Linux.yaroslavl.ru

GNU Make

Программа управления компиляцией

GNU `make` Версия 3.79

Апрель 2000

Richard M. Stallman и Roland McGrath, перевод (C) Владимир Игнатов, 2000

Версия перевода 0.1

Английский оригинал этого текста находится здесь.

Оригинал перевода находится на моей домашней страничке.

Оглавление

Назначение программы make

Утилита make автоматически определяет какие части большой программы должны быть перекомпилированы, и выполняет необходимые для этого действия. В данном руководстве описывается программа GNU make, авторами которой являются Richard Stallman и Roland McGrath. Начиная с версии 3.76, разработку программы ведет Paul D. Smith.

GNU make удовлетворяет требованиям раздела 6.2 стандарта *IEEE Standard 1003.2-1992* (POSIX.2).

В приводимых примерах будут фигурировать программы на языке Си, поскольку они широко распространены. Однако, вы можете использовать make с любым языком программирования для которого имеется компилятор, работающий из командной строки. На самом деле, область применения make не ограничивается только сборкой программ. Вы можете использовать ее для решения любых задач, где одни файлы должны автоматически обновляться при изменении других файлов.

Перед тем, как использовать make, вы должны создать так называемый **make-файл (makefile)**, который будет описывать зависимости между файлами вашей программы, и содержать команды для обновления этих файлов. Как правило, исполняемый файл программы зависит от объектных файлов, которые, в свою очередь, получаются в результате компиляции соответствующих файлов с исходными текстами.

После того, как нужный make-файл создан, простой команды :

будет достаточно для выполнения всех необходимых перекомпиляций если какие-либо из исходных файлов программы были изменены. Используя информацию из make-файла, и, зная время последней модефикации файлов, утилита make решает, каких из файлов должны быть обновлены. Для каждого из этих файлов будут выполнены указанные в make-файле команды.

При вызове make, в командной строке могут быть заданы параметры, указывающие, какие файлы следует перекомпилировать и каким образом это делать. Смотрите раздел Запуск make.

Как читать данное руководство

Если вы - начинающий пользователь make, или просто хотите получить общее представление об этой утилите, то вам следует прочесть несколько первых разделов из каждой главы, пропуская остальные. В каждой главе первые несколько разделов посвящены введению в тему и содержат общую информацию, а последующие разделы содержат специальную и техническую информацию. Исключение составляет раздел Знакомство с make-файлами, который целиком посвящен введению в данную тему.

Если вы знакомы с другими версиями программы make, обратите внимание на раздел Возможности GNU make, в котором описан широкий набор возможностей, имеющихся в утилите GNU make, а также раздел Несовместимость и нереализованные функции, в котором описаны несколько вещей, которые имеются в других реализациях, но отсутствуют в GNU make

Для быстрого получения справки, смотрите разделы Обзор опций и Справочник, а также раздел Имена специальных целей.

Ошибки и проблемы

Если у вас возникли проблемы с использованием GNU make или вам кажется, что вы обнаружили ошибку в ее работе, пожалуйста, сообщите об этом разработчикам. Мы не может обещать невозможного, однако постараемся исправить положение.

Прежде, чем сообщать об ошибке, убедитесь в том что это - действительно ошибка. Еще раз внимательно перечитайте документацию. Если у вас что-то не получается - посмотрите, действительно ли в документации говорится о том, что это можно сделать. Если из документации непонятно - допустимы ли ваши действия или нет, сообщите нам об этом. Это означает ошибку в документации!

Прежде, чем сообщать об ошибке или пытаться исправить ее самостоятельно, попробуйте локализовать ошибку - создать make-файл минимального размера, на котором она проявляется. Затем, вышлите нам

этот make-файл вместе с полученными результатами работы `make`. Укажите также, каких результатов вы на самом деле ожидали - это поможет нам обнаружить возможные ошибки в документации.

Если вы действительно обнаружили проблему, пожалуйста, сообщите нам об этом по следующему адресу:

```
bug-make@gnu.org
```

Пожалуйста, укажите номер версии вашей программы `make`. Вы можете получить эту информацию, набрав в командной строке `` `make --version' ``. Не забудьте также указать тип вашей машины и операционной системы. По возможности, укажите содержимое файла `` `config.h' ``, который получается в результате работы процесса конфигурирования на вашей машине.

Знакомство с make-файлами (makefiles)

Для работы с утилитой `make`, вам понадобится так называемый **make-файл (makefile)**, который будет содержать описание требуемых действий. Как правило, make-файл описывает, каким образом нужно компилировать и компоновать программу.

В этой главе мы обсудим простой make-файл, который описывает, как скомпилировать и скомпоновать программу - текстовой редактор. Наш текстовой редактор будет состоять из восьми файлов с исходными текстами на языке Си и трех заголовочных файлов. Make-файл также может инструктировать `make` как выполнять те или иные действия, когда явно будет затребовано их выполнение (например, удалить определенные файлы в ответ на команду "очистка"). Пример более сложного make-файла будет приведен в разделе Пример "сложного" make-файла.

При компиляции текстового редактора, любой файл с исходным текстом, который был модефицирован, должен быть откомпилирован заново. Если был модефицирован какой-либо из заголовочных файлов, то, во избежании проблем, должны быть перекомпилированы все исходные файлы, которые включали в себя этот заголовочный файл. Каждая компиляция исходного файла породит новую версию соответствующего ему объектного файла. И, наконец, если какие-либо из исходных файлов были перекомпилированы, то все объектные файлы (как "новые", так и оставшиеся от предыдущих компиляций) должны быть заново скомпонованы для получения новой версии исполняемого файла нашего текстового редактора.

Как выглядят правила (rules)

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

```
цель ... : пререквизит ...
        команда
```

. . .
. . .

Обычно, **цель (target)** представляет собой имя файла, который генерируется в процессе работы утилиты `make`. Примером могут служить объектные и исполняемый файлы собираемой программы. Цель также может быть именем некоторого действия, которое нужно выполнить (например, `` `clean' `` - очистить). Подробнее это обсуждает в разделе Абстрактные цели).

Пререквизит (prerequisite) - это файл, который используется как исходдные данные для порождения цели. Очень часто цель зависит сразу от нескольких файлов.

Команда - это действие, выполняемое утилитой `make`. В правиле может содержаться несколько команд - каждая на свое собственной строке. **Важное замечание:** строки, содержащие команды обязательно должны начинаться с символа табуляции! Это - "грабли", на которые наступают многие начинающие пользователи.

Обычно, команды находятся в правилах с пререквизитами и служат для создания файла-цели, если какой-нибудь из пререквизитов был модефицирован. Однако, правило, имеющее команды, не обязательно должно иметь пререквизиты. Например, правило с целью `` `clean' `` ("очистка"), содержащее команды удаления, может не иметь пререквизитов.

Правило (rule) описывает, когда и каким образом следует обновлять файлы, указанные в нем в качестве цели. Для создания или обновления цели, `make` исполняет указанные в правиле команды, используя пререквизиты в качестве исходных данных. Правило также может описывать каким образом должно выполняться некоторое действие. Подробно это обсуждается в разделе Составление правил.

Помимо правил, make-файл может содержать и другие конструкции, однако, простой make-файл может состоять и из одних лишь правил. Правила могут выглядеть более сложными, чем приведенный выше шаблон, однако все они более или менее соответствуют ему по структуре.

Пример простого make-файла

Вот пример простого make-файла, в котором описывается, что исполняемый файл `edit` зависит от восьми объектных файлов, которые, в свою очередь, зависят от восьми соответствующих исходных файлов и трех заголовочных файлов.

В данном примере, заголовочный файл `` `defs.h' `` включается во все файлы с исходным текстом. Заголовочный файл `` `command.h' `` включается только в те исходные файлы, которые относятся к командам редактирования, а файл `` `buffer.h' `` - только в "низкоуровневые" файлы, непосредственно оперирующие буфером редактирования.

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

Для повышения удобочитаемости, мы разбили длинные строки на две части с помощью символа обратной косой черты, за которым следует перевод строки.

Для того, чтобы с помощью этого make-файла создать исполняемый файл `edit', наберите:

```
make
```

Для того, чтобы удалить исполняемый и объектные файлы из директории проекта, наберите:

```
make clean
```

В приведенном примере, целями, в частности, являются объектные файлы `main.o' и `kbd.o', а также исполняемый файл `edit'. К пререквизитам относятся такие файлы, как `main.c' и `defs.h'. Каждый объектный файл, фактически, является одновременно и целью и пререквизитом. Примерами команд могут служить `cc -c main.c' и `cc -c kbd.c'.

В случае, если цель является файлом, этот файл должен быть перекомпилирован или перекомпонован всякий раз, когда был изменен какой-либо из его пререквизитов. Кроме того, любые пререквизиты, которые сами генерируются автоматически, должны быть обновлены первыми. В нашем примере, исполняемый файл `edit' зависит от восьми объектных файлов; объектный файл `main.o' зависит от исходного файла `main.c' и заголовочного файла `defs.h'.

За каждой строкой, содержащей цель и пререквизиты, следует строка с командой. Эти команды указывают, каким образом надо обновлять целевой файл. В начале каждой строки, содержащей команду, должен находится символ табуляции. Именно наличие символа табуляции является

признаком, по которому make отличает строки с командами от прочих строк make-файла. Имейте ввиду, что make не имеет ни малейшего представления о том, как работают эти команды. Поэтому, ответственность за то, что выполняемые команды нужным образом обновят целевой файл, целиком ложится на вас. Утилита make просто исполняет указанные в правиле команды если цель нуждается в обновлении.

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)*. Абстрактные цели подробно рассматриваются в разделе Абстрактные цели. В разделе Ошибки при выполнении команд описано, как заставить make игнорировать ошибки, которые могут возникнуть при выполнении команды rm и любых других команд.

Как make обрабатывает make-файл

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. В разделе Аргументы для задания главной цели обсуждается, каким образом можно явно задать главную цель.

В примере из предыдущего раздела, главная цель заключалась в обновлении исполняемого файла `edit', поэтому мы поместили данное правило в начало make-файла.

Таким образом, когда вы даете команду:

```
make
```

make читает make-файл из текущей директории и начинает его обработку с первого встреченного правила. В нашем примере это правило обеспечивает перекомпоновку исполняемого файла `edit'. Однако, прежде чем make сможет полностью обработать это правило, ей нужно обработать правила для всех файлов, от которых зависит `edit'. В данном случае - от всех объектных файлов программы. Каждый из этих объектных файлов обрабатывается согласно своему собственному правилу. Эти правила говорят, что каждый файл с расширением `.o' (объектный файл) получается в результате компиляции соответствующего ему исходного файла. Такая компиляция должна быть выполнена, если исходный файл или какой-либо из заголовочных файлов, перечисленных в качестве пререквизитов, являются "более новыми", чем объектный файл, либо объектного файла вообще не существует.

Другие правила обрабатывается потому, что их цели прямо или косвенно являются пререквизитами для главной цели. Если какое-либо правило никоим образом не "связано" с главной целью (то есть ни прямо, ни косвенно не являются его пререквизитом), то это правило не обрабатывается. Чтобы задействовать такие правила, придется явно указать `make` на необходимость их обработки (подобным, например, образом: `make clean`).

Перед перекомпиляцией объектного файла, `make` рассматривает необходимость обновления его пререквизитов, в данном случае - файла с исходным текстом и заголовочных файлов. В нашем make-файле не содержится никаких инструкций по обновлению этих файлов - файлы с расширениями `.c' и `.h' не являются целями каких-либо правил. Таким образом, утилита `make` не предпринимает никаких действий с этими файлами. Однако, `make` могла бы автоматически обновлять и исходные тексты, если бы они, например, генерировались с помощью программ, подобных Bison или Yacc, и для них были бы определены соответствующие правила.

После перекомпиляции объектных файлов, которые нуждаются в этом, `make` принимает решение - нужно ли перекомпоновывать файл `edit'. Это нужно делать, если файла `edit' не существует или какой-нибудь из объектных файлов по сравнению с ним является более "свежим". Если какой-либо из объектных файлов только что был откомпилирован заново, то он будет "моложе", чем файл `edit'. Соответственно, файл `edit' будет перекомпонован.

Так, если мы модефицируем файл `insert.c' и запустим `make`, этот файл будет скомпилирован заново для обновления объектного файла `insert.o', и, затем, файл `edit' будет перекомпонован. Если мы изменим файл `command.h' и запустим `make`, то будут перекомпилированы объектные файлы `kbd.o', `command.o' и `files.o', а затем исполняемый файл `edit' будет скомпонован заново.

Упрощение make-файла с помощью переменных

В приведенном выше примере, в правиле для `edit' нам дважды пришлось перечислять список объектных файлов программы:

```
edit : main.o kbd.o command.o display.o \
               insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                    insert.o search.o files.o utils.o
```

Подобное дублирование чревато ошибками. При добавлении в проект нового объектного файла, можно добавить его в один список и забыть про другой. Мы можем устранить подобный риск, и, одновременно, упростить make-файл, используя переменные. **Переменные (variables)** позволяют, один раз определив текстовую строку, затем использовать ее многократно

в нужных местах. Переменные подробно обсуждаются в разделе Использование переменных).

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. Мы могли бы определить подобную переменную с именем `objects` таким образом:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Далее, всякий раз, когда нам нужен будет список объектных файлов, мы можем использовать значение этой переменной с помощью записи `` `$(objects)' `` (смотрите раздел Использование переменных).

Вот как будет выглядеть наш простой пример с использованием переменной для хранения списка 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 : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

Неявные правила упрощают make-файл

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, так называемое, **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. Смотрите раздел Использование неявных правил.

Когда файл с расширением `` `.c' `` автоматически используется подобным образом, он также автоматически добавляется в список пререквизитов

"своего" объектного файла. Таким образом, мы вполне можем убрать файлы с расширением `.c' из списков пререквизитов объектных файлов.

