда опция `-k' может быть "унаследована" через переменную MAKEFLAGS от make "верхнего уровня" (смотрите раздел Рекурсивный вызов make) или в случае, если эта опция была установлена через переменную среду MAKEFLAGS.
* **`-t', `--touch'** - не выполняя команд, просто обновляет время последней модефикации файлов (в действительности не изменяя их). Таким образом, make "делает вид" что все необходимые команды были выполнены, дабы "обмануть" последующие запуски make. Смотрите раздел Вместо исполнения команд.
* **`-v', `--version'** - выдает информацию о версии программы make, ее авторах, авторских правах, замечание об отсутствии гарантий и завершает работу.
* **`-w', `--print-directory'** - печатать сообщение с именем текущего каталога до и после обработки make-файла. Это может оказаться полезным при поиске нетривиальных ошибок, связанных с рекурсивным вызовом make. Смотрите раздел Рекурсивный вызов make. (На практике, вам редко когда понадобится указывать эту опцию, поскольку, во многих случаях, make включает ее автоматически; смотрите раздел Опция `--print-directory'.)
* **`--no-print-directory'** - отменить печать рабочего каталога (опции -w). Эта опция может оказаться полезной в тех случаях, когда make автоматически включает опцию -w, а вы не хотели бы получать дополнительные сообщения. Смотрите раздел Опция `--print-directory'.
* **`-W файл', `--what-if=файл', `--new-file=файл', `--assume-new=файл'** - "притвориться", что цель файл только что была модифицирована. Будучи использована совместно с опцией `-n', покажет, какие действия будут выполнены если этот файл действительно будет модифицирован. Без опции `-n', эффект сходен с выполнением команды touch для указанного файла с последующим запуском make, за исключением того, что время последней модификации этого файла происходит только в "воображении" make. Смотрите раздел Вместо исполнения команд.
* **`--warn-undefined-variables'** - когда make будет встречать ссылки на неопределенные переменные, будут выдаваться соответствующие предупреждающие сообщения. Это может оказаться полезным при отладке make-файлов, в которых переменные используются нетривиальным образом.

***Использование неявных правил (implicit rules)***

* **Неявные правила (implicit rules)** указывают make на некоторые "стандартные" приемы обработки файлов, дабы пользователь мог использовать их, не занимаясь каждый раз детальным описанием способа обработки. Так, например, имеется неявное правило для компиляции исходных файлов на языке Си. Вопрос о запуске тех или иных правил решается исходя из имен обрабатываемых файлов.
* Как правило, например, при компиляции программ на Си, из "входного" файла с расширением `.c' получается файл с расширением `.o'. Таким образом, при наличии подобной комбинации расширений файлов, make может применить к ним неявное правило для компиляции Си-программ.
* Неявные правила могут применяться последовательно, связываюсь в "цепочки"; так например, make может получить файл `.o' из файла `.y', используя "промежуточный" файл `.c'. Смотрите раздел "Цепочки" неявных правил.
* Во встроенных неявных правилах используются некоторые переменные, изменяя которые, можно влиять на работу этих правил. Так, например, в неявном правиле для компиляции Си-программ используется переменная **CFLAGS**, содержащая передаваемые компилятору опции.
* Вы можете определить свои собственные неявные правила с помощью **шаблонных правил (pattern rules)**.
* **Суффиксные правила (suffix rules)** - более ограниченный (и устаревший) механизм задания неявных правил. Механизм шаблонных правил является более общим и понятным, однако суффиксные правила по-прежнему поддерживаются из соображений обеспечения совместимости.

***Использование неявных правил (implicit rules)***

* Для того, чтобы предоставить make возможность использовать "общепринятую" методику для обновления целевого файла, достаточно воздержаться от самостоятельного задания команд. Для этого, можно либо задать правило, не содержащее команд, либо вообще не задавать правила. Исходя из имеющихся в наличии исходных файлов, make решит, каким неявным правилом следует воспользоваться.
* В случае, если неявное правило найдено, из него могут быть получены как нужные команды, так и пререквизиты (исходные файлы). Для задания дополнительных пререквизитов, которые не могут быть получены из неявного правила (например, заголовочных файлов), вы можете написать дополнительное правило для `foo.o' без указания команд.
* Каждое неявное правило имеет шаблон цели и шаблоны пререквизитов. Один и тот же шаблон цели может фигурировать сразу в нескольких неявных правилах.
* Например, сразу несколько правил описывают создание файлов `.o': одно - из файлов `.c' с помощью компилятора языка Си; другое - из файлов `.p' с помощью компилятора языка Паскаль; и так далее. В действительности, будет использовано то правило, чьи пререквизиты существуют или могут быть получены. Так, если у вас имеется файл `foo.c', make запустит компилятор Си; в противном случае, если у вас есть файл `foo.p', make запустит компилятор Паскаля; и так далее.
* Выше, мы уже сказали, что неявное правило может быть использовано в том случае, если его пререквизиты "существуют или могут быть построены". Считается, что файл "может быть построен", если его имя упоминается в make-файле в качестве цели или пререквизита, или же для его построения может быть использовано подходящее неявное правило. В случае, когда пререквизит одного неявного правила является результатом работы другого неявного правила, мы говорим, что эти правила связаны в цепочку (происходит **chaining**).
* В общем случае, make производит поиск подходящих неявных правил для каждой цели и каждого правила с двойным двоеточием, которые не имеют команд. Файлы, упоминаемые только в качестве пререквизитов, рассматриваются как цели, описанные в "пустых" правилах (без пререквизитов и команд), поэтому к таким файлам также могут быть применены неявные правила. Смотрите раздел *Алгоритм поиска неявных правил, где детально описана процедура поиска подходящих неявных правил*.
* Заметьте, что явное указание пререквизитов не влияет на процедуру поиска неявных правил. Рассмотрим, например, следующее явное правило: **foo.o: foo.p**
* Наличие пререквизита `foo.p' не означает, что для получения объектного файла `foo.o' make обязательно будет использовать неявное правило, описывающее получение файла `.o' из исходного файла `.p' на языке Паскаль. Например, при наличии файла `foo.c', вместо неявного правила компиляции исходных файлов на Паскале, будет использовано правило для компиляции исходных файлов на Си, поскольку в списке предопределенных неявных правил оно находится ближе к началу списка (смотрите раздел *Перечень имеющихся неявных правил*).
* Если вы не хотите, чтобы неявное правило было использовано для цели, не имеющей команд, задайте для этой цели пустую команду с помощью символа ';' (смотрите раздел Пустые команды).