Вот новый вариант нашего примера, в который были внесены оба описанных выше изменения, а также используется переменная objects:

```
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
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

Примерно так и выглядят make-файлы в реальной практике. Для правила с `clean' здесь использована более сложная запись, которую мы обсудим позже (смотрите разделы Абстрактные цели, и Ошибки при исполнении команд).

Из-за своего удобства, неявные правила широко используются и играют важную роль в работе make.

Другой стиль написания make-файлов

Если для создания объектных файлов используются только неявные правила, то можно использовать другой стиль написания make-файлов. В таком make-файле 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.

Какой стиль построения make-файлов предпочесть - является делом вкуса. Альтернативный стиль более компактен, однако он нравится не всем - многие считают более "естественным" располагать информацию о каждой

цели в одном месте, а не "распылять" ее по make-файлу.

Правило для очистки каталога

Компиляция программы - не единственная вещь, для которой вы, возможно, захотите написать правила. Часто, в make-файле указывается, каким образом можно выполнить некоторые другие действия, не относящиеся к компиляции программы. Таким действием, например, может быть удаление все объектных и исполняемых файлов программы для очистки каталога.

Вот как можно было бы написать правило для очистки каталога в нашем проекте текстового редактора:

```
clean:
        rm edit $(objects)
```

На практике, скорее всего, мы бы записали это правило чуть более сложным способом, предполагающим возможность непредвиденных ситуаций:

```
.PHONY : clean
clean :
        -rm edit $(objects)
```

Такая запись предотвратит возможную путаницу если, вдруг, в каталоге будет находится файл с именем `clean', а также позволит make продолжить работу, даже если команда rm завершится с ошибкой (смотрите раздел Абстрактные цели, а также раздел Ошибки при исполнении команд.)

Подобное правило не следует помещать в начало make-файла, поскольку мы не хотим, чтобы оно запускалось "по умолчанию"! В нашем примере, мы помещаем данное правило в конец make-файла, чтобы главной целью по умолчанию оставалась сборка файла edit.

Поскольку clean не является пререквизитом цели edit, это правило не будеть выполняться, если вызывать `make' без аргументов. Для запуска данного правила, нужно будет набрать `make clean'. Смотрите раздел Запуск make.

Создание make-файлов

Make-файл является хранилищем информации, указывающей программе make, каким образом нужно перекомпилировать проект.

Из чего состоят make-файлы

Make-файл может состоять из конструкций пяти видов: **явные правила**, **неявные правила**, **определения переменных**, **директивы** и **комментарии**. Правила, переменные и директивы подробно рассматриваются в следующих главах.

- **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. Смотрите раздел Составление правил.
- **Неявное правило (implicit rule)** описывает, когда и каким образом нужно обновлять некоторую группу файлов, имена которых подходят под определенный шаблон. Такое правило описывает, как цель может зависеть от файла со "сходным" именем и задает команды для обновления целей. Смотрите раздел Использование неявных правил.
- **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 (смотрите раздел Упрощение make-файла с помощью переменных).
- **Директива** указывает программе `make` на необходимость произведения некоторого специального действия во время чтения make-файла. Возможны, в частности, следующие действия:
  - Чтение другого make-файла (смотрите раздел Подключение других make-файлов).
  - Решение (на основе значения переменных) об использовании или игнорировании части make-файла (смотрите раздел Условные части make-файла).
  - Определение многострочной переменной, состоящей из нескольких строк (смотрите раздел Многострочные переменные).
- Символ `` `#' `` обозначает начало **комментария**. Весь текст, начиная с этого символа и до конца строки, будет игнорирован. Комментарий может быть продолжен на следущую строку с помощью одиночного символа обратной косой черты, находящегося в конце строки. Комментарии могут находиться практически в любом месте make-файла за несколькими исключениями. Они не могут находится внутри директивы `define` и, возможно, внутри команд (поскольку, здесь уже интерпретатор командной строки будет решать - что именно является комментарием). Строка make-файла, целиком состоящая из комментария, рассматривается как пустая и игнорируется.

Имена make-файлов

По умолчанию, когда `make` ищет make-файл для обработки, она поочередно пробует найти файлы со следующими именами (в указанном порядке): `` `GNUmakefile' ``, `` `makefile' `` и `` `Makefile' ``.

Обычно, вам имеет смысл давать своему make-файлу имя `` `makefile' ``, либо `` `Makefile' ``. Мы рекомендуем использовать имя `` `Makefile' ``, потому что при выводе содержимого каталога, файл с таким именем будет находится в начале списка, наряду с такими важными файлами как `` `README' ``. Первое из проверяемых имен - `` `GNUmakefile' `` - не может быть рекомендовано для большинсства make-файлов. Это имя можно использовать, если ваш

make-файл специфичен для GNU `make` и не будет обрабатываться другими версиями `make`. Другие версии программы `make` ищут make-файлы с именами `` `makefile' `` и `` `Makefile' ``, но не `` `GNUmakefile' ``.

В том случае, если `make` не может найти файлов с перечисленными выше именами, то она пробует продолжить работу без использования make-файла. В таком случае, при вызове `make` вы должны явно указать главную цель и утилита попробует достичь этой цели, используя только "встроенные" в нее неявные правила. Смотрите раздел Использование неявных правил.

Если вы хотите использовать "нестандартное" имя для вашего make-файла, вы можете указать его в командной строке, используя опции `` `-f' `` или `` `--file' ``. Аргументы `` `-f *имя_файла*' `` или `` `--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. При наличии в командной строке опций `` `-f' `` или `` `--file' ``, автоматического поиска make-файлов с именами `` `GNUmakefile' ``, `` `makefile' `` и `` `Makefile' ``, не производится.

Подключение других make-файлов

Встретив в make-файле директиву `include`, `make` приостанавливает чтение текущего make-файла и, прежде чем продолжить работу, прочитывает один или несколько указанных в этой директиве make-файлов. Эта директива представляет собой строку make-файла, выглядящую подобным образом:

```
include имена_файлов...
```

Имена файлов могут представлять собой шаблоны имен, допустимые в интерпретаторе командной строки.

В начале строки могут находится дополнительные пробелы - все они будут игнорированы. Наличие символа табуляции в начале строки недопустимо, поскольку такие строки `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. В конце строки с директивой может находится комментарий, начинающийся, как обычно, с символа `` `#' ``. Если имена файлов содержат ссылки на переменные или функции, то эти ссылки "раскрываются" (вместо них подставляются вычисленные значения). Смотрите раздел Использование переменных.

Если, например, у вас есть три файла с расширением `` `.mk' `` - `` `a.mk' ``, `` `b.mk' ``, и `` `c.mk' ``, а переменная `$(bar)` ссылается на строку `bish bash`, то следующая запись

```
include foo *.mk $(bar)
```

будет эквивалентна

```
include foo a.mk b.mk c.mk bish bash
```

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 оказаться полезной если, предположим, у нас имеется несколько программ, собираемых при помощи отдельных make-файлов, которым требуется наличие некоторого "общего" набора определений переменных (смотрите раздел Установка значения переменной) или шаблонных правил (смотрите раздел Определение и переопределение шаблонных правил).

Другой случай, когда директива `include` может быть использована - это 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, которое традиционно практикуется при работе с другими версиями `make`. Смотрите раздел Автоматическая генерация списка пререквизитов.

Если указанное в директиве имя начинается не с символа '/' и файл с таким именем отсутствует в текущей директории, производится его поиск еще в нескольких каталогах. Сначала поиск производится во всех каталогах, которые были указаны в командной строке с помощью опций `` `-I' `` и `` `--include-dir' `` (смотрите раздел Обзор опций). Затем, поиск производится поочередно в следующих директориях (если, конечно, они существуют): `` `*prefix*/include' `` (обычно `` `/usr/local/include' `` (1)) `` `/usr/gnu/include' ``, `` `/usr/local/include' ``, `` `/usr/include' ``.

Если поиск включаемого make-файла завершился неудачно, `make` выдает предупреждающее сообщение, которое, однако 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. Смотрите раздел Автоматическое обновление make-файлов. 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`*`...`

Эта директива работает аналогично директиве `include`, за исключением того, что отсутствие включаемых make-файлов *имена_файлов* не вызывает ошибки (даже не выдается каких-либо 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. Этот механизм работает во многом аналогично директиве `include`. Аналогичным образом производится и поиск этих дополнительных make-файлов в разных каталогах (смотрите раздел Подключение других make-файлов). При этом, главная цель не может браться из этих файлов, а отсутствие какого-либо из них не рассматривается как ошибка.

Одно из основных применений переменной `MAKEFILES` - это организация "связи" между рекурсивными вызовами `make` (смотрите раздел Рекурсивный вызов `make`). Обычно, нежелательно устанавливать переменную `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 (смотрите раздел Поиск пререквизитов по каталогам).

Некоторые пользователя соблазняются возможностью автоматически устанавливать переменную `MAKEFILES` при входе в систему, и пишут свои make-файлы в рассчете на это. Делать этого 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`. Смотрите раздел Подключение других make-файлов .

Автоматическое обновление make-файлов

Иногда make-файлы могут быть получены из других файлов, таких как файлы RCS или SCCS. Если make-файл может быть получен из других файлов, скорее всего, вы захотите, чтобы `make` всегда работала с самой "свежей" версией этого файла.

Для этого, после чтения всех make-файлов, утилита `make` поочередно рассматривает каждый из них в качестве главной цели, пробуя обновить их. Если в make-файле имеется правило (найденное в этом же make-файле, либо в каком-нибудь другом), указывающее на способ его обновления, или имеется неявное правило, которое может быть к нему применено, то этот

make-файл, при необходимости, обновляется (смотрите раздел Использование неявных правил). После того, как все make-файлы были проверены, если хотя бы один из них был действительно обновлен, `make` начинает всю процедуру сначала и перечитывает все make-файлы заново. Возможно, утилита опять попытается обновить некоторые из make-файлов, но как правило, они уже не будут меняться, поскольку только что была получена их самая свежая версия.

Если вам заранее известно, что некоторые ваши make-файлы не могут быть "обновлены", и вы хотите, чтобы `make` не пыталась искать подходящие для них неявные правила (скажем, по соображениям эффективности), вы можете использовать любой "стандартный" прием для отключения поиска неявных правил. Например, вы можете создать явное правило с пустой командой, где в качестве цели выступает нужный вам make-файл (смотрите раздел Пустые команды).

Если в make-файле для обновления файла имеется правило с двойным двоеточием (double-colon rule), не имеющее пререквизитов, то этот файл всегда будет обновляться (смотрите раздел Правила с двойным двоеточием). В случае, если бы целью такого правила являлся make-файл, мог бы возникнуть бесконечный цикл: make-файл все время бы обновлялся, что, в свою очередь, заставляло бы `make` заново перечитывать все make-файлы и так далее, до бесконечности. Поэтому, во избежание зацикливания, `make` **не** пытается обновить make-файлы, которые являются целями правил с двойным двоеточием без пререквизитов.

В том случае, если при запуске `make`, ей не были указаны make-файлы для обработки (с помощью опций `` `-f' `` и `` `--file' ``), то утилита попытается найти подходящий make-файл, поочередно пробую принятые по умолчанию имена make-файлов (смотрите раздел Имена make-файлов). В отличие от make-файлов, указываемых с помощью опций `` `-f' `` и `` `--file' ``, `make` не может быть уверена, что эти файлы вообще существуют. Однако, если make-файл с именем, принятым по умолчанию, в данный момент не существует, но может быть построен с использованием каких-либо известных `make` правил, то вы, скорее всего захотите, чтобы эти правила были выполнены и нужный make-файл был создан.

Поэтому, если ни один из make-файлов с принятыми по умолчанию именами не существует, `make` предпринимает попытку создать их (в том же самом порядке, каком происходил их поиск). Смотрите раздел Имена make-файлов . Эти попытки продолжаются до тех пор, пока какой-либо из make-файлов не будет создан, либо `make` "перепробует" все имена make-файлов, принятых по умолчанию. Заметьте, что невозможность найти или построить make-файл не является ошибкой, поскольку наличие make-файла не является обязательным условием работы `make`.

При использовании опций `` `-t' `` или `` `--touch' `` (смотрите раздел Вместо исполнения команд), вряд ли вы захотите оказаться в ситуации, когда для определения того, какие цели нужно пометить как обновленные, будет использована устаревшая версия make-файла. Поэтому, опция `` `-t' `` не оказывает влияния на процедуру обновления make-файлов - они обновляются даже тогда, когда она указана. Аналогично, опции `` `-q' `` (или

`--question') и `-n' (или `--just-print') не отменяют процедуру обновления файлов; в противном случае, использование устаревшей версии make-файла могло бы вызвать некорректную работу этих опций. Таким образом, при обработке `make -f mfile -n foo', файл `mfile' будет обновлен, затем он будет перечитан заново, и, после этого, будут напечатаны команды, обновляющие цель `foo' и ее пререквизиты. Эти команды будут соответствовать работе с обновленной версией `mfile'.

Однако, в определенных ситуациях, вам может понадобиться избежать обновления make-файлов. Вы может сделать это, указав в командной строке эти файлы в качестве целей и, одновременно, указать их (с помощью опций `-f' и `--file') в качестве make-файлов. Когда make-файл явно указан в командной строке в качестве цели, опция `-t' и аналогичные опции, могут быть к нему применены.

Таким образом, при обработке `make -f mfile -n mfile foo' будет прочитан make-файл `mfile', затем будут напечатаны команды, требуемые для его обновления (без их реального исполнения), и, затем, будут напечатаны команды, необходимые для обновления `foo' (также без их выполнения). Команды для обновления `foo' будут соответствовать нынешнему состоянию `mfile'.

"Перекрытие" (overriding) части make-файла

Иногда, возникает потребность в нескольких make-файлах, лишь незначительно различающихся между собой. Зачастую, в такой ситуации может быть использована директива `include': нужный make-файлы можно получить путем включения другого make-файла и добавления своего набора правил или определений переменных. Однако, с помощью подобной методики, вам не удастся задать разные команды для обновления одной и той же цели. Этого можно добиться другим способом.

Во "внешнем" make-файле (make-файле, который включает в себя 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. Смотрите раздел Определение и переопределение шаблонных правил, где подробно описаны шаблонные правила.

Например, если у нас имеется make-файл с именем `Makefile', который описывает цель `foo' (и другие цели), то 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. Далее, на втором проходе, это внутреннее представление используется для определения того, какие именно цели нуждаются в обновлении и исполняются соответствующие правила.

Понимание такой двухпроходной схемы является важным, поскольку она оказывает непосредственное влияние на ход вычисления переменных и функций; непонимание, зачастую, является источником недоразумений при написании make-файлов. Здесь мы опишем, как происходит "пофазная" обработка различных конструкций. Мы будем говорить, что расширение (expansion) является **немедленным (immediate)**, если оно производится во время первой фазы работы: это означает, make будет вычислять переменные и функции в момент считывания и "разбора" make-файла. Мы будем говорить, что расширение является **отложенным (deferred)**, если оно 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.

Возможно, вы еще не знакомы со всеми конструкциями. В таком случае, вы сможете вернуться к данному разделу потом, когда вы ознакомитесь с этими конструкциями в следующих главах.

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.

Условные конструкции

Все условные конструкции (во всех формах - `ifdef`, `ifeq`, `ifndef` и `ifneq`) целиком и полностью обрабатываются "немедленно".

Определения правил

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)

Правила (rules) содержатся в make-файле и описывают, когда и каким образом должны быть обновлены или созданы некоторые файлы, называемые **целями (targets)**. Чаще всего, каждое правило содержит только одну цель. В правиле перечисляются файлы, которые являются **пререквизитами (prerequisites)** для этой цели и **команды**, которые должны быть выполнены для создания или обновления цели.

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. Здесь есть два 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 (смотрите раздел Определение и переопределение шаблонных правил).

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'). Смотрите раздел Аргументы для задания главной цели.

Синтаксис правил

В общем виде, правило выглядит так:

```
цели : пререквизиты
        команда
        ...
```

или так:

```
цели : пререквизиты ; команда
        команда
        ...
```

Цели (targets) - это имена файлов, разделенные пробелами. В именах целей могут быть использованы шаблонные символы (смотрите раздел Использование шаблонных символов в именах файлов). 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 (смотрите раздел Использование элементов архива в качестве целей). Обычно, в правиле содержится только одна цель, однако, иногда имеет смысл задать несколько целей в одном правиле (смотрите раздел Правила с несколькими целями).

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. Смотрите раздел Написание команд.

Поскольку знак доллара используется для ссылки на переменные, для использования его в правилах, нужно писать `$$' (смотрите раздел Использование переменных). Длинные строки make-файла могут быть разделены на части с помощью символа '\', находящегося в конце строки. Это может повысить удобочитаемость make-файла, но "технической" необходимости в этом нет - 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 (смотрите раздел Использование make для обновления архивов). 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'). Смотрите раздел Написание команд.

Использование шаблонных символов (wildcard characters) в именах файлов

При использованием **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.

Символ `~' в начале имени файла, также имеет специальное значение. Одиночный символ `~' или сочетание `~/' означает ваш домашний каталог. Например, выражение `~/bin' будет означать `/home/you/bin'. Если сразу за символом `~' следует некоторое имя, такая строка будет представлять собой домашнюю директорию пользователя с этим именем. Например, строка `~john/bin' будет означать `/home/john/bin'. В системах, где пользователи не имеют своего домашнего каталога (таких как MS-DOS или MS-Windows), такое поведение может эмулироваться с помощью установки переменной окружения *HOME*.

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'.

Примеры шаблонных имен

Шаблонные имена могут быть использованы в командах, которые содержатся в правилах. Такие имена будут "раскрыты" интерпретатором командной строки. Вот пример правила для удаления всех объектных файлов из текущей директории:

```
clean:
        rm -f *.o
```

Шаблоны также могут быть полезны в качестве пререквизитов правил. В следующем примере, команда `make print' вызовет печать всех файлов с исходными текстами (файлов с расширением `.c'), которые были модефицированы с тех пор, как вы последний раз распечатывали их подобным образом:

```
print: *.c
        lpr -p $?
        touch print
```

В данном правиле, цель `print' является пустой целью (empty target file); смотрите раздел Использование пустых целей для фиксации событий. Автоматичесая переменная `$?' используется для печати только тех пререквизитов, которые были изменены (смотрите раздел Автоматические переменные.)

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)
```

Смотрите раздел Функция `wildcard`.

Проблемы при использовании шаблонных имен

Вот простой пример "неправильного" использования шаблонного имени, результат которого совершенно отличен от ожидаемого. Предположим, составляя make-файл, вы хотели сказать, что исполняемый файл `foo' собирается из всех объектных файлов, находящихся в текущем каталоге, и записали это следующим образом:

```
objects = *.o

foo : $(objects)
        cc -o foo $(CFLAGS) $(objects)
```

При такой записи, переменная `objects` получит значение `*.o'. Расширение шаблона `*.o' будет произведено только при обработке правила с `foo', и пререквизитами этой цели станут все *существующие* в данный момент файлы `.o'. При необходимости, эти объектные файлы будут перекомпилированы.

Но что будет, если вы удалите все файлы с расширением `.o'? Когда шаблону не соответствует ни один файл, этот шаблон остается в "первоначальном" виде. И, таким образом, получится что цель `foo' будет

зависеть от файла со странным именем `*.o'. Поскольку, такого файла на самом деле не существует, make аварийно завершит работу, выдав сообщение, что она не знает как построить файл `*.o'. Пожалуй, это совсем не то, чего вы хотели добиться!

На самом деле, нужный вам результат получить вполне возможно, но для этого надо использовать более сложную методику, использующую функцию wildcard и строковые подстановки. Подобная методика будет обсуждаться в следующих разделах.

В операционных системах фирмы Microsoft для разделения имен директорий используется символ '\':

```
c:\foo\bar\baz.c
```

Приведенное выше имя эквивалентно имени `c:/foo/bar/baz.c' в стиле Unix (здесь `c:' - это, так называемое, имя диска). Когда программа make работает в таких операционных системах, она допускает использование обоих символов ('/' и '\') в именах файлов. Поддержка символа '\' *не* распространяется, однако, на шаблонные имена, где этот символ имеет специальное значение. В таких случаях, вы *должны* использовать имена в стиле Unix (с символом '/' в качестве "разделителя").

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, которая выглядит следующим образом:

```
$(wildcard шаблон...)
```

Подобная строка, будучи использована в любом месте make-файла, будет заменена списком существующих в данный момент файлов, которые удовлетворяет указанному шаблону (шаблонам). Имена файлов отделяются друг от друга пробелами. В том случае, если не будет найдено файлов, удовлетворяющих заданному шаблону, функция возвращает пустую строку. Заметьте, что такое поведение функции wildcard отличается от поведения обычных шаблонов в правилах, которые, в таких случаях, остаются в исходном виде, а не игнорируются (смотрите раздел Проблемы при использовании шаблонных имен).

Одно из возможных применений фукнции wildcard - это получение списка исходных файлов, находящихся в текущем каталоге, например:

```
$(wildcard *.c)
```

Затем, мы может превратить список исходных файлов в список объектных файлов, заменив их расширение с `.c' на `.o', например:

```
$(patsubst %.c,%.o,$(wildcard *.c))
```

(Здесь, для замены текста, мы использовали функцию `patsubst`. Смотрите раздел Функции анализа и подстановки строк.)

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. Смотрите раздел Две разновидности переменных, где описывается оператор `:=', который является вариантом "стандартного" оператора `='.

Поиск пререквизитов по каталогам

Для больших систем, часто бывает полезным хранить бинарные файлы программы и файлы с ее исходными текстами отдельно, в разных каталогах. Утилита `make` может способствовать использованию такой методики с помощью механизма автоматического **поиска пререквизитов по каталогам**. Когда вы будете распределять исходные файлы по директориям, вам не придется менять отдельные правила, нужно лишь будет указать пути для поиска этих файлов.

Переменная `VPATH`: список каталогов для поиска пререквизитов

Значение переменной `VPATH` указывает утилите `make` список директорий, где следует производить поиск файлов. Чаще всего, этот путь представляет собой список каталогов с файлами, которые являются пререквизитами каких-либо правил и находятся не в текущем каталоге. Однако, содержимое переменной `VPATH` используется для поиска любых файлов (а не только пререквизитов), в том числе и файлов, которые являются целями каких-либо правил.

Таким образом, если файл, который является целью или пререквизитом, не найден в текущем каталоге, `make` предпримет попытку найти его в каталогах, перечисленных в `VPATH`. Если в одном из этих каталогов файл будет найден, он может стать пререквизитом (смотрите ниже). Учитывая это, правила могут составляться таким образом, что имена пререквизитов указываются так, как если бы они находились в текущем каталоге. Смотрите раздел Написание команд с учетом поиска по каталогам.

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).

Например, следующая запись:

```
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

Средством, аналогичным переменной VPATH, но только более "избирательным", является директива vpath (обратите внимание на нижний регистр букв). Эта директива позволяет задать пути поиска для некоторой группы файлов, а именно, файлов, чье имя подходит под определенный шаблон. Таким образом, вы можете задать некоторый список каталогов поиска для одной группы файлов, и совсем другой список - 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 (смотрите раздел Определение и переопределение шаблонных правил), может соответствовать любой последовательности символов (в том числе и пустой). Например, шаблону %.h удовлетворяют имена файлов, имеющие расширение .h. (Если шаблон не содержит символа `%', он должен в точности совпадать с именем искомого файла, а необходимость в этом

возникает достаточно редко.)

Специальное значение символа `%' в шаблоне директивы vpath может быть отменено с помощью предшествующего ему символа `\'. Специальное значение символа `\', предшествующего символу `%' может быть, в свою очередь, отменено добавлением еще одного символа `\' (строка "\\%" будет интерпретироваться как два символа. Первый из них - символ `\', второй - символ `%', который будет интерпретироваться как шаблонный). Символы `\', имеющие специальное значение, удаляются из шаблона перед тем, как он будет сравниваться с именами файлов. Символы `\' не имеющие специального значения, остаются в шаблоне.

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.

Например, запись

```
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.

Вот алгоритм, который использует `make` при решении вопроса о том, следует ли оставлять или отбрасывать найденный в процессе поиска по каталогам путь:

1. Если целевой файл не может быть найден в директории, которая указана в make-файле, проводится его поиск по каталогам.
2. Если поиск завершился успешно, найденный путь запоминается и искомый целевой файл временно помечается как найденная цель.
3. Используя этот же метод, проверяются все пререквизиты данной цели.
4. После обработки всех пререквизитов, для цели, возможно потребуется ее обновление:
    1. Если цель *не* нуждается в обновлении, директория, найденная в процессе поиска по каталогам, используется для всех пререквизитов этой цели. Иначе говоря, если цель не нуждается в обновлении, то используется директория, найденная в процессе поиска по каталогам.
    2. Если цель *нуждается* в обновлении (является устаревшей), то найденный в процессе поиска по каталогам путь *отбрасывается*, и цель обновляется с использовании только ее имени, указанном в make-файле. Другими словами, если `make` должна обновить цель, то эта цель строится или обновляется "локально", а не в той директории, которая была найдена во время поиска по каталогам.

Хотя этот алгоритм и выглядит сложным, на практике, он, как правило, дает именно тот результат, который вы и ожидали.

Другии версии `make` используют более простой алгоритм: если файла нет в текущей директории, но он был найден в процесс поиска по каталогам, установленный таким образом путь к этому файлу используется всегда, независимо от того, нуждается ли цель в обновлении или нет. Таким образом, при обновлении цели, новая версия файла будет расположена по тому же пути, где была найдена и "старая" версия.

Если для вас желательно именно такое поведение `make` по отношению к некоторым (или всем) каталогам, вы можете использовать переменную `GPATH`.

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**, таких как `$^' (смотрите раздел Автоматические переменные). Например, значением переменной `$^' является список всех пререквизитов правила с именами каталогов, где эти пререквизиты были найдены, а значением переменной `$@' является имя цели. Например:

```
foo.o : foo.c
        cc -c $(CFLAGS) $^ -o $@
```

(Переменная CFLAGS используется для того, чтобы иметь возможность указать опции компиляции исходных текстов на Си для неявных правил; в данном случае мы использовали ее просто в целях "унификации" процесса компиляции (смотрите раздел Используемые в неявных правилах переменные).

Часто, в список пререквизитов попадают файлы, которые не нужно передавать в исполняемую команду (например, заголовочные файлы). В такой ситуации можно использовать автоматическую переменную `$<', которая содержит лишь первый пререквизит правила:

```
VPATH = src:../headers
foo.o : foo.c defs.h hack.h
        cc -c $(CFLAGS) $< -o $@
```

Поиск в каталогах и неявные правила

Поиск в каталогах, указанных с помощью VPATH или vpath происходит также и при использовании неявных правил (смотрите раздел Использование неявных правил).

Например, если для файла `foo.o' не имеется явных правил, то make пробует использовать имеющиеся у нее неявные правила, в частности, правило, говорящее что для получения `foo.o', надо скомпилировать файл `foo.c' (если, конечно, такой файл существует). Если искомого файла нет в текущей директории, то make предпринимает его поиск по каталогам. Если файл `foo.c' будет найден в каком-либо из каталогов (или упомянут в make-файле), то к нему будет применено соответствующее неявное правило для компиляции программ на языке Си.

Командам из неявных правил "по необходимости" приходится пользоваться автоматическими переменными, следовательно они будут использовать имена файлов, полученных в результате поиска по каталогам, без каких-либо дополнительных усилий с вашей стороны.

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 `-l*имя*'. (Возможно, вам это покажется странным, поскольку пререквизит, обычно, является именем файла, а файл библиотеки *имя*, как правило, называется

`lib*имя*.a', а не `-l*имя*'.)

Когда имя пререквизита имеет форму `-l*имя*', make обрабатывает ее специальным образом, производя поиск файла с именем `lib*имя*.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 `lib*имя*.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 `lib*имя*.so' и `lib*имя*.a', это поведение может быть изменено с помощью переменной .LIBPATTERNS. Каждое слово в значении этой переменной рассматривается как шаблонная строка. Встретив пререквизит вида `-l*имя*', 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) - это 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.

Если вы напишите правило, которое не будет порождать указанный в нем целевой файл, то команды этого правила будут выполняться всякий раз при попытке достижения цели правила. Например:

```
clean:
        rm *.o temp
```

Поскольку исполнение команды `rm` не приводит к созданию файла `` `clean' ``, такой файл, скорее всего, вообще не будет существовать. В таком случае, команда `rm` будет выполняться всякий раз, когда вы скажете `` `make clean' ``.

Однако, 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` (смотрите раздел Имена специальных целей). В нашем примере достаточно записать:

```
.PHONY : clean
```

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 (смотрите раздел Использование неявных правил). В результате, использование абстрактных целей может ускорить обработку make-файла.

Таким образом, сначала должна идти строка, объявляющая `clean` абстрактной целью, а затем уже следует правило, описывающее эту цель:

```
.PHONY: clean
clean:
        rm *.o temp
```

Следующий пример демонстрирует полезность использования абстрактных целей при рекурсивном вызове `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 (смотрите раздел Аргументы для задания главной цели).

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.

Например:

```
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' ``. Смотрите раздел Абстрактные цели.

Использование пустых целей (empty target files) для фиксации событий

Пустая цель (empty target) является вариантом абстрактной цели. Такие цели используются для хранения команд, исполняющих действие, выполнение которого вам может иногда потребоваться. В отличие от астрактных 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 модефикации), 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. Например:

```
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 (смотрите раздел Автоматические переменные).

Имена специальных целей

Некоторые имена имеют специальное значение, когда используются в качестве 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 модефицирован. Смотрите раздел Абстрактные цели.

.SUFFIXES
: Пререквизиты специальной цели .SUFFIXES представляют собой список суффиксов (расширений) имен файлов, которые будут использоваться при поиске суффиксных правил. Смотрите раздел Устаревшие суффиксные правила.

.DEFAULT
: Команды, определенные для цели .DEFAULT, будут использованы со всеми целями make-файла, для которых не найдено ни явных, ни неявных правил. Смотрите раздел Определение правил "последнего шанса". Команды, определенные для .DEFAULT, будут использованы для всех пререквизитов, не являющихся целями каких-либо правил. Смотрите раздел Алгоритм поиска неявных 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. Смотрите раздел Прерывание или принудительное завершение make. Также, если цель является "промежуточным" файлов, он не будет, как обычно, удаляться после того, как необходимость в нем отпала. Смотрите раздел "Цепочки" неявных правил. 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. Смотрите раздел "Цепочки" неявных правил. .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. Смотрите раздел "Цепочки" неявных правил. .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 модефицированы, а обновляющая ее команда завершилась с ненулевым кодом возврата; аналогично, цель будет удаляться при прерывании работы make. Смотрите раздел Ошибки при исполнении команд.

.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. Смотрите раздел Ошибки при исполнении команд.

.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. Смотрите раздел Отображение исполняемых команд. Временно подавить печать исполняемых команд можно, запуская make с опциями `-s' или `--silent' (смотрите раздел Обзор опций).

.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. Смотрите раздел Связь с make "нижнего уровня" через переменные.