***Перечень имеющихся неявных правил***

* Эти правила доступны всегда, если только make-файл явно не заменяет или не отменяет их. Опции `-r' и `--no-builtin-rules' отменяют все предопределенные правила.
* Даже в отсутствие опции `-r', не всегда все эти правила являются определенными. Дело в том, что многие из них реализованы в make в виде **суффиксных правил**, поэтому список действительно определенных правил будет зависеть от используемого **списка суффиксов** (списка пререквизитов специальной цели **.SUFFIXES**).
* По умолчанию используется следующий список суффиксов: .out, .a, .ln, .o, .c, .cc, .C, .p, .f, .F, .r, .y, .l, .s, .S, .mod, .sym, .def, .h, .info, .dvi, .tex, .texinfo, .texi, .txinfo, .w, .ch .web, .sh, .elc, .el. Правила, чьи пререквизиты имеют одно из перечисленных расширений, на самом деле являются суффиксными правилами. Если вы модифицируете список суффиксов, "в силе" останутся только те предопределенные правила, чьи суффиксы остались в этом списке; прочие правила будут "отключены".
* На самом деле, для выполнения команд, предопределенные неявные правила используют такие переменные, как COMPILE.c, LINK.p, и PREPROCESS.S, которые содержат команды.
* make следует соглашению, согласно которому, правило, описывающее процесс компиляции исходного файла `.x', использует переменную COMPILE.x. Аналогично, правило для получения исполняемого файла из файла `.x' использует переменную LINK.x, а правило для препроцессорной обработки файла `.x' использует переменную PREPROCESS.x.
* Каждое правило, создающее объектный файл, использует переменную OUTPUT\_OPTION. В зависимости от опций, с которыми была скомпилирована программа make, она определяет эту переменную как содержащую строку `-o $@' или как пустую. Опция `-o' нужна для того, чтобы выходные файлы помещались в нужное место, если исходные файлы находятся в других каталогах (например, при использовании VPATH). Однако, компиляторы в некоторых системах не допускают использования опции `-o' для объектных файлов. Если у вас именно такая система и вы используете VPATH, некоторые скопилированные файлы могут быть помещены в "неверные" каталоги. Для обхода этой проблемы, поместите в переменную OUTPUT\_OPTION строку `; mv $\*.o $@'.

***Используемые в неявных правилах переменные***

* Команды, содержащиеся во "встроенных" неявных правилах, широко используют некоторые предопределенные переменные. Вы можете изменять эти переменные внутри make-файла, с помощью аргументов, передаваемых программе make или с помощью переменных среды, влияя, таким образом, на работу неявных правил. Переменные, используемые неявными правилами, могут быть "отключены" с помощью опций `-R' или `--no-builtin-variables'.
* Например, команда для компиляции исходных текстов на Си выглядит как `$(CC) -c $(CFLAGS) $(CPPFLAGS)'. По умолчанию, первая переменная содержит значение `cc', остальные - пустое значение. В результате, используется команда `cc -c'. Присвоив переменной `CC' значение `ncc', вы заставите make использовать `ncc' для всех компиляций Си-программ, выполняемых предопределенными неявными правилами. Задав переменной `CFLAGS' значение `-g', вы можете включить опцию `-g' для всех выполняемых неявными правилами компиляций.
* Все неявные правила, выполняющие компиляцию программ на Си, используют переменную `$(CC)' для получения имени используемого компилятора и переменную `$(CFLAGS)' для передачи компилятору требуемых опций.
* Переменные, используемые в неявных правилах, можно разделить на два класса: переменные для хранения имен программ (такие, как CC) и переменные для передачи аргументов (такие, как CFLAGS). ("Имя программы", на самом деле, может включать в себя какие-то дополнительные аргументы, но в любом случае оно должно начинаться с имени исполняемого файла программы.) Если значение переменной содержит более одного аргумента, они должны быть разделены пробелами.