.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. В принципе, любое имя могло бы, таким образом, стать специальной целью, если разбить его на две части и добавить обе из них к списку суффиксов. На практике, суффиксы обычно начинаются с символа `.', поэтому такие специальные цели также начинаются с `.'. Смотрите раздел Устаревшие суффиксные правила.

Правила с несколькими целями

Правило с несколькими целями, эквивалентно нескольким правилам с одной целью, которые, за исключением имени цели, полностью идентичны друг другу. Ко всем целям будет применяться один и тот же набор команд, однако, эффект от их исполнения может быть разным, поскольку они могут ссылаться на имя обрабатываемой в данный момент цели, используя автоматическую переменную `$@'. А также, все цели подобного правила имеют один и тот же список пререквизитов.

Это может быть полезно в двух случаях.

- Вам нужны только пререквизиты, а не команды. Например, строка:

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

  объявляет дополнительный пререквизит для каждого из трех указанных объектных файлов.
- Сходные команды используются для обновления всех целей. Эти команды не обязаны быть абсолютно идентичными, поскольку, для подстановки конкретного имени цели, может быть использована автоматическая переменная `$@' (смотрите раздел Автоматические переменные). Например:

  ```
  bigoutput littleoutput : text.g
          generate text.g -$(subst output,,$@) > $@
  ```

  эквивалентно

  ```
  bigoutput : text.g
          generate text.g -big > bigoutput
  littleoutput : text.g
          generate text.g -little > littleoutput
  ```

  Здесь, мы предполагаем, что гипотетическая программа `generate` может генерировать выходную информацию двух видов, переключаясь с помощью опций `-big' и `-little'. Работа функции `subst` описана в разделе Функции анализа и подстановки строк.

Предположим, вы хотели бы менять список пререквизитов для конкретной цели, подобно тому, как переменная `$@' позволяет вам варьировать исполняемые команды. Этого невозможно добиться при помощи обычного правила с несколькими целями, однако вы можете это сделать, используя **статические шаблонные правила**. Смотрите раздел Статические шаблонные правила .

Несколько правил с одной целью

Один и тот же файл может являться целью нескольких правил. Все пререквизиты такой цели, перечисленные в разных правилах, объединяются в один общий список ее пререквизитов. Команды для обновления цели, будут выполняться в том случае, если хотя бы один пререквизит из любого правила окажется "более новым", чем эта цель.

Для одной цели может быть исполнен только один набор команд. Если команды для обновления цели указаны сразу в нескольких правилах, make выполнит только последний встретившийся набор команд и выдаст сообщение об ошибке. (В специальном случае, когда имя целевого файла начинается с точки, сообщение об ошибке не выдается. Такое странное поведение сохранено только для совместимости с другими реализациями make). У вас нет причин писать свои make-файлы таким странным образом, поэтому вы получите сообщение об ошибке.

Дополнительное правило, содержащее только пререквизиты, может быть использовано для "быстрого" добавления нескольких дополнительных пререквизитов одновременно ко многим файлам. Например, в make-файле обычно имеется переменная objects, содержащая список всех объектных файлов собираемой программы. 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 (смотрите раздел "Перекрытие" переменных). В следующем примере,

```
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 (смотрите раздел Использование неявных правил).

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 не обязательно *идентичными*.

Синтаксис статических шаблонных правил

Для статических шаблонных правил используется следующий синтаксис:

```
цели ...: шаблон-цели: шаблоны-пререквизитов ...
        команды
        ...
```

В списке *целей* перечисляются цели, к которым будет применяться данное правило. Так же, как и в обычных правилах, при задании имен целей могут использоваться шаблонные символы (смотрите раздел Использование шаблонных символов в именах файлов).

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.

Специальное значение символа `%' в шаблоне может быть отменено с

помощью предшествующего ему символа `\'. Специальное значение символа `\', предшествующего символу `%', может быть, в свою очередь, отменено добавлением еще одного символа `\' (строка \\% будет интерпретироваться как два символа. Первый из них - символ `\', второй - символ `%', который будет интерпретироваться как шаблонный). Символы `\', имеющие специальное значение, удаляются из шаблона перед тем, как он будет сравниваться с именами файлов. Символы `\', которые не могут повлиять на интерпретацию `%', остаются в шаблоне. Например, шаблон `the\%weird\\%pattern\\' состоит из строки `the%weird\', за которой следуют шаблонный символ `%' и строка `pattern\\'. Последние два символа остаются без изменений, поскольку не могут повлиять на способ интерпретации какого-либо символа %.

Вот пример, где объектные файлы `foo.o' `bar.o' компилируются из соответствующих им исходных файлов с расширением `.c':

```
objects = foo.o bar.o

all: $(objects)

$(objects): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
```

В этом примере, автоматическая переменные `$<' и `$@' содержат, соответственно, имя пререквизита и имя цели (смотрите раздел Автоматические переменные).

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 filter (смотрите раздел Функции анализа и подстановки строк):

```
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'.

Сравнение статических шаблонных правил (static pattern

rules) и неявных правил (implicit rules)

Статические шаблонные правила имеют много общего с обычными шаблонными правилами (смотрите раздел Определение и переопределение шаблонных правил). Оба вида правил содержат шаблон для цели и шаблон для конструирования имен пререквизитов. Разница заключается в том, каким образом `make` принимает решение о необходимости применения данного правила.

Неявное правило *может* быть применено к любой цели, которая подходит под его шаблон, однако оно действительно *будет* применено только в том случае, когда цель не имеет команд, определенных иным способом и имеются все необходимые для этого пререквизиты. Если для одной цели могут быть применены сразу несколько неявных правил, будет использовано только одно из них; какое именно - будет зависеть от порядка их следования.

Статическое шаблонное правило, напротив, применяется к точно указанному набору целей. Оно не может быть применено к каким-либо другим целям и одинаковым образом применяется ко всем перечисленным в нем целям. Случай, когда к одной и той же цели могут быть применены два разных статических шаблонных правила, оба из которых имеют команды, считается ошибкой.

Статические шаблонные правила могут быть предпочтительней неявных правил по следующим причинам:

- Вы можете использовать статические шаблонные правила для тех файлов, чье имя синтаксически не подходит для использования в неявных правилах, но может быть явно указано в списке целей.
- Если вы не уверены, какие, в точности, файлы содержатся в вашей директории, вы не можете быть уверены в том, что какие-нибудь "ненужные" в данный момент файлы не повлияют на работу `make`. Может случится, что будут использованы не те неявные правила, на которые вы рассчитывали, поскольку их выбор будет зависеть от применяемой процедуры поиска. В случае статических шаблонных правил такой неоднозначности не существует: каждое правило будет применяться в точности к тем целям, которые были в нем указаны.

Правила с двойным двоеточием (double-colon rules)

Правила с двойным двоеточием (double-colon rules) представляют собой правила, записанные с помощью `::' вместо обычного `:' после имени цели. По сравнению с обычными правилами, они обрабатываюся по-другому в случаях, когда одна и та же цель указана сразу в нескольких правилах.

Когда цель указана в нескольких правилах, все они должны быть одного и того же типа: либо обычными правилами, либо правилами с двойным

двоеточием. Если все они являются правилами с двойным двоеточием, то каждое из них является независимым от других. Команды, указанные в правиле с двойным двоеточием, будут выполняться, если его цель окажется "старше", чем какой-либо из его пререквизитов. Результатом может быть выполнение всех или нескольких правил. Может получиться и так, что ни одно из правил не будет выполнено.

На самом деле, правила с двойным двоеточием полностью независимы друг от друга. Каждое из этих правил, обрабатывается индивидуально, как если бы они были обычными правилами с разными целями.

Правила с двойным двоеточием исполняются в том порядке, как они описаны в make-файле. Однако, в случаях, когда использование таких правил действительно необходимо, порядок исполнения команд, как правило, не имеет значения.

В типичных ситуациях, правила с двойным двоеточиям не слишком полезны. Они используются в тех случаях, когда способ обновления цели зависит от того, какой именно из пререквизитов вызвал необходимость ее обновления. А такие случаи встречаются нечасто.

Каждое правило с двойным двоеточием должно содержать команды, в противном случае `make` попытается применить подходящее неявное правило. Смотрите раздел Использование неявных правил.

Автоматическая генерация списка пререквизитов

Как правило, в типичном make-файле значительная часть правил служит лишь для того, чтобы описать зависимость объектных файлов от некоторых заголовочных файлов. Например, если `` `main.c' `` использует `` `defs.h' ``, включая его с помощью директивы `#include`, вы можете написать:

```
main.o: defs.h
```

Это правило необходимо для того, чтобы `make` обновляла объектный файл `` `main.o' `` при каждом изменении `` `defs.h' ``. Для большой программы, скорее всего, вам придется написать большое количество подобных правил. Далее, каждый раз при изменении исходных текстов путем добавления или удаления директивы `#include`, вам придется модефицировать соответствующим образом и make-файл.

Чтобы избежать подобных неудобств, большинство современных компиляторов могут автоматически создать для вас такие правила, просматривая содержимое исходных файлов, и учитывая встретившиеся в них директивы `#include`. Обычно, это делается при помощи опции компилятора `` `-M' ``. Например, результатом работы команды

```
cc -M main.c
```

будет строка:

```
main.o : main.c defs.h
```

Таким образом, вам больше не потребуется писать подобные правила "вручную" - эту работу возьмет на себя компилятор.

Обратите внимание, что в приведенной выше зависимости упоминается файл `main.o' и, следовательно, этот файл не сможет рассматриваться как промежуточный в процессе поиска неявных правил. Как следствие, make никогда не будет удалять этот файл после его использования; смотрите раздел "Цепочки" неявных правил.

При работе со старыми реализациями make, обычной практикой являлась генерация пререквизитов по отдельному запросу, наподобие `make depend'. Эта команда создаст файл `depend', содержащий все автоматически сгенерированные пререквизиты; затем они могут быть включены в make-файл с помощью директивы include (смотрите раздел Подключение других make-файлов).

Возможность автоматического обновления make-файлов, заложенная в GNU make, делает эту практику устаревшей --вам никогда не понадобится явно указыватть утилите make на необходимость обновления списка пререквизитов, поскольку она всегда обновляет любые используемые make-файлы, если они устарели. Смотрите раздел Автоматическое обновление make-файлов.

Мы рекомендуем вам использовать подход, когда для каждого исходного файла имеется свой маленький make-файл, содержащий список пререквизитов этого исходного файла. Для каждого исходного файла `*имя*.c' имеется make-файл `*имя*.d', в котором перечислен список файлов, от которых зависит объектный файл `*name*.o'. При таком подходе, новые списки пререквизитов могут строиться только для тех исходных файлов, которые действительно были модефицированы.

Вот пример шаблонного правила для генерации файлов пререквизитов (то есть make-файлов), имеющих имя `*имя*.d' из файлов с исходным текстом `*имя*.c':

```
%.d: %.c
        set -e; $(CC) -M $(CPPFLAGS) $< \
                  | sed 's/\($*\)\.o[ :]*/\1.o $@ : /g' > $@; \
                [ -s $@ ] || rm -f $@
```

Для получения подробной информации об определении шаблонных правил смотрите раздел Определение и переопределение шаблонных правил. Флажок `-e' заставляет интерпретатор командной строки немедленно завершать работу в случае, если при вызове $(CC) произойдет ошибка (команда возвратит отличный от нуля код завершения). Обычно, интерпретатор командной строки завершается с кодом возврата, полученным от последней выполненной команды из всей цепочки команд (sed в данном случае) и make может просто "не заметить" ошибочной ситуации, произошедшей при вызове компилятора.

При использовании компилятора GNU C, вместо опции `-M' вы можете

попробовать использовать опцию `-MM'. При этом, в список пререквизитов не будут попадать системные заголовочные файлы. Смотрите раздел `Options Controlling the Preprocessor' руководства по компилятору GNU C.

Назначение команды sed заключается в преобразовании (например) строки:

```
main.o : main.c defs.h
```

в строку:

```
main.o main.d : main.c defs.h
```

Таким образом, получается, что каждый файл с расширением `.d' зависит от всех тех же исходных и заголовочных файлов, что и соответствующий ему объектный файл `.o'. Теперь, make будет заново генерировать список пререквизитов при любом изменении исходного либо заголовочных файлов программы.

После того, как вы создали правило, обновляющее все `.d' файлы, надо сделать эти файлы доступными для make. Для этого используется директива include. Смотрите раздел Подключение других make-файлов. Например:

```
sources = foo.c bar.c

include $(sources:.c=.d)
```

(В этом примере, для преобразования списка исходных файлов `foo.c bar.c' в список пререквизитов (`foo.d bar.d'), используется техника ссылки на переменную с подстановкой. Смотрите раздел Ссылка с заменой). Поскольку файлы с расширением `.d' являются полноправными make-файлами, утилита make будет сама заботится об их своевременном обновлении, не требуя от вас каких-либо дополнительных усилий. Смотрите раздел Автоматическое обновление make-файлов.

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). Смотрите раздел Исполнение команд.

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. Текст до символа `#' не является частью комментария.

Отображение исполняемых команд (command echoing)

Обычно, make печатает каждую командную строку перед тем, как она будет выполнена. Мы называем такой механизм **эхом (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. Смотрите раздел Обзор опций. 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 (смотрите раздел Имена специальных целей). 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. (На практике, make может использовать некоторую оптимизацию, однако это не сказывается на конечном результате.)

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. Например:

```
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 '\'. Таким образом, предыдущий пример можно записать так:

```
foo : bar/lose
        cd bar;  \
        gobble lose > ../foo
```

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

При работе в операционной системе MS-DOS, если переменная SHELL не установлена, имя интерпретатора командной строки берется из переменной COMSPEC (которая установлена всегда).

При работе в операционной системе MS-DOS, обработка строк make-файла, изменяющих содержимое переменной SHELL, имеет некоторые особенности. Стандартный для MS-DOS интерпретатор командной строки `command.com' обладает столь ограниченными возможностями, что большое число пользователей make предпочитают заменить его на что-нибудь более приемлимое. Поэтому, работая под управлением MS-DOS, make отслеживает значение переменной SHELL, и меняет свое поведение в зависимости от того, указывает ли эта переменная на интерпретатор с ограниченными функциональными возможностями (в стиле MS-DOS), либо на "полнофункциональный" интерпретатор (в стиле Unix). Это позволяет утилите make сохранять приемлимую функциональность даже в случае, когда переменная SHELL указывает на командный интерпретатор `command.com'.

Если переменная SHELL указывает на интерпретатор командной строки, выдержанный в Unix-стиле, программа make, работающая под управлением MS-DOS, дополнительно проверяет - действительно ли указанный интерпретатор существует; если нет, make будет игнорировать строки make-файла, изменяющие значение переменной SHELL. Работая под управлением MS-DOS, утилита GNU make проводит поиск интерпретатора командной строки в следующих местах:

1. В точности том месте, куда указывает значение переменной SHELL. Например, если в make-файле указано `SHELL = /bin/sh', make будет искать

интерпретатор в каталоге `/bin' текущего диска.
2. В текущем каталоге.
3. В каждом из каталогов, перечисленных в переменной PATH (в том порядке, как они указаны).

В каждом из проверяемых каталогов make сначала пытается найти файл с известным ей конкретным именем (`sh' в приведенном выше примере). Если такого файла не существует, делается попытка найти файл с подобным именем, но имеющим другое расширение. При этом, проверяются нзвестные расширения, используемые для исполняемых файлов ( `.exe', `.com', `.bat', `.btm', `.sh' и некоторые другие).

При успешном завершении любой из этих попыток, в переменную SHELL записывается полное имя (с путем) найденного интерпретатора командной строки. При неудачном завершении всех попыток найти командный интерпретатор, присваивание переменной SHELL нового значения игнорируется и ее содержимое не изменяется. Таким образом, make будет поддерживать специфических для оболочек Unix-подобного стиля возможности, только в том случае, если подходящая оболочка действительно имеется в системе, где была запущена make.

Обратите внимание, что описанная выше процедура поиска командного интерпретатора выполняется только в тех случаях, когда переменная SHELL устанавливается из из make-файла. Если значение этой переменной берется из среды (environment) или устанавливается с помощью командной строки, ожидается что она будет содержать полное имя (с путем) интерпретатора командной строки, подобно тому, как это происходит в Unix.

Эффект от подобной специфической для MS-DOS обработки, состоит в том, что строка make-файла `SHELL = /bin/sh' (встречающаяся во многих make-файлах, ориентированных на Unix), без именений будет работать в среде MS-DOS - достаточно лишь поместить (например) программу `sh.exe' в какой-нибудь из каталогов, перечисленных в PATH.

В отличие от большинства переменных, SHELL никогда не получает свое значение из среды (environment). Это делается потому, что значение переменной SHELL используется для указания командного интерпретатора, который выбран вами для интерактивного использования. Было бы неправильно, если бы ваш персональный выбор отражался бы на работоспособности make-файлов. Смотрите раздел Переменные из операционного окружения. Переменная среды SHELL **используется**, однако, при работе make в операционных системах MS-DOS и MS-Windows, поскольку, как правило, пользователи, работающие в этих системах, не используют переменную SHELL для каких-либо иных целей, кроме работы с make. Если, при работе в MS-DOS, вы по какой-либо причине не можете воспользоваться переменной SHELL, вместо нее вы можете использовать переменную MAKESHELL. При наличии переменой MAKESHELL, значение переменной SHELL не учитывается.

Параллельное исполнение команд

GNU make умеет одновременно выполнять несколько команд. Обычно, make выполняет команды поочередно, ожидая завершения очередной команды, перед тем, как выполнять следующую. Однако, с помощью опций `-j' и `--jobs' можно заставить make выполнять одновременно несколько команд.

В операционной системе MS-DOS, опция `-j' не работает, поскольку эта система не поддерживает многозадачность.

За опцией `-j' может следовать целое число, которое будет указывать количество одновременно исполняемых команд (это понятие называется числом **слотов задания (job slots)**). В случае, если после опции `-j' не указано конкретное число слотов задания, подразумевается их неограниченное количество. По умолчанию, количество слотов задания равняется единице, что, фактически, означает последовательное выполнение (одновременно может выполняться только одна команда).

Одно из неприятных последствий одновременной работы нескольких команд, заключается в возможности "перемешивания" их сообщений, поскольку нескольким командам может "одновременно" понадобится вывести некоторую информацию.

Другая проблема заключается в том, что два разных процесса не могут одновременно читать данные из одного устройства. Дабы иметь уверенность в том, что одновременно только одна команда сможет попытаться прочесть входные данные с терминала, make закрывает стандартные потоки ввода у всех запущенных команд, кроме одной. Это означает, что попытка запущенной команды прочитать данные из стандартного ввода обычно заканчивается фатальной ошибкой (например, получением сигнала `Broken pipe').

Невозможно заранее предсказать, какая именно команда получит в качестве стандартного ввода работающий входной поток (подключенный к терминалу или другому источнику, куда был перенаправлен стандартный ввод утилиты make). Сначала этот входной поток получит в свое распоряжение первая запущенная команда, затем первая команда, которая была запущена по окончанию работы этой команды и так далее.

Возможно, в будущем, если мы найдем лучшую альтернативу, подобное поведение make будет изменено. Пока же, вам не следует использовать команды, работающие со стандартным вводом, если вы используете механизм параллельного исполнения команд. Если вы не пользуетесь подобной возможностью, во всех командах стандартный ввод будет работать нормально.

Наконец, параллельное исполнение команд имеет некоторые особенности при рекурсивной работе make. Подробно это обсуждается в разделе Передача опций в make "нижнего уровня".

При аварийном завершении команды (от нее был получен ненулевой код возврата либо она была прервана полученным сигналом) для которой не был указан режим игнорирования ошибок (смотрите раздел Ошибки при

исполнении команд), оставшиеся командные строки (обновляющие ту же цель) не будут выполняться. При этом, если в командной строке не были указаны опции `-k' или `--keep-going' (смотрите раздел Обзор опций), программа make аварийно завершается. При любом аварийном завершении (в том числе и из-за получения сигнала), перед тем как закончить работу, программа make дожидается завершения всех своих дочерних процессов.

Если ваша компьютерная система сильно загружена, вам возможно захочется, чтобы make одновременно запускала меньшее число заданий, чем это делается при нормальной загрузке. В таком случае можно использовать опцию `-l' для того, чтобы поставить число одновременно выполняемых команд в зависимость от средней загрузки системы. Для этого, за опцией `-l' или `--max-load' должно следовать число с плавающей точкой. Например в следующем случае:

```
-l 2.5
```

make будет запускать одновременно не более одной команды, если средняя загрузка системы будет больше чем 2.5. Опция `-l', за которой не следует число, отменяет установленное ранее ограничение на максимальную загрузку систему.

При наличии ограничения на загрузку системы, make поступает следующим образом. Перед тем как запустить новое задание, если, по крайней мере, одно задание уже работает, make проверяет среднюю загрузку системы. Если она превышает установленный предел, make ожидает, пока загрузка не упадет до нужного уровня либо все прочие задания завершатся.

По умолчанию, загрузка системы не ограничивается.

Ошибки при исполнении команд

После завершения очередной команды, make проверяет полученный от нее код возврата. В случае успешного ее завершения, выполняется (своим экземпляром командного интерпретатора) 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. Например, вы можете использовать команду mkdir, дабы быть уверенным в существовании некоторого 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.

В следующем примере:

```
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 изпользовании специальной цели `.IGNORE` в правиле, не имеющем пререквизитов. Вместо подобной устаревшей практики, мы рекомендуем применять более гибкую методику с использованием `-'.

Когда произошедшая ошибка игнорируется (например, вследствие использования `-' или опции `-i'), ошибочное завершение команды обрабатывается аналогично нормальному завершению, за исключением того, что при этом 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. Соответственно, никакие команды, обновляющие эти цели, более 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. Смотрите раздел Обзор опций. 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 собщить о происшедшей ошибке. Напротив, задание опции `-k' означает, что ваша цель - наболее полно протестировать процесс сборки программы, по возможности обнаружив максимальное количество проблем. После устранения всех найденных проблем, можно будет заново повторить процесс компиляции. Подобными соображениями, например, руководствуется редактор Emacs, по умолчанию запуская `make` с опцией `-k', при выполнении команды `compile`.

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 модефикации 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 (смотрите раздел Прерывание или принудительное завершение make. 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" соображений). Для того, чтобы make автоматически удаляла некорректно построенные целевые файлы, вы должны явно этого потребовать.

Прерывание (interrupting) или принудительное завершение (killing) make

Если при выполнении какой-либо команды, программа make получит прерывающий ее сигнал, она может удалить целевой файл, который предполагалось обновить с помощью этой команды. Файл будет удален в том случае, если время его последней модефикации изменилось с тех пор, как make проверяла его впервые.

Смысл удаления целевого файла заключается в том, чтобы при следующем запуске make, он был построен заново. Для чего это делается? Предположим, что во время работы компилятора, в то самое время как он начал записывать на диск объектный файл `foo.o', вы нажали Ctrl-c. Нажатие Ctrl-c немедленно прервет работу компилятора, в результате чего на диске останется фрагмент не полностью записанного файла с более поздним временем модефикации, чем исходный файл `foo.c'. К счастью, make также получит сигнал Ctrl-c и удалит этот "не доделанный" файл. Если бы make этого не сделала, при следующем вызове она могла бы подумать, что уже имеется "свежая" версия файла `foo.o', и он больше не нуждается в обновлении - результатом был бы странный сбой в работе компоновщика, попытающегося скомпоновать поврежденный объектный файл.

Вы можете предотвратить удаление целевого файла в подобной ситуации, сделав его пререквизитом специальной цели .PRECIOUS. Перед тем, как обновить цель, make проверяет - не является ли она пререквизитом цели .PRECIOUS, и на основании этого решает - нужно ли удалять ее при получении сигнала или нет. В некоторых ситуациях вам может потребоваться, чтобы при возникновении сигнала, цель, тем не менее, не удалялась. Например, если цель служит только для запоминания времени последней модефикации (и ее содержимое не имеет значия), или, по каким-либо соображениям, она должна существовать всегда, или же процесс ее обновления является "атомарной" операцией.

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 (смотрите раздел Обзор опций):

```
subsystem:
        $(MAKE) -C subdir
```

Разумеется, вы может просто взять и использовать приведенные выше примеры в своих make-файлах, однако, для понимания механизма рекурсивного вызова make, вам необходимо знать множество вещей, в том числе, каким образом make "верхнего уровня" взаимодействует с рекурсивно вызванными копиями make.

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.

Использование переменной MAKE оказывает влияние на работу опций `-t' (`--touch'), `-n' (`--just-print') и `-q' (`--question'). Командная строка, содержащая переменную MAKE, работает так, как если бы в ее начале находился специальный символ `+' (смотрите раздел Вместо исполнения команд).

Предположим, в предыдущем примере make была вызвана следующим

образом: `make -t'. (Опция `-t' заставляет make пометить все цели как "обновленные", не выполняя в действительности каких-либо команд; смотрите раздел Вместо исполнения команд.) Следуя тому, каким образом обычно описывается поведение опции `-t', команда `make -t' создала бы файл с именем `subsystem' и на этом завершила бы свою работу. Скорее всего, однако, вы хотели бы получить другой результат, а именно, запуск команды `cd subdir && make -t'. Но для достижения такого результата потребовалось бы выполнение команды, в то время как опция `-t' говорит, что команды выполняться не должны.

Для получения желаемого результата, make обрабатывает строки с командами, содержащие ссылку на переменную MAKE специальным образом: опции `-t', `-n' и `-q' на такие строки не действуют. Командные строки с переменной MAKE выполняются обычными образом, не обращая внимания на опции, отключающие выполнение команд. Для передачи параметров от make "высшего" уровня на "нижние" уровни используется обычный механизм - переменная MAKEFLAGS (смотрите раздел Передача опций в make "нижнего уровня"). Таким образом, все ваши запросы на обновление даты модефикации 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'; смотрите раздел Обзор опций).

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. Смотрите раздел Переменные из операционного окружения.

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 (смотрите следующий раздел).

Переменные, которые по умолчанию созданы самой make, *не* передаются "вниз" (смотрите раздел Используемые в неявных правилах переменные). Такие переменные каждая make "нижнего уровня", при необходимости, создаст самостоятельно.

Для экспорта указанной переменной в make "нижнего уровня", используется директива export:

```
export переменная ...
```

Для *запрета* экспорта переменной, используется директива unexport:

```
unexport переменная ...
```

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
export переменная
```

Аналогично,

```
export переменная += значение
```

эквивалентно:

```
переменная += значение
export переменная
```

Смотрите раздел Добавление текста к переменной.

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:

```
export
```

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 (смотрите раздел Условные части make-файла); 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. Смотрите раздел Переменная MAKEFILES.

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). Смотрите раздел Обзор опций.

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). Смотрите раздел "Перекрытие" переменных.

Опции `-C', `-f', `-o', и `-W' не записываются в переменную MAKEFLAGS и, соответственно, не передаются в "порожденные" копии make.

Опция `-j' обрабатывается специальным образом (смотрите раздел Параллельное исполнение команд). 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 (смотрите раздел Вместо исполнения команд), 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, вместо указанного вами числового значения, 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 строки, 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' (смотрите раздел Обзор опций), 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 начинатеся 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', следующая строка:

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

будет выведена перед тем, как make начнет что-либо делать, а строка:

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

будет выведена перед тем, как работа будет закончена.

Как правило, вам не придется самостоятельно указывать эти опции, 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.

Вот пример определения именованной командной последовательности:

```
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. Смотрите раздел Многострочные переменные, где описана работа директивы define.

В этом примере, 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.

Для того, чтобы задействовать именованную последовательность команд, подставьте переменную с этой последовательностью в качестве команды правила (подстановка делается "обычным" для всех переменных способом; смотрите раздел Обращение к переменным). Переменные, определенные с помощью define являются рекурсивно вычисляемыми, поэтому все все ссылки на переменные, находящиеся внутри директивы define, будут при этом вычислены. Так, в следующем примере:

```
foo.c : foo.y
        $(run-yacc)
```

при вычислении значения переменной run-yacc, вместо `$^' будет подставлено имя `foo.y', и имя `foo.c' вместо `$@'.

Это достаточно реалистичный пример, однако, на практике, в данном конкретном правиле нет необходимости, поскольку make имеет соответствующие неявные правила, имеющие аналогичный эффект (смотрите раздел Использование неявных правил).

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 `+'. Смотрите раздел Написание команд. Например, при использовании следующей командной последовательности:

```
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. (Смотрите раздел Отображение исполняемых команд, где обсуждается использование специального символа `@'.)

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. Например:

```
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.

Если вам непонятно, зачем может понадобиться пустая команда, вспомните о наличии неявных правил. Правило с пустой командой может предовратить применение для указанной цели команд, определенных в неявных правилах (или правилах со специальной целью .DEFAULT); смотрите разделы Использование неявных правил и Определение правил "последнего шанса").

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. Это, однако, 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 (смотрите раздел Абстрактные цели).

Использование переменных (variables)

Переменная (variable) представляет собой имя, определенное в make-файле для представления строки текста, называемой **значением** переменной. Далее, по вашему запросу, эти значения могут быть подставлены в нужные места make-файла (например, в имена целей, имена пререквизитов, команды и так далее). В некоторых других версиях make, переменные называются **макросами (macros)**.

За исключением нескольких мест, 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. Однако, мы рекомендуем 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`. (смотрите раздел Связь с make "нижнего уровня" через переменные).

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. Мы, однако, рекомендуем вам пользоваться нижним регистром для всех переменных, используемых для "внутренних нужд" make-файла. Верхний же регистр мы рекомендуем оставить для переменных, влияющих на работу неявных правил или содержащих параметры, которые пользователь может переопределять (смотрите раздел "Перекрытие" переменных).

Некоторые переменные имеют имена, состоящие лишь из одного или нескольких символов пунктуации. Это, так называемые, **автоматические переменные**; они используются специальным образом. Смотрите раздел Автоматические переменные.