***"Цепочки" (chains) неявных правил***

* В некоторых случаях, файл может быть получен путем последовательного применения нескольких неявных правил. Подобная последовательность называется **цепочкой (chain)**.
* Если файл `n.c' существует или упоминается в make-файле, то ситуация проста - make "догадается" что объектный файл может быть построен путем компиляции из файла `n.c'; далее, при рассмотрении способа получения `n.c', будет использовано правило для запуска Yacc. В результате, оба файла - `n.c' и `n.o', будут обновлены.
* Однако, даже если файл `n.c' не существует и не упоминается в make-файле, make знает, как "восстановить" недостающее звено между `n.o' и `n.y'! В этом случае, `n.c' называется промежуточным (intermediate) файлом. Как только make решит воспользоваться промежуточным файлов, он будет занесен в "базу данных" также, как если бы он упоминался в make-файле. Туда же добавляется неявное правило, описывающее процесс создания промежуточного файла.
* Промежуточные файлы порождаются практически также, как и "обычные" файлы. Имеется, однако, два отличия.
* Первое отличие проявляется в случае, когда промежуточный файл не существует. Если "обычный" файл b не существует и make рассматривает цель, зависящую от b, всегда сначала будет создан файл b, а затем будет обновлена зависящая от него цель. Однако, если b является промежуточным файлом, make может оставить "все как есть". Она не будет беспокоиться об обновлении b или рассматриваемой цели, если только какой-нибудь из пререквизитов b не являются "более новым", чем цель, или имеются какие-либо другие причины для обновления цели.
* Второе отличие состоит в том, что, если make создала промежуточный файл b для того, чтобы обновить что-нибудь еще, файл b будет потом удален (когда он станет больше не нужен). Таким образом, если до запуска make промежуточный файл не существовал, его не будет и после завершения работы make. Программа make сообщит вам об удалении, печатая для каждого удаляемого файла соответствующую команду `rm -f'.
* Обычно, файл не может считаться промежуточным, если он упоминается в make-файле в качестве цели или пререквизита. Однако, вы можете явно пометить файл как промежуточный, указав его в качестве пререквизита специальной цели .INTERMEDIATE. Это сработает даже в том случае, если рассматриваемый файл тем или иным образом упоминается "явно".
* Вы можете предотвратить автоматическое удаление промежуточного файла, пометив его как вторичный (secondary) Для этого, укажите нужный файл в качестве пререквизита специальной цели .SECONDARY. Когда файл считается вторичным, make не будет создавать его без крайней необходимости, и, в тоже время, не будет автоматически удалять его. Помечая файл как вторичный, вы также помечаете его как промежуточный.
* Указав шаблон неявного правила (например, `%.o') в качестве пререквизита специальной цели .PRECIOUS, можно предотвратить автоматическое удаление промежуточных файлов, полученных с помощью неявных правил, чьи цели подходят под этот шаблон; смотрите раздел Прерывание или принудительное завершение make.
* "Цепочка" может состоять более чем из двух неявных правил. Например, файл `foo' может быть получен из `RCS/foo.y,v' путем последовательного запуска RCS, Yacc и cc. В этом случае, оба файла `foo.y' и `foo.c' считаются промежуточными и будут в конце удалены.
* Ни одно из неявных правил не может использоваться в цепочке правил более одного раза. Это, например, означает, что make даже не будет пытаться получить файл `foo' из `foo.o.o' путем двухкратного запуска компоновщика. Помимо всего прочего, это ограничение предотвращает возможное зацикливание в процессе поиска подходящих неявных правил для построения цепочки.
* Несколько специальных неявных правил предназначены для оптимизации распространенных случаев, которые, в противном случае, обрабатывались бы с помощью цепочки правил. Например, файл `foo' может быть получен из `foo.c' путем поочередной компиляции и компоновки с использованием отдельных правил, связанных в цепочку (при этом, `foo.o' будет рассматриваться как промежуточный файл). На самом деле, для этого случая существует специальное правило, осуществляющее компиляцию и компоновку за "один шаг", однократным вызовом команды cc. Оптимизированное правило используется вместо цепочки правил, поскольку находится ближе к началу списка правил.

***Определение и переопределение шаблонных правил (pattern rules)***

* Вы можете задать неявное правило, написав **шаблонное правило (pattern rule)**. Шаблонное правило выглядит подобно обычному правилу, за исключением того, что имя его цели содержит специальный шаблонный символ `%' (в точности один). Цель рассматривается как шаблон имени файлов; символ `%' может соответствовать любой непустой подстроке, прочие символы должны совпадать. В именах пререквизитов также используется символ `%', показывающий, как их имена связаны с именем цели.
* Так, шаблонное правило `%.o : %.c' показывает, как файлы с именами `основа.o' могут быть получены из соответствующих файлов с именами `основа.c'.
* "Расширение" символа `%' в шаблонном правиле производится после вычисления всех переменных и функций (что имеет место в момент чтения make-файла).