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 `$$'.

Ссылка на переменную может быть использована практически в любом контексте: в качестве цели, пререквизита, команды. Она может использоваться в большинстве директив, а также выступать в качестве нового значения другой переменной. Вот типичный пример, где переменная используется для хранения списка объектных файлов программы:

```
objects = program.o foo.o utils.o
program : $(objects)
        cc -o program $(objects)

$(objects) : defs.h
```

Ссылки на переменную обрабатываются при помощи простой текстовой

подстановки. Таким образом, правило, описывающее процесс компиляции программы из исходного файла `prog.c', могло бы выглядеть так:

```
foo = c
prog.o : prog.$(foo)
        $(foo)$(foo) -$(foo) prog.$(foo)
```

Пробелы, находящиеся в операторе присваивания перед новым значеним переменной, игнорируются, поэтому переменная foo будет содержать строку `c'. (Не следует, однако, принимать этот пример "всерьез" и писать подобным образом свои make-файлы!)

Если за знаком доллара следует символ, отличный от доллара, открывающейся круглой скобки или открывающейся фигурной скобки, то этот одиночный символ рассматривается как имя переменной. Таким образом, вы можете обратиться к переменной x, используя запись `$x'. Однако, такая практика крайне нежелательна, за исключением случаев обращения к автоматическим переменным (смотрите раздел Автоматические переменные).

Две разновидности (flavors) переменных

В GNU make есть два способа, с помощью которых переменные могут получить свое значение. Мы называем это двумя разновидностями (flavors) переменных. Две разновидности отличаются тем, каким образом переменная была определена и что происходит при вычислении ее значения.

Первая разновидность - это **рекурсивно вычисляемые (recursively expanded)** переменные. Такие переменные определяются с помощью `=' (смотрите раздел Установка значения переменной) или директивы define (смотрите раздел Многострочные переменные). 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)**.

Например, следущий make-файл:

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:;echo $(foo)
```

выдаст на экран `Huh?': при вычислении ссылки `$(foo)', она будет заменена на `$(bar)', которая, свою очередь, заменена на `$(ugh)', которая , наконец, будет расширена в `Huh?'.

Эта разновидность переменных - единственная, поддерживаемая другими версиями make. Она имеется свои достоинства и недостатки. Ее

преимущество (по мнению большинства) заключается в том, что, например, следующий фрагмент:

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

будет работать так, как и ожидалось: 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. (На самом деле, make распознает ситуацию зацикливания и сообщает об ошибке.)

Другой недостаток рекурсивно вычисляемых переменных состоит в том, что все функции, на которые они ссылаются (смотрите раздел Функции преобразования текста) будут вычисляться заново при каждой "подстановке" этой переменной. Работа make при этом, разумеется, замедляется. Но еще хуже то, что результат выполнения функций wildcard и shell становится труднопредсказуемым, поскольку сложно в точности сказать, когда и сколько раз эти функции будут выполнены.

Подобных недостатков лишена другая разновидность переменных - упрощенно вычисляемые переменные.

Упрощенно вычисляемые (simply expanded) переменные определяются с помощью `:=' (смотрите раздел Установка значения переменной). 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).

Вот чуть более сложный пример, иллюстрирующей использование `:=' совместно с функцией shell. (Смотрите раздел Функция shell.) Этот пример также демонстрирует использование переменной MAKELEVEL, которая изменяется во время переходов "с уровня на уровень" при рекурсивном использовании make. Смотрите раздел Связь с make "нижнего уровня" через переменные.

```
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 (смотрите раздел Функции преобразования текста).

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. Поскольку конечные пробелы *не* удаляются из значения переменной, единственного пробела в конце строки было бы достаточно для получения нужного эффекта (но такая запись была бы менее понятна). 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 *не* желательно наличие пробелов в конце значения переменной, помните, что 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 (смотрите раздел Функция `origin`):

```
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.

"Расширенные" способы обращения к переменным

В этом разделе обсуждаются некоторые дополнительные возможности, которые можно использовать в ссылках на переменные для придания им большей гибкости.

Ссылка с заменой (substitution reference)

При использовании **ссылки с заменой (substitution reference)**, вместо нее подставляется значение переменной, модефицированное указанным вами способом. Такая ссылка имеет форму `$(*переменная*:*a*=*b*)' (или `${*переменная*:*a*=*b*}'). Это значит, что должно быть взято значение переменной *переменная*, и каждая найденная в нем цепочка символов *a*, находящаяся в конце слова, должна быть заменена на цепочку символов *b*.

Говоря "находящаяся в конце слова", мы имеем ввиду, что 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. В следующем примере:

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

переменная `bar' получит значение `a.c b.c c.c'. Смотрите раздел Установка значения переменной.

На самом деле, ссылка с заменой представляет собой "укороченый" вариант использования функции `patsubst` (смотрите раздел Функции анализа и подстановки строк). 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*))'. Смотрите раздел Функции анализа и подстановки строк, где описана работа функции `patsubst`. В следующем примере:

```
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 лишь 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)**. В следущем примере:

```
x = y
y = z
a := $($(x))
```

переменная a получит значение `z': ссылка `$(x)' внутри выражения `$($(x))' будет заменена на `y', так что строка `$($(x))' будет преобразована в `$(y)' и, далее, расширена в `z'. Здесь имя переменной не указано "прямо", а вычисляется при расширении выражения `$(x)'. Ссылка `$(x)' здесь находится внутри другой ссылки на переменную.

Предыдущий пример демонстрировал два "уровня вложенности". На самом деле, число таких уровней может быть неограниченным. Следующий пример демонстрирует три "уровня вложенности":

```
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. В следующем примере:

```
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 (смотрите раздел Функции преобразования текста). Вот пример, в котором используется функция subst (смотрите раздел Функции анализа и подстановки строк):

```
x = variable1
variable2 := Hello
y = $(subst 1,2,$(x))
z = y
a := $($($(z)))
```

Здесь, переменная a получает значение `Hello'. Вряд ли кто-нибудь будет писать подобным образом, однако, это пример действительно работает: 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 значеним является строка `Hello'.

Вычисляемое имя переменной совсем не обязательно 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. В следующем примере:

```
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** (смотрите раздел Две разновидности переменных), - это разные концепции, хотя при написаниии "нетривиальных" make-файлов, они часто используются совместно.

Как переменные получают свои значения

Переменные могут получать свои значения разными путями:

- При запуске `make`, в командной строке вы можете задать значение переменной, которое "перекроет" значение, устанавливаемое в make-файле. Смотрите раздел "Перекрытие" переменных.
- Вы можете задать значение переменной внутри make-файла с помощью оператора присваивания (смотрите раздел Установка значения переменной) или с помощью директивы `define` (смотрите раздел Многострочные переменные).
- Переменные среды (environment variables) автоматически становятся переменными `make`. Смотрите раздел Переменные из операционного окружения.

- Несколько переменных, называемых **автоматическими**, "автоматически" получают новое значение при обработке каждого правила. Каждая из таких переменных предназначена для какого-то конкретного применения. Смотрите раздел Автоматические переменные.
- Некоторые переменные имеют "фиксированное" начальное значение. Смотрите раздел Используемые в неявных правилах переменные.

Установка значения переменной

Для установки значения переменной внутри make-файла, используется строка, начинающаяся с имени переменной, за которым следует `=' или `:='. Все, что в этой строке следует за символом `=' или `:=', становится значением переменной. В следующем примере:

```
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). Смотрите раздел Две разновидности переменных.

Имена переменных могут содержать ссылки на другие переменные и функции, которые будут вычислены (во время чтения make-файла) для получения действительного имени переменной.

На длину строки, являющейся значением переменной, не накладывается каких-либо ограничений (за исключением доступного программе объема памяти). "Хорошей идеей" является разбиение длинного определения переменной на несколько отдельных строк с помощью символа '\', за которым следует символ перевода строки. Такое разбиение никак не отразится на работе make, но 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; смотрите раздел Используемые в неявных правилах переменные). Некоторые специальные переменные автоматически получают новое значение при обработке каждого правила; они называются **автоматическими** переменными (смотрите раздел 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 (смотрите раздел Функция origin):

```
FOO ?= bar
```

и

```
ifeq ($(origin FOO), undefined)
FOO = bar
endif
```

Добавление текста к переменной

Часто возникает необходимость добавить некоторый текст к значению уже определенной переменной. Вы можете это сделать с помощью оператора `+=', например:

```
objects += another.o
```

Здесь, к значению переменной objects добавляется текст `another.o' (предваренный одиночным пробелом). Так, в результате выполнения фрагмента:

```
objects = main.o foo.o bar.o utils.o
objects += another.o
```

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 (смотрите раздел Две разновидности переменных).

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 `:='; смотрите раздел Установка значения переменной, где описана работа оператора `:='). Таким образом, фрагмент

```
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. Вспомните, что 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" (смотрите раздел Две разновидности переменных). Таким образом, при использовании оператора `+=' с рекурсивно вычисляемой переменной, указанный вами текст добавляется к "нераскрытому" значению, которое хранится внутри переменной. Поэтому, следующий фрагмент:

```
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; смотрите раздел Перечень имеющихся неявных правил.) Использование оператора `=' определяет переменную CFLAGS как рекурсивно вычисляемую, и, значит, выражение `$(includes) -O' *не* будет вычисляться вв момент определения этой переменной. Таким образом, переменная includes совсем не обязана 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 (смотрите раздел "Перекрытие" переменных), 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
```

Смотрите раздел Добавление текста к переменной.

Разумеется, директива `override` была придумана не для "экскалации войны" между make-файлом и параметрами командной строки. Идея заключалась в том, 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. Это можно сделать с помощью директивы `override`:

```
override CFLAGS += -g
```

Директиву `override` можно также использовать совместно с директивой `define`. Выглядит это аналогично:

```
override define foo
bar
endef
```

Смотрите следующий раздел, где описана работы директивы `define`.

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 (смотрите раздел 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 (смотрите раздел Две разновидности переменных). 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. Смотрите раздел Исполнение команд.

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
```

Смотрите раздел Директива `override`.

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. Смотрите раздел Обзор опций. 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, стропроцентной 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 (смотрите раздел Рекурсивный вызов make). 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. Смотрите раздел Связь с make "нижнего уровня" через переменные, где обсуждается этот вопрос.

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.

Скорее всего, такие проблемы возникли бы и с переменной SHELL, которая обычно присутствует в операционной среде для указания выбранной пользователем командной оболочки. Было бы очень нежелательно, чтобы этот выбор пользователя влиял на работу make. Поэтому make игнорирует значение переменной среды SHELL (за исключением случаев, когда она работает в операционных системах MS-DOS и MS-Windows, где переменная SHELL , как правило, не устанавливается. Смотрите раздел Исполнение команд.)

Целе-зависимые (target-specific) значения переменных

Значения переменных в make обычно являются глобальными; другими словами, они одинаковы - в каком бы месте make-файла они ни вычислялись. Одно из исключений - автоматические переменные (смотрите раздел 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 в следующем: когда вы определили целе-зависимую переменную, ее значение также будет "видно" для всех пререквизитов данной цели (конечно, если для пререквизитов не определены свои собственные целе-зависимые переменные). Так, например, в следующем выражении:

```
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

В дополнении к целе-зависимым значениям переменных (смотрите раздел Целе-зависимые значения переменных), GNU make поддерживает шаблонно-зависимые значения переменных. Эти переменные считается определенными для всех целей, подходящих под указанный шаблон. Определенные таким образом переменные "принимаются в рассчет" после целе-зависимых переменных, явно определенных для рассматриваемой цели но до того, как будут рассмотрены целе-зависимые переменные "родительских" целей.

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.

В следующем примере:

```
%.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. Условные конструкции управляют тем, "каким" make "увидит" обрабатываемый make-файл, и, поэтому, их *нельзя* использовать для управления командами оболочки во время их исполнения.

Пример условной конструкции

В следующем примере, 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. В предыдущем примере, это означает использование второй, альтернативной команды компоновки, если первая альтернатива не будет использована. 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.

Если переменная `CC` будет содержать значение `` `gcc' ``, то приведенный выше пример будет работать как:

```
foo: $(objects)
        $(CC) -o foo $(objects) $(libs_for_gcc)
```

При любом другом значении переменной `CC`, тот же пример будет работать

как:

```
foo: $(objects)
        $(CC) -o foo $(objects) $(normal_libs)
```

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 простой (без директивы `else`) условной конструкции выглядит следующим образом:

```
условная-директива
фрагмент-для-выполненного-условия
endif
```

Фрагмент-для-выполненного-условия представляет собой последовательность любых строк текста, которые будут рассматриваться как часть make-файла, если проверяемое условие выполняется. В противном случае, этот текст не используется.

Синтаксис "сложной" условной конструкции выглядит так:

```
условная-директива
фрагмент-для-выполненного-условия
else
фрагмент-для-невыполненного-условия
endif
```

При выполнении условия, используется фрагмент *фрагмент-для-выполненного-условия*; иначе, используется фрагмент *фрагмент-для-невыполненного-условия*. Фрагмент *фрагмент-для-невыполненного-условия* может содержать любое количество строк.

Условная-директива имеет одинаковый синтаксис как в простой, так и в сложной условной конструкции. Имеется четыре разных директивы, проверяющих разные условия. 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` (смотрите раздел Функции анализа и подстановки строк). В следующем примере:

```
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.

`ifneq (`*`параметр1`*`, `*`параметр2`*`)`

`ifneq '`*`параметр1`*`' '`*`параметр2`*`'`

`ifneq "`*`параметр1`*`" "`*`параметр2`*`"`

`ifneq "`*`параметр1`*`" '`*`параметр2`*`'`

`ifneq '`*`параметр1`*`' "`*`параметр2`*`"`

Значения параметров *параметр1* и *параметр2* вычисляются и сравниваются между собой. Если они не совпадают, используется фрагмент *фрагмент-для-выполненного-условия*; иначе используется фрагмент *фрагмент-для-невыполненного-условия* (если он имеется).

`ifdef` *имя-переменной*

Если переменная *имя-переменной* имеет непустое значение, будет использован фрагмент *фрагмент-для-выполненного-условия*; иначе, будет использован фрагмент *фрагмент-для-невыполненного-условия* (если таковой имеется). 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` 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'.

`ifndef` *имя-переменной*
: Если переменная *имя-переменной* имеет пустое значение, будет использован фрагмент *фрагмент-для-выполненного-условия*; иначе, будет использован фрагмент *фрагмент-для-невыполненного-условия* (если такой фрагмент имеется).

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 (смотрите раздел Автоматические переменные).

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 (смотрите раздел Функции анализа и подстановки строк). 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'. Смотрите раздел Рекурсивный вызов make.

Функции преобразования текста

С помощью **функций**, вы можете производить в make-файле некоторую текстовую обработку, определяя с ее помощью имена обрабатываемых файлов или используемые команды. При **вызове функции**, вы указываете ее имя и некоторый текст (**параметры**), который будет обрабатываться этой функцией. Результат работы функции будет "подставлен" в make-файл на месте ее вызова (подобно тому, как вместо ссылки на переменную подставляется ее значение).

Синтаксис вызова функций

Вызов функции внешне напоминает ссылку на переменную. Он выглядит так:

```
$(функция параметры)
```

или так:

```
${функция параметры}
```

Здесь *функция* является именем функции. 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

Ниже перечислены функции, оперирующие со строками:

`$(subst` *заменяемый_фрагмент*`,`*замена*`,`*текст*`)`

Производит текстовую замену в тексте *текст*: каждой вхождение подстроки *заменяемый_фрагмент* заменяется на фрагмент *замена*. Результат подставляется в место вызова функции. Результатом следующего примера:

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

будет строка `` `fEEt on the strEEt' ``.

`$(patsubst` *шаблон*`,`*замена*`,`*текст*`)`

Находит в *тексте* разделенные пробелом слова, удовлетворяющие *шаблону* и заменяет их на строку *замена*. *Шаблон* может содержать символ `%', который работает как специальный шаблонный символ, соответствующий любому количеству произвольных символов внутри слова. Если строка *замена* также содержит символ `%', он будет заменен текстом, соответствующим символу `%' в *шаблоне*. Специальное значение символа `%' может быть отменено предшествующим ему символом `\'. Специальное значение символа `\', который мог бы отменить специальное значение символа `%', может, в свою очередь, быть отменено дополнительным символом `\'. Символы `\', отменяющие специальное значение символов `%' и `\', удаляются из шаблона перед тем, как он будет использоваться для сравнения или подстановки. Символы `\', не могущие повлиять на трактовку `%', остаются нетронутыми. Например, в шаблоне `the\%weird \\%pattern\\' за строкой `the%weird\' следует шаблонный символ `%' и строка `pattern\\'. Два завершающих символа `\' остаются нетронутыми, поскольку они не могут повлиять на трактовку символа `%'. Пробельные символы между словами преобразуются в одиночные пробелы; начальные и конечные пробелы отбрасываются. Например, результатом выражения

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

будет строка `x.c.o bar.o'. Ссылка с заменой (смотрите раздел 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. Выражение:

```
$(переменная:суффикс=замена)
```

эквивалентно:

```
$(patsubst %суффикс,%замена,$(переменная))
```

Пусть, например, у вас имеется список объектных файлов:

```
objects = foo.o bar.o baz.o
```

Тогда, для получения списка соответствующих исходных файлов, вы можете просто написать:

```
$(objects:.o=.c)
```

вместо того, чтобы использовать "обобщенную" форму 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 (смотрите раздел Условные части make-файла). 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. Результатами следующих двух примеров:

```
$(findstring a,a b c)
$(findstring a,b c)
```

будут, соответственно, строки `a' и `' (пустая строка). В разделе Проверка опций запуска make в условных конструкциях приведен достаточно реалистичный пример использования функции findstring.

`$(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. Шаблоны записываются с использованием шаблонного символа `%', аналогично тому, как это делается в функции patsubst (описана выше). Функция filter может быть использована для отделения друг от друга строк (например, имен файлов) разных "типов". В следующем примере:

```
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*. Эта функция представляет собой "противоположность" функции filter.

Если, например, у нас имеется такой фрагмент:

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

то следующее выражение возвратит список объектных файлов, не входящих в `mains':

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

`$(sort` *список*`)`

Отсортировывает слова из *списка* в лексикографическом порядке, удаляя дубликаты (повторяющейся слова). Результатом является список слов, разделенных одиночными пробелами. Так, результатом выражения

```
$(sort foo bar lose)
```

будет строка `bar foo lose'. Даже если вас не интересует лексикографическая сортировка, вы можете пользоваться фунцией `sort` просто для удаления повторяющихся слов.

Вот довольно реалистичный пример использования функций `subst` и `patsubst`. Предположим, у вас имеется make-файл, в котором для указания списка каталогов, где `make` следует производить поиск пререквизитов, используется переменная VPATH (смотрите раздел Переменная VPATH: список каталогов для поиска пререквизитов). В следующем примере демонстрируется, как можно указать компилятору на необходимость поиска заголовочных файлов в том же списке каталогов.

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 (смотрите раздел Директива override).

Функции для обработки имен файлов

Несколько функций ориентированы на работу с именами файлов или списками имен файлов.

Каждая из перечисленных ниже функций выполняет некоторое преобразование имени файла. Аргумент функции рассматривается как последовательность имен файлов, разделенных пробелами. (Начальные и конечные пробелы игнорируются.) Каждое из перечисленных имен файлов преобразуется одинаковым образом и результаты этих преобразований, разделенные одиночными пробелами, объединяются вместе.

`$(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 `` `./' ``. Результатом следующего примера:

```
$(dir src/foo.c hacks)
```

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. К сожалению, мы пока не видим лучшей альтернативы. Результатом следующего примера:

```
$(notdir src/foo.c hacks)
```

будет строка `` `foo.c hacks' ``.

`$(suffix` *имена*`...)`
: Из каждого имени файла, перечисленного в *именах*, выделяется его суффикс. Если имя файла содержит точку, суффиксом имени считается строка, начинающаяся с последнего встреченного символа точки. В противном случае (имя не содержит точек), суффиксом считается пустая строка. Из-за этого, во многих случаях результатом функции может быть пустая строка, хотя список *имен* был непуст. Вдобавок, список имен файлов, полученный на выходе функции, может оказаться "короче" чем список, переданный ей на вход. Например, результатом следующего выражения:

```
$(suffix src/foo.c src-1.0/bar.c hacks)
```

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. Если имя файла содержит точку, то базовым именем считается все, что находится до последнего символа точки (не включая ее). Точки, которые находятся внутри имени каталога, игнорируются. Если рассматриваемое имя файла не содержит точек,

оно целиком считается базовым именем. Результатом следующего примера:

```
$(basename src/foo.c src-1.0/bar hacks)
```

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*. Результатом следующего примера:

```
$(addsuffix .c,foo bar)
```

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*. Например, результатом следующего выражения:

```
$(addprefix src/,foo bar)
```

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 так далее. 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. В следующий примере:

```
$(word 2, foo bar baz)
```

будет получена 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. В следующем примере:

```
$(wordlist 2, 3, foo bar baz)
```

будет получен результат `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. Например, результатом выражения:

```
$(firstword foo bar)
```

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. Смотрите раздел Использование шаблонных символов в именах файлов.

Функция `foreach`

Функция `foreach` сильно отличается от других функций. При ее использовании, некоторый фрагмент текста используется многократно, каждый раз с "подстановкой" в него нового значения. Это напоминает команду `for` командного интерпретатора `sh` или команду `foreach` оболочки `csh`.

Синтаксис функции `foreach` выглядит следующим образом:

```
$(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 определении 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. Смотрите раздел Две разновидности переменных.

Следует быть осторожным при использовании сложных выражений, вычисляющих имя используемой переменной, поскольку допустимым именем переменной могут считаться достаточно странные вещи. Например, следующее выражение:

```
files := $(foreach Esta escrito en espanol!,b c ch,$(find_files))
```

по-видимому, может оказаться полезным только в том случае, если

`find_files` будет содержать ссылку на переменную с именем `` `Esta escrito en espanol!' `` (нет ли у вас такой переменной?). Но, скорее всего, вы просто ошиблись.

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; смотрите раздел Синтаксис условных конструкций).

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`

Функция `call` уникальна тем, что 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.

Для функции `call` используется следующий синтаксис:

`$(call` *переменная*`,`*параметр*`,`*параметр*`,...)`

При вычислении этой функции, `make` помещает каждый из *параметров* во временные переменные `$(1)`, `$(2)` и так далее. 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.

Вот несколько примеров использования функции `call`.

Следующая функция "переставляет" свои аргументы в обратном порядке:

```
reverse = $(2) $(1)

foo = $(call reverse,a,b)
```

Переменная *foo* будет содержать `` `b a' ``.

Следующий пример более интересен: здесь 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. Вот пример реализации функции **map**:

```
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`

В отличие от других функций, функция `origin` не оперирует значениями переменных; вместо этого, она позволяет вам получить некоторую информацию *о самой* переменной. Точнее, она позволяет узнать, откуда "взялась" рассматриваемая переменная.

Синтаксис функции `origin` следующий:

```
$(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'
: Если *переменная* не была определена.

`default'
: Если *переменная* была определена по умолчанию (как, например, переменная `CC` и ей подобные). Смотрите раздел Используемые в неявных правилах переменные. Обратите внимание, что если вы переопределили переменную, имеющую значение по умолчанию, функция `origin` возвратит информацию о более позднем переопределении.

`environment'
: Если *переменная* была создана из соответствующей переменной среды и опция `-e' *не была* включена (смотрите раздел Обзор опций).

`environment override'
: Если *переменная* была создана из соответствующей переменной среды и опция `-e' *была* включена (смотрите раздел Обзор опций).

`file'
: Если *переменная* была определена внутри make-файла.

`command line'
: Если *переменная* была определена с помощью командной строки.

`override'
: Если *переменная* была определена в make-файле с с использованием директивы `override` (смотрите раздел Директива `override`).

`automatic'

Если *переменная* является автоматической переменной, определяемой во время выполнения команд каждого правила (смотрите раздел Автоматические переменные).

Помимо "праздного любопытства", подобная информация может быть полезна, в первую очередь, для того, чтобы определить, насколько вы можете "доверять" значению, содержащемуся в рассматриваемой переменной. Предположим, для примера, 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'. Смотрите раздел Функции анализа и подстановки строк.

Функция shell

В отличие от большинства других функций (кроме, пожалуй, функции wildcard; смотрите раздел Функция 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.

Вот несколько примеров использования функции `shell`:

```
contents := $(shell cat foo)
```

В этом примере, в переменную `contents` записывается содержимое файла `` `foo' `` (видоизмененное таким образом, что все символы перевода строки заменены в нем на пробелы). В следующем примере:

```
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". В следующем примере:

```
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*`...)`

Эта функция 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. Смотрите раздел Вместо исполнения команд.

Аргументы для задания make-файла

Для указания имени make-файла, который следует интерпретировать, служат опции `` `-f' `` и `` `--file' `` (также работает опция `` `--makefile' ``). Например, `` `-f altmake' `` указывает на необходимость использования `` `altmake' `` в качестве make-файла.

При задании сразу нескольких опций `` `-f' `` с аргументом, все указанные make-файлы будут использованы (логически "объединяясь" в один make-файл).

При отсутствии опций `` `-f' `` или `` `--file' ``, по умолчанию, `make` пытается найти

файлы с именами `GNUmakefile', `makefile', и `Makefile' (в указанном порядке). Первый же найденный файл, который существует или может быть построен, используется как make-файл для работы (смотрите раздел Создание make-файлов).

Аргументы для задания главной цели (goal)

Главная цель (goal) - это цель которую стремится достичь make в результате своей работы. Прочие цели make-файла обновляеются 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 (смотрите раздел Автоматическая генерация списка пререквизитов), 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 (смотрите раздел Абстрактные цели) или пустой (смотрите раздел Использование пустых целей для фиксации событий) 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' ``. Ниже приведен список типичных абстрактных и пустых целей. Смотрите раздел Стандартные имена целей для пользователей, где описываются все "стандартные" имена целей, используемые в программах GNU.

`` `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 модефицированы.

`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 модефикации равно текущему времени (при этом, реальное время модефикации этих файлов не меняется). Совместно используя опции `-W' и `-n', можно увидеть, какие действия предпримет `make`, если перечисленные файлы

действительно будут модефицированы.

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 (смотрите раздел Как работает переменная MAKE).

Опцию `-W' можно использовать двумя путями:

- При наличии опций `-n' или `-q', вы можете увидеть, какие действия предпринила бы make, если бы вы модефицировали 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 модефицированы (хотя, на самом деле, они и не были модефицированы).

Другую информацию о make и используемых make-файлах вы можете получить с помощью опций `-p' и `-v' (смотрите раздел Обзор опций).

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 модефикации. Поступая так, вам следует придерживаться следующей процедуры:

1. Используйте команду `make' для перекомпиляции всех файлов, которые действительно в этом нуждаются.
2. Внесите изменения в заголовочные файлы.
3. 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" (смотрите раздел Обзор опций). В таком случае ни этот файл, ни зависящие от него файлы обновляться не будут. Придерживайтесь следующей процедуры:

1. Перекомпилируйте все исходные файлы, которые нуждаются в компиляции по причинам, не связанным с вашей модефикацией заголовочного файла, используя `make -o *имя_заголовочного_файла*'. Если вы изменили несколько заголовочных файлов, используйте отдельные опции `-o' для каждого из них.
2. Обновите время модефикации 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, что 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*' (смотрите раздел Директива override).

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 модефицировали, такое поведение make не слишком удобно. Скорее, вам хотелось бы, чтобы make попробовала скомпилировать все модефицированные 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 объъектных 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; смотрите раздел Обзор опций).

Обычно, make исходит из того, что ваша задача - получить новую версию главной цели; как только make понимает что это невозможно, она сообщает об этом сразу же. Запуск с опцией `-k' говорит, что, на самом деле, 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

Вот полный список опций, распознаваемых программой make:

`-b'
`-m'
: Эти опции 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 (смотрите раздел Рекурсивный вызов make).

`-d'
: В дополнение к основной работе, выводить отладочную информацию. Отладочная информация содержит в себе много интересного: например, 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. Смотрите раздел Переменные из операционного окружения.

`-f *файл*'
`--file=*файл*'
`--makefile=*файл*'
: Указанный *файл* рассматривается в качестве make-файла. Смотрите раздел Создание make-файлов.

`-h'
`--help'
: Напоминает вам 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. Смотрите раздел Подключение других 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 модефицирована. Будучи использована совместно с опцией `-n', покажет, какие действия будут выполнены если этот файл действительно будет модефицирован. Без опции `-n', эффект сходен с выполнением команды `touch` для указанного файла с последующим запуском `make`, за исключеним того, что время последней модефикации 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)

Некоторые "стандартные" способы обновления целевых файлов используются очень часто. Например, одним из типичных способов порождения объектного файла является его получение из соответствующего исходного файла на языке Си с использованием

программы cc (компилятора языка Си).

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. Смотрите раздел Используемые в неявных правилах переменные.

Вы можете определить свои собственные неявные правила с помощью **шаблонных правил (pattern rules)**. Смотрите раздел Определение и переопределение шаблонных правил.

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. Смотрите раздел Устаревшие суффиксные правила.

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.

Представьте себе такой фрагмент make-файла:

```
foo : foo.o bar.o
        cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

Поскольку, вы упомянули файл `foo.o', но не указали правила для его создания, make автоматически попробует найти неявное правило, с помощью

которого можно этот файл обновить. Такая попытка будет произведена независимо от того, существует ли в данный момент файл `foo.o' или нет.

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.

Разумеется, составляя make-файл, вы заранее представляете, какие именно неявные правила вы хотели бы использовать. Ваша уверенность, что make выберет нужные правила, основана на том, что вы знаете, какие именно файлы пререквизитов реально существуют. Смотрите раздел Перечень имеющихся неявных правил, где приведен полный список встроенных неявных правил.