***Введение в шаблонные правила (pattern rules)***

* Шаблону `%.c' удовлетворяют все имена файлов, оканчивающиеся на `.c'. Под шаблон `s.%.c' подходят все имена файлов, которые начинаются с `s.', заканчиваются на `.c' и имеют длину по крайней мере пять символов. (Требуется, по крайней мере, один символ для сопоставления с `%'.) Подстрока, которая соответствует символу`%', называется **основой (stem)**.
* Символ `%' в пререквизите шаблонного правила означает ту же основу, которая соответствует символу `%' в имени цели. Для того, чтобы шаблонное правило могло быть применено к рассматриваемому файлу, имя этого файла должно подходить под шаблон цели, а из шаблонов пререквизитов должны получиться имена файлов, которые существуют или могут быть получены. Эти файлы станут пререквизитами рассматриваемого целевого файла.
* %.o : %.c ; *команда*...
* указывает, как файл `n.o' может быть получен из файла `n.c', являющегося его пререквизитом, при условии, что `n.c' существует или может быть создан.
* В шаблонном правиле также могут присутствовать пререквизиты, не использующие шаблонный символ `%'; такие пререквизиты будут добавлены к каждому файлу, полученному с помощью этого шаблонного правила. Такие "константные" пререквизиты иногда могут оказаться полезными.
* Шаблонное правило не обязано содержать пререквизитов с `%'. Более того, оно вообще не обязано иметь каких-либо пререквизитов. Такое правило можно рассматривать как "обобщенный" шаблон. Оно будет описывать способ получения любых файлов, подходящих под шаблон цели.
* Когда выполняется поиск подходящего шаблонного правила для обрабатываемой цели, прочие присутствующие в правиле шаблоны целей не принимаются во внимание: для make существенны лишь указанные в правиле пререквизиты и команды рассматриваемой цели. Однако, после выполнения команд, обновляющих эту цель, все прочие цели, перечисленные в шаблонном правиле, также помечаются как обновленные.
* Порядок следования шаблонных правил в make-файле является существенным, поскольку именно в этом порядке они рассматриваются. Из нескольких подходящих правил будет использовано только первое из найденных. Написанные вами правила имеют "приоритет" над предопределенными правилами. Помните, однако, что правила, чьи пререквизиты в действительности существуют или могут быть построены, всегда имеют больший приоритет, нежели правила, чьи пререквизиты, для своего построения, требуют использовать цепочки неявных правил.

***Примеры шаблонных правил***

% :: RCS/%,v

$(CO) $(COFLAGS) $<

* С его помощью, любой файл `x' может быть получен из соответствующего файла `x,v', находящегося в подкаталоге `RCS'. Так как в качестве имени цели указан шаблон `%', это правило может быть применено к любому файлу, имеющему подобный пререквизит. Двойное двоеточие определяет это правило как терминальное (terminal), вследствие чего, его пререквизит не может быть промежуточным файлом.

***Автоматические переменные***

* Значения этих переменных автоматически вычисляются заново для каждого исполняемого правила в зависимости от его целей и пререквизитов.
* $@ - имя файла цели правила. Если цель является элементом архива (archive member), то `$@' обозначает имя архивного файла.
* $% - для целей, являющихся элементами архива, обозначает имя этого элемента. Например, для цели `foo.a(bar.o)' переменная `$%' принимает значение `bar.o', а переменная `$@' - значение `foo.a'. Если цель не является элементом архива, `$%' имеет пустое значение.
* $< - имя первого пререквизита.
* $? - имена всех пререквизитов (разделенные пробелами), которые являются "более новыми", чем цель. Для членов архива, используется имя самого элемента.
* $^ - имена всех пререквизитов (разделенные пробелами). Для пререквизитов, которые являются элементами архивов, используются только имена элементов. Независимого от того, сколько раз конкретный файл был указан в списке пререквизитов, цель будет иметь только одну зависимость от этого файла. Таким образом, если в списке пререквизитов одно и то же имя файла будет фигурировать несколько раз, переменная $^ все равно будет содержать только одну копию этого имени.
* $+ - аналогично `$^', но пререквизиты, перечисленные более, чем один раз, также будут продублированы (в том порядке, как они были указаны в make-файле). В основном, эта переменная может быть полезна в командах компоновки, где порядок следования библиотек и их возможное дублирование является существенным.
* $\* - основа (stem), с которой было сопоставлено неявное правило. Например, для цели `dir/a.foo.b' и шаблона цели `a.%.b', основой будет строка `dir/foo'. Основа имени может быть полезной для конструирования имен взаимосвязанных файлов. В статическом шаблонном правиле, основой является часть имени файла, соотвествующая символу `%' в шаблоне цели. Для явного правила такое определение неприменимо, поэтому `$\*' вычисляется по-другому. В случае явного правила, если имя цели имеет один из "известных" суффиксов, в переменную `$\*' записывается имя цели без этого суффикса. Например, для цели `foo.c', переменная `$\*' будет установлена в `foo', поскольку `.c' является одним из "известных" суффиксов. Программа GNU make поступает столь причудливым образом лишь по соображениям совместимости с другими версиями make. Мы рекомендуем вам избегать использования переменной `$\*' где-либо, кроме неявных или статических шаблонных правил. В случае, если цель явного правила не имеет один известных make суффиксов, для данного правила значением переменной `$\*' является пустая строка.
* Каждая из указанных шести переменных имеет варианты, позволяющие получить вместо полного имени файла только имя каталога, где он расположен, или же только имя файла внутри каталога. Имена этих дополнительных вариантов переменных образуются путем добавления к "основному" имени переменной символов `D' и `F', соответственно. В GNU make эти варианты переменных можно считать устаревшими, поскольку вместо них лучше использовать функции dir и notdir. Заметьте, однако что `D'-варианты переменных не содержат конечного '/', который всегда присутствует на выходе функции dir. Вот список дополнительных вариантов автоматических переменных:
* `$(@D)' - часть имени файла, определяющее имя каталога, где он расположен (без конечного символа '/'). Например, если значением `$@' является `dir/foo.o', то `$(@D)' получит значение `dir'. В случае, если `$@' не содержит символов '/', переменная `$(@D)' будет содержать `.'.
* `$(@F)' - имя файла без имени каталога. Например, если значением `$@' является `dir/foo.o', то `$(@F)' будет содержать `foo.o'. Выражение `$(@F)' эквивалентно `$(notdir $@)'.
* `$(\*D)', `$(\*F)' - часть основы, определяющая имя каталога и имя файла внутри каталога, соответственно; в данном примере, это будут строки `dir' и `foo'.
* `$(%D)', `$(%F)' - для целей, являющихся элементами архивов, определяет имя каталога и имя файла элемента архива. Эти переменные имеют смысл только целей, которые являются элементами архива (имеющих форму `архив(элемент)') и полезны только в случае, если элемент может содержать имя каталога.
* `$(<D)', `$(<F)' - имя каталога и имя файла (внутри каталога) первого пререквизита.
* `$(^D)', `$(^F)' - список каталогов и имен файлов всех пререквизитов.
* `$(?D)', `$(?F)' - список каталогов и имен файлов всех пререквизитов, которые являются "более новыми", чем цель.
* Вы можете обратить внимание на то, что при ссылке на автоматические переменные, чьи имена состоят из одного символа, мы используем запись вида $имя\_переменной, а не $(имя\_переменной), как это практикуется для "обычных" переменных. На самом деле, это лишь ничего не значащее стилистическое соглашение. С тем же успехом, мы могли бы писать `$(<)' вместо `$<'.

***Процедура сопоставления с шаблоном***

* Таким образом, файл `test.o' удовлетворяет шаблону `%.o' и его основой является строка `test'.
* Если шаблон цели не содержит символа '/' (так, обычно, и происходит), то имя каталога будет удалено из имени рассматриваемого файла, перед тем, как оно будет сравниваться с префиксом и суффиксом шаблона. Далее, после сравнения имени файла с шаблоном, это имя каталога будет опять добавлено к уже сгенерированным именам пререквизитов (имя каталога удалялось лишь для нахождения подходящего неявного правила). Так, имя `src/eat' подойдет под шаблон `e%t', а его основой будет являться `src/a'. Далее, при получении имен пререквизитов, имя каталога, содержащегося в выделенной основе, будет добавляться в начало каждого пререквизита, а оставшаяся часть основы будет подставляться вместо символа `%'. Например, основа `src/a' в комбинации с шаблоном пререквизита `c%r', даст имя файла `src/car'.