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**). Смотрите раздел "Цепочки" неявных правил.

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, расматриваются 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

Ниже приведен полный список предопределенных неявных правил. Эти правила доступны всегда, если только make-файл явно не заменяет или не отменяет их. Смотрите раздел Отмена действия неявных правил, где обсуждается вопрос перекрытия и отмены действия неявных правил. 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'. Все описанные ниже неявные правила, чьи пререквизиты имеют одно из перечисленных расширений, на самом деле являются суффиксными правилами. Если вы модефицируете список суффиксов, "в силе" останутся только те предопределенные правила, чьи суффиксы остались в этом списке; прочие правила будут "отключены". Смотрите раздел Устаревшие суффиксные правила, где подробно описаны суффиксные правила.

Компиляция программ на языке Си
: `*n*.o' автоматически получается из `*n*.c' при помощи команды `$(CC) -c $(CPPFLAGS) $(CFLAGS)'.

Компиляция программ на языке C++
: `*n*.o' автоматически получается из `*n*.cc' или `*n*.C' с помощью команды `$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)'. Мы рекомендуем вам пользоваться суффиксом `.cc' (вместо `.C').

Компиляция программ на языке Паскаль
: `*n*.o' автоматически получается из `*n*.p' с помощью команды `$(PC) -c $(PFLAGS)'.

Компиляция программ на языках Фортран и Ратфор
: `*n*.o' автоматически получается из `*n*.r', `*n*.F' или `*n*.f' при помощи компилятора Фортрана. При этом, используются следующие команды:

  `.f'
  : `$(FC) -c $(FFLAGS)'.

  `.F'
  : `$(FC) -c $(FFLAGS) $(CPPFLAGS)'.

  `.r'

`$(FC) -c $(FFLAGS) $(RFLAGS)'.

Препроцессорная обработка программ на языках Фортран и Ратфор
: `*n*.f' автоматически получается из `*n*.r' или `*n*.F'. При этом, исходный текст на Ратфоре или Фортране пропускается через препроцессор для получения "обычной" фортрановской программы. Для этого, используются следующие команды:

  `.F'
  `$(FC) -F $(CPPFLAGS) $(FFLAGS)'.

  `.r'
  `$(FC) -F $(FFLAGS) $(RFLAGS)'.

Компиляция программ на языке Modula-2
: `*n*.sym' получается из `*n*.def' с помощью команды `$(M2C) $(M2FLAGS) $(DEFFLAGS)'. `*n*.o' получается из `*n*.mod' с помощью команды `$(M2C) $(M2FLAGS) $(MODFLAGS)'.

Ассемблирование и препроцессорная обработка ассемблерных программ
: `*n*.o' автоматически получается из `*n*.s' путем запуска ассемблера (`as'). Для этого используется команда `$(AS) $(ASFLAGS)'. `*n*.s' автоматически получается из `*n*.S' с помощью препроцессора языка Си (`cpp'). При этом используется команда `$(CPP) $(CPPFLAGS)'.

Компоновка одиночного объектного файла
: `*n*' автоматически получается из `*n*.o' путем запуска компоновщика (обычно, `ld') с помощью компилятора Си. Для этого используется команда `$(CC) $(LDFLAGS) *n*.o $(LOADLIBES) $(LDLIBS)'. Это правило будет работать правильно как в простейшем случае, когда текст программы состоит только из одного исходного файла, так и в более сложной ситуации, когда программа состоит из нескольких объектных файлов, один из которых имеет имя, соответствующее имени исполняемого файла. Таким образом, обрабока правила

  ```
  x: y.o z.o
  ```

  при наличии файлов `x.c', `y.c' и `z.c', вызовет выполнение команд:

  ```
  cc -c x.c -o x.o
  cc -c y.c -o y.o
  cc -c z.c -o z.o
  cc x.o y.o z.o -o x
  rm -f x.o
  rm -f y.o
  rm -f z.o
  ```

  В более сложных ситуациях, когда, например, отсутствует объектный файл, чье имя соответствует имени исполняемого файла, вам придется задать явное правило для компоновки программы. Все типы файлов, автоматически преобразуемые в объектные файлы `.o', автоматически компонуются путем вызова соответствующего компилятора (`$(CC)', `$(FC)' или `$(PC)'; компилятора Си `$(CC)' для ассемблирования файлов `.s') без опции `-c'. Для ускорения работы, компиляция и компоновка делается за один шаг (вместо того, чтобы использовать объектные файлы `.o' в качестве промежуточных).

Генерация программы на Си с помощью Yacc
: `*n*.c' автоматически получается из `*n*.y' путем запуска программы Yacc с помощью команды `$(YACC) $(YFLAGS)'.

Генерация программы на Си с помощью Lex

`n.c' автоматически получается из `n.l' с помощью программы Lex. При этом, используется команда `$(LEX) $(LFLAGS)'.

Генерация программы на Ратфоре с помощью Lex
: `n.r' автоматически получается из `n.l' с помощью программы Lex. Для этого используется команда `$(LEX) $(LFLAGS)'. Соглашение об использовании суффикса `.l' для всех Lex-файлов, независимо от того, должны ли они преобразовываться в код на языках Си или Ратфор, делает невозможным автоматическое определение используемого в данном случае языка. При необходимости получения объектного файла из файла `.l', программа `make` должна решить - какой компилятор нужно использовать. Она будет использовать компилятор Си, поскольку он более распространен. Если вы используете язык Ратфор, убедитесь в том, что `make` знает об этом. Для этого достаточно, чтобы нужный файл `n.r' был упомянут в make-файле. Или, если вы пользуетесь исключительно Ратфором и не используете Си, удалите `.c' из списка суффиксов для неявных правил:

```
.SUFFIXES:
.SUFFIXES: .o .r .f .l ...
```

Получение Lint-библиотек из исходных файлов на Си, Yacc или Lex
: `n.ln' получается из `n.c' путем запуска `lint`. При этом, используется команда `$(LINT) $(LINTFLAGS) $(CPPFLAGS) -i'. Эта же команда используется для Си-программ, полученных из `n.y' или `n.l'.

Файлы программ TeX и Web
: `n.dvi' получается из `n.tex' с помощью команды `$(TEX)'. `n.tex' получается из `n.web' с помощью `$(WEAVE)', или из `n.w' (и из `n.ch' если такой файл существует или может быть получен) с помощью `$(CWEAVE)'. `n.p' получается из `n.web' с помощью `$(TANGLE)' и `n.c' получается из `n.w' (и из `n.ch' если такой существует или может быть создан) с помощью `$(CTANGLE)'.

Файлы программ Texinfo и Info
: `n.dvi' получается из `n.texinfo', `n.texi', или `n.txinfo', с помощью команды `$(TEXI2DVI) $(TEXI2DVI_FLAGS)'. `n.info' получается из `n.texinfo', `n.texi', или `n.txinfo', с помощью команды `$(MAKEINFO) $(MAKEINFO_FLAGS)'.

RCS
: Любой файл `n' при необходимости может быть извлечен из RCS-файла с именем `n,v' или `RCS/n,v'. Для этого используется команда `$(CO) $(COFLAGS)'. Если файл `n' уже существует, он не будет извлекаться из RCS-файла, даже если этот RCS-файл является более "новым". Правила для RCS являются терминальными (смотрите раздел Шаблонные правила с призвольным соответствием), поэтому RCS-файлы не могут быть получены из других файлов - они должны действительно существовать.

SCCS
: Любой файл `n' при необходимости может быть извлечен SCCS-файла с именем `s.n' или `SCCS/s.n'. Для этого используется команда `$(GET) $(GFLAGS)'. Правила для SCCS являются терминальными (смотрите раздел Шаблонные правила с призвольным соответствием), поэтому SCCS-файлы не могут быть получены из других файлов - они должны в действительности существовать. При работе с системой SCCS, файл с именем `n.sh' извлекается в файл `n' и его атрибуты устанавливаются

как "исполняемый всеми (everyone)". Это делается для нормальной работы скриптов, помещаемых в SCCS. При работе с системой RCS подобных манипуляций не производится, поскольку она запоминает атрибуты помещаемых в нее файлов. Мы не рекомендуем вам использовать систему SCCS. Общепризнано, что RCS более совершенна и, к тому же, является свободным программным обеспечением. Используя свободное программное обеспечение вместо аналогичного или даже худшего "закрытого" (proprietary) программного обеспечения, вы, тем самым, поддерживаете движение за свободное программное обеспечение.

Как правило, у вас может возникнуть необходимость в изменении только упоминаемых выше переменных (все они перечислены в следующем разделе).

Однако, на самом деле, для выполнения команд, предопределенные неявные правила используют такие переменные, как `COMPILE.c`, `LINK.p`, и `PREPROCESS.S`, которые содержат указанные выше 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`; смотрите раздел Поиск пререквизитов по каталогам). 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.

Вот список переменных, используемых в неявных правилах для хранения имен программ:

AR
: Программа работы с архивами; по умолчанию, `ar'.

AS
: Ассемблер; по умолчанию, `as'.

CC
: Компилятор Си; по умолчанию, `cc'.

CXX
: Компилятор C++; по умолчанию, `g++'.

CO
: Программа для извлечения файлов из RCS; по умолчанию, `co'.

CPP
: Препроцессор языка Си, выдающий результат на стандартный вывод; по умолчанию, `$(CC) -E'.

FC
: Препроцессор и компилятор для Фортрана и Ратфора; по умолчанию, `f77'.

GET
: Программа для извлечения файлов из SCCS; по умолчанию, `get'.

LEX
: Программа для преобразования Lex-грамматики в текст на языках Си или Ратфор; по умолчанию - `lex'.

PC
: Компилятор Паскаля; по умолчанию, `pc'.

YACC
: Программа для преобразования Yacc-грамматики в текст на Си; по умолчанию - `yacc'.

YACCR
: Программа для преобразования Yacc-грамматики в текст на языке Ратфор; по умолчанию - `yacc -r'.

MAKEINFO

Программа для преобразования исходного файла формата Texinfo в файл формата Info; по умолчанию, `makeinfo'.

TEX
: Программа для преобразования исходных файлов на TeX в файлы формата DVI; по умолчанию - `tex'.

TEXI2DVI
: Программа, преобразующая исходные файлы в формате Texinfo, в DVI-файлы программы TeX; по умолчанию - `texi2dvi'.

WEAVE
: Программа, преобразующая текст из формата Web в формат TeX; по умолчанию - `weave'.

CWEAVE
: Программа, преобразующая текст на Си-Web в формат TeX; по умолчанию - `cweave'.

TANGLE
: Программа, преобразующая текст на Web в Паскаль; по умолчанию - `tangle'.

CTANGLE
: Программа, преобразующая текст на Си-Web в текст на Си; по умолчанию - `ctangle'.

RM
: Команда удаления файла; по умолчанию, `rm -f'.

Ниже приведена таблица переменных, содержащих дополнительные параметры для перечисленных выше программ. По умолчанию, значением этих переменных является пустая строка (если не указано другое).

ARFLAGS
: Опции, передаваемые программе, манипулирующей с архивными файлам; по умолчанию `rv'.

ASFLAGS
: Дополнительные параметры, передаваемые ассемблеру (при его явном вызове для файлов `.s' и `.S').

CFLAGS
: Дополнительные параметры, передаваемые компилятору Си.

CXXFLAGS
: Дополнительные параметры, передаваемые компилятору C++.

COFLAGS
: Дополнительные параметры, передаваемые программе `co` (входящей в систему RCS).

CPPFLAGS
: Дополнительные параметры, передаваемые препроцессору языка Си и программам, его использующим (компиляторам Си и Фортрана).

FFLAGS
: Дополнительные параметры для компилятора Фортрана.

GFLAGS
: Дополнительные параметры, передаваемые программе `get` (входящей в систему SCCS).

LDFLAGS
: Дополнительные параметры, передаваемые компиляторам, когда предполагается вызов компоновщика `ld'.

`LFLAGS`
: Дополнительные параметры, передаваемые программе Lex.

`PFLAGS`
: Дополнительные параметры, передаваемые компилятору Паскаля.

`RFLAGS`
: Дополнительные параметры, передаваемые компилятору Фортрана при компиляции программ на Ратфоре.

`YFLAGS`
: Дополнительные параметры, передаваемые программе Yacc.

"Цепочки" (chains) неявных правил

В некоторых случаях, файл может быть получен путем последовательного применения нескольких неявных правил. Например, файл `` `n.o' `` может быть получен из `` `n.y' `` с помощью программы Yacc и последующего запуска компилятора `cc`. 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'.

Обратите внимание, что "расширение" символа `%' в шаблонном правиле прозводится **после** вычисления всех переменных и функций (что имеет место в момент чтения make-файла). Смотрите раздел Использование переменных и раздел Функции преобразования текста.

Введение в шаблонные правила (pattern rules)

Шаблонное правило содержит в имени цели символ `%' (в точности один); в остальном, оно выглядит как обычное правило. Цель является шаблоном, с которым сверяются имена файлов; символ `%' может соответствовать любой непустой строке, другие символы должны в точности совпадать.

Например, 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.

Таким образом, правило вида

```
%.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. Смотрите раздел Определение правил "последнего шанса".

Шаблонные правила могут иметь несколько целей. Однако, в отличие от обычных правил, такой случай не рассматривается как несколько правил с одними и теми же пререквизитами и командами. Наличие в шаблонном правиле нескольких целей, означает для `make`, что указанные в правиле команды обновляют сразу все перечисленные цели. Команды правила исполняются только один раз, вызывая обновление всех целей. 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, посколько 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

Вот несколько примеров шаблонных правил, в действительности предопределенных в `make`. Сперва, правило, компилирующее файлы `` `.c' `` в файлы `` `.o' ``:

```
%.o : %.c
        $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

С помощью этого правила, любой файл `` `x.o' `` может быть получен из соответствующего файла `` `x.c' ``. Для того, чтобы при каждом применении этого правила, использовались правильные имена целевого и исходного файлов, его команда использует автоматические переменные `` `$@' `` и `` `$<' `` (смотрите раздел Автоматические переменные).

Вот другое встроенное правило:

```
% :: 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. (смотрите раздел Шаблонные правила с призвольным соответствием).

Следующее правило имеет две цели:

```
%.tab.c %.tab.h: %.y
        bison -d $<
```

Правило говорит о том, что выполнение команды `bison -d x.y' приведет к созданию обоих файлов - `x.tab.c' и `x.tab.h'. Например, если файл `foo' зависит от файлов `parse.tab.o' и `scan.o', а файл `scan.o' зависит от файла `parse.tab.h', то при изменении `parse.y', команда `bison -d parse.y' будет запущена только один раз, в результате чего пререквизиты обоих файлов `parse.tab.o' и `scan.o', будут обновлены. (Предполагается, что файл `parse.tab.o' будет скомпилирован из `parse.tab.c', а файл `scan.o' - из файла `scan.c'. В свою очередь, `foo' компонуется из `parse.tab.o', `scan.o' и прочих, после чего успешно работает.)

Автоматические переменные

Предположим, вы хотите написать шаблонное правило для компиляции файлов `.c' в