***Шаблонные правила с призвольным (match-anything) соответствием***

* Когда в качестве цели шаблонного правила выступает шаблон `%', ему может соответствовать любое имя файла. Такие правила мы называем **правилами с произвольным (match-anything) соответствием**. Зачастую, они весьма полезны, но, к сожалению, могут требовать очень много времени для своей обработки, поскольку make должна будет рассмотреть возможность их применения к любому файлу, указанному в качестве цели или пререквизита.
* Допустим, что в make-файле упоминается имя `foo.c'. Для такой цели make должна будет рассмотреть возможность компоновки ее из объектного файла `foo.c.o', возможность использования компилятора Си для компиляции и компоновки ее из файла `foo.c.c', возможность использования компилятора Паскаля для компиляции и компоновки ее из файла `foo.c.p', и множество других аналогичных возможностей.
* В конце концов make также отвергнет подобные возможности, поскольку файлы `foo.c.o' и `foo.c.p' не будут существовать. Однако, из-за большого числа разнообразных возможностей, make придется затратить очень много времени на их проверку.
* Для ускорения работы, существуют два возможных вида ограничений. Каждый раз, при задании правила с произвольным соответствие, вы должны выбрать один или другой вид ограничения.
* Один из вариантов - пометить такое правила как **терминальное**, задав его с помощью двойного двоеточия. Если правило является терминальным, оно может быть применено только в том случае, когда его пререквизит реально существуют. Случаи, когда пререквизит "может быть получен", не рассматриваются. Иными словами, терминальное правило не может служить "окончанием" цепочки правил.
* Если вы не помечаете правило с произвольным соответствием как **терминальное**, оно будет считаться **нетерминальным**. Для таких правил существует другое ограничение - они не могут быть применены к файлам, имеющим некоторые "известные" типы. Такими "известными" типами считаются все типы, которые подходят под шаблон цели любого из неявных правил (кроме правил с произвольным соответствием).
* Так, например, файл `foo.c' подходит под шаблон цели неявного правила `%.c : %.y' (правила для запуска Yacc). Независимо от того, применимо ли в данном случае это правило (это зависит от существования файла `foo.y'), самого факта наличия правила с такой целью, достаточно, чтобы предотвратить попытки применения любых нетерминальных правил с произвольным соответствием к файлу `foo.c'. Таким образом, make даже не будет рассматривать возможность получения файла `foo.c' из таких файлов, как `foo.c.o', `foo.c.c', `foo.c.p' и так далее.
* Смысл этого ограничения состоит в том, что нетерминальные правила с произвольным соответствием используются для создания файлов, содержащих определенный вид данных (например, исполняемых файлов), а файлы с "известными" суффиксами содержат какие-то другие специфические виды данных (например, исходные файлы на Си).
* На самом деле, make содержит большое число специальных шаблонных правил-"пустышек", единственное назначение которых - распознать определенные имена файлов, дабы нетерминальные правила с произвольным соответствием к таким файлам не применялись. Эти правила-"пустышки" не содержат ни команд, ни пререквизитов, и, будучи "бесполезны", игнорируются во всех других случаях.
* Так, например, встроенное неявное правило **%.p :** существует только для того, чтобы файлы с исходными текстами на Паскале (например, `foo.p') сопоставлялись с определенным шаблоном цели, и, соответственно, не тратилось время на поиск файлов `foo.p.o', `foo.p.c' и тому подобных.
* Аналогичные шаблонные правила-"пустышки" существуются для всех суффиксов, которые используются в суффиксных правилах.

***Отмена действия неявных правил***

* Вы можете "перекрыть" встроенное неявное правило (или ваше собственное правило), задав новое шаблонное правило, имеющее такую же цель и пререквизиты, но другие команды. При определении нового правила, встроенное правило будет заменено. "Позиция" нового правила в списке неявных правил, будет зависеть от месторасположения его определения.
* Вы можете отменить встроенное неявное правило, определив собственное шаблонное правило с такой же целью и пререквизитами, но не имеющее команд. Так, например, для отключения правила, запускающего ассемблер, достаточно написать: **%.o : %.s**

***Определение правил "последнего шанса" (last-resort rules)***

* Вы можете определить правило **"последнего шанса" (last-resort rule)**, написав терминальное правило с произвольным соответствием, не имеющее пререквизитов. Такое правило является обычным шаблонным правилом; единственная его особенность заключается в том, что с ним может быть "сопоставлена" любая цель. Таким образом, команды из этого правила будут использованы для всех целей и пререквизитов, не имеющих своих собственных команд и к которым не подходит ни одно из имеющихся неявных правил.
* Например, при тестировании работы make-файла, обычно, важен лишь сам факт наличия требуемых исходных файлов, а не их реальное содержимое. Написав, в таком случае, следующее правило:

%::

touch $@

* вы получите эффект, что все требуемые (в качестве пререквизитов) исходные файлы, при необходимости, будут созданы автоматически.
* Возможен и другой подход - вы можете задать команды, которые будут использованы для целей, не имеющих подходящих правил (даже правил без команд). Для этого надо задать правило со специальной целью **.DEFAULT**. Команды из этого правила будут использованы для всех пререквизитов, не имеющих явных правил и к которым не может быть применено ни одно из неявных правил. Естественно, по умолчанию правила с целью .DEFAULT не существует. Такое правило нужно писать самостоятельно.
* Правило для цели .DEFAULT, не имеющее пререквизитов и команд:

.DEFAULT:

* очищает список команд, ранее определенных для .DEFAULT. Далее, make работает так, как если бы .DEFAULT никогда не определялась.
* Если вы не хотите, чтобы для цели были использованы команды из шаблонного правила с произвольным (match-anything) соответствием или из правила для цели .DEFAULT и, в то же время, вы не хотите, чтобы для цели запускались какие-либо команды, вы можете использовать для нее пустые команды.
* Вы можете использовать правило "последнего шанса" для "перекрытия" части другого make-файла.

***Устаревшие суффиксные правила (suffix rules)***

* **Суффиксные правила (suffix rules)** являются устаревшим способом задания **неявных (implicit) правил**. Поскольку механизм шаблонных (pattern) правил является более "общим" и понятным, суффиксные правила можно считать устаревшими. Программа GNU make поддерживает их только по соображениям совместимости со "старыми" make-файлами. Суффиксные правила разделяются на два вида: с одиночным суффиксом (single-suffix) и с двойным суффиксом (double-suffix).
* Правило с двойным суффиксом содержит в себе пару суффиксов: один для цели и один для пререквизита. Под него подходит любая цель, чье имя имеет указанный суффикс цели. Соответствующее имя пререквизита получается путем замены суффикса цели на суффикс пререквизита в имени обрабатываемого файла. Правило с двойным суффиксом, чей суффикс цели `.o' и суффикс пререквизита `.c', эквивалентно шаблонному правилу `%.o : %.c'.
* Правило с одиночным суффиксом определяет только суффикс исходного файла. Под это правило может подойти любое имя файла, а имя соответствующего пререквизита получается путем добавления указанного суффикса исходного файла. Так, правило с одиночным суффиксом `.c' эквивалентно шаблонному правилу `% : %.c'.
* Суффиксное правило распознается путем сравнения цели каждого правила с определенным в данный момент списком "известных" суффиксов. Когда make видит правило, чья цель является одним из "известных" суффиксов, это правило рассматривается как суффиксное правило с одиночных суффиксом. Если make видит правило, чья цель представляет собой конкатенацию (сцепление) двух "известных" суффиксов, такое правило рассматривается как суффиксное правило с двойным суффиксом.
* Например, оба суффикса - `.c' и `.o' присутствуют в используемом по умолчанию списке "известных" суффиксов. Следовательно, при наличии у правила цели `.c.o', make будет рассматривать его как суффиксное правило с двумя суффиксами. При этом, `.c' будет являться исходным суффиксом, а `.o' будет суффиксом цели. Вот пример устаревшего способа, которым можно задать правило, компилирующее файлы с исходными текстами на языке Си:

.c.o:

$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<

* Суффиксные правила не могут иметь собственных пререквизитов. Более того, при наличии пререквизитов, суффиксное правило будет рассматриваться как обычное правило со "странным" именем целевого файла. Так, например, правило:

.c.o: foo.h

$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<

* описывает процесс получения файла `.c.o' из пререквизита `foo.h', а не рассматривается как шаблонное правило:

%.o: %.c foo.h

$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<

* которое задает способ получения файлов `.o' из файлов `.c' и указывает на зависимость всех файлов `.o', полученных с помощью этого шаблонного правила, от файла `foo.h'.
* Суффиксные правила без команд также не имеют смысла, поскольку не отменяют действие ранее определенных правил (как это делают шаблонные правила, не имеющие команд). Такие правила просто добавляют в базу данных свой суффикс (или пару сцепленных суффиксов) в качестве цели.
* "Известные суффиксы" являются просто списком пререквизитов специальной цели .SUFFIXES. Вы можете добавить в этот список новые суффиксы, указав их в правиле для цели .SUFFIXES, например:

.SUFFIXES: .hack .win

* Здесь, суффиксы `.hack' и `.win' добавляются в конец списка "известных" суффиксов.
* Если вы хотите задать свой собственный список суффиксов, а не добавлять свои суффиксы к существующему списку, используйте правило с целью .SUFFIXES, не имеющее пререквизитов. Такое правило обрабатывается специальным образом и очищает список "известных" суффиксов. Далее, вы можете написать еще одно правило для добавления нужных вам суффиксов. Например:

.SUFFIXES: # Delete the default suffixes

.SUFFIXES: .c .o .h # Define our suffix list

* Задание опций `-r' или `--no-builtin-rules' приводит к тому, что используемый по умолчанию список суффиксов, будет пуст.
* Определенный по умолчанию список "известных" суффиксов (который был определен до того, как make начнет обрабатывать make-файлы) заносится в переменную SUFFIXES. Далее, вы можете менять список суффиксов с помощью правил для специальной цели .SUFFIXES, но это не будет отражаться на значении этой переменной.

***Использование make для обновления архивов***

* **Архивные файлы** представляют из себя файлы, содержащие внутри себя набор файлов. Эти "внутренние" файлы называются **элементами (members)**. Для работы с архивными файлами используется программа **ar**. Одно из основных применений архивных файлов - хранение библиотеки подпрограмм, используемых при компоновке программы.

***Использование элементов архива в качестве целей***

* Отдельные элементы архивного файла могут быть использованы в качестве целей или пререквизитов. Указать элемент *элемент* архива *архив* можно с помощью конструкции: **архив(элемент)**
* Эту конструкцию можно использовать только в именах целей и пререквизитов, но не в командах! Большинство программ не поддерживают такой синтаксис и не могут "напрямую" работать с элементами архивов. Только программа **ar** и другие программы, специально спроектированные для работы с архивами, могут это делать. Поэтому, как правило, все команды для обновления цели, являющейся элементом архива, так или иначе будут использовать программу **ar**. Например, следующее правило создает элемент `hack.o' архива `foolib' путем копирования в архив файла `hack.o':

foolib(hack.o) : hack.o

ar cr foolib hack.o

* На самом деле, практически все цели, являющиеся элементами архивов, обновляются подобным образом; для этого даже существует соответствующее неявное правило. Обратите внимание на необходимость задания для программы ar опции `c' в том случае, если архивный файл пока не существует.
* Для задания сразу нескольких элементов одного архива, можно написать имена всех элементов "вместе" внутри скобок.
* foolib(hack.o kludge.o) эквивалентен: foolib(hack.o) foolib(kludge.o)
* При задании имен элементов архивов можно также использовать шаблонные символы "в стиле" командного интерпретатора. Например, `foolib(\*.o)' будет ссылаться на все существующие элементы архива `foolib', чьи имена заканчиваются на `.o'; возможно, это будут элементы `foolib(hack.o) foolib(kludge.o)'.

***Неявные правила для целей - элементов архива***

* Цель вида **`a(m)'**, означает элемент **m** архивного файла **a**.
* Когда make подбирает подходящее неявное правило для подобной цели, наряду с правилами, которые могут соответствовать цели **`a(m)'**, она рассматривает также и правила, которые могут быть применены к цели **`(m)'**.
* При этом, будет запущено специальное правило с целью `(**%**)'. Это правило обновляет цель **`a(m)'** путем копирования файла **m** в архив. Например, это правило сможет обновить цель **`foo.a(bar.o)'** путем копирования файла **`bar.o'** в архив **`foo.a'** в качестве элемента с именем **`bar.o'**.
* Будучи связанным в цепочку с другими, это правило может сослужить хорошую службу. Так, при наличии файла `bar.c', команды `make "foo.a(bar.o)"' (здесь, кавычки использованы для предотвращения специальной интерпретации символов `(' и `)' интерпретатором командной строки) будет достаточно для запуска такой последовательности команд (при этом, даже не потребуется make-файла):

cc -c bar.c -o bar.o

ar r foo.a bar.o

rm -f bar.o

* В этом примере, make использовала файл `bar.o' в качестве промежуточного.
* Подобные неявные правила пишутся с использованием автоматической переменной `$%'.
* Имя каталога не может содержаться в имени элемента архива, но его можно использовать в make-файле для получения аналогичного эффекта. Например, цель `foo.a(dir/file.o)' будет автоматически обновляться make с помощью команды:

ar r foo.a dir/file.o

* которая будет копировать файл `dir/file.o' в элемент `file.o' архива foo.a. В подобных случаях, могут быть полезны автоматические переменные %D и %F.

***Обновление каталога символов архивного файла***

* Архивный файл, используемый в качестве библиотеки, как правило, содержит специальный элемент с именем `\_\_.SYMDEF', содержащий каталог всех внешних символов, на которые ссылаются другие элементы этого архива. После обновления любого другого элемента архива, вам потребуется обновить `\_\_.SYMDEF', дабы он содержал актуальную информацию. Это делается путем запуска программы ranlib: ranlib *файл\_архива*
* Обычно, эта команда помещается в правило для архивного файла, пререквизитами которого являются все элементы этого архива. Например,

libfoo.a: libfoo.a(x.o) libfoo.a(y.o) ...

ranlib libfoo.a

* Здесь, при обновлении любого из элементов архива (`x.o', `y.o', и так далее), будет запущена программа ranlib, которая обновит каталог архива - элемент `\_\_.SYMDEF'. В приведенном выше фрагменте, правила, обновляющие элементы архива, не показаны; в большинстве ситуаций, вы можете не указывать их явно, положившись на встроенное неявное правило, копирующее файлы в архив (это правило обсуждалось в предыдущем разделе).
* Программа GNU **ar** автоматически обновляет элемент `\_\_.SYMDEF', вследствие чего, необходимость в подобного рода правилах, отпадает.

***Проблемы при использовании архивов***

* Следует быть осторожным при одновременном использовании параллельного исполнения команд (опция -j) и архивных файлов. При работе сразу нескольких программ **ar** с одним и тем же архивным файлов, этот файл может быть поврежден, поскольку эти программы ничего не знают друг о друге и никак не синхронизируют между собой свою работу.

***Суффиксные правила для архивных файлов***

* Для работы с архивными файлами, вы можете использовать суффиксное правило специального вида. Суффиксные правила для архивных файлов можно считать устаревшими, поскольку поддерживаемые в GNU make шаблонные правила для архивов являются более "общим" механизмом. Однако, суффиксные правила для архивов по-прежнему поддерживаются из соображений совместимости с другими реализациями make.
* Суффиксное правило для архивов пишется с использованием целевого суффикса `.a' (это обычный суффикс архивных файлов). Вот пример устаревшей записи - суффиксное правило для обновления библиотеки (которая является архивным файлом) из исходных файлов на языке Си:

.c.a:

$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $\*.o

$(AR) r $@ $\*.o

$(RM) $\*.o

* Это будет работать аналогично следующему шаблонному правилу:

(%.o): %.c

$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $\*.o

$(AR) r $@ $\*.o

$(RM) $\*.o

* На самом деле, make именно так и поступает, видя суффиксное правило с целевым суффиксом `.a'. Любое правило с двойным суффиксом вида `.x.a', преобразуется make в соответствующее шаблонное правило с шаблоном цели `(%.o)' и шаблоном пререквизита `%.x'.
* Поскольку вы вправе использовать суффикс `.a' для совершенно произвольных файлов (а не только для архивов), make также преобразует подобное суффиксное правило в шаблонное правило и "обычным" способом (смотрите раздел Устаревшие суффиксные правила). Так, суффиксное правило `.x.a' будет преобразовано в два шаблонных правила: `(%.o): %.x' и `%.a: %.x'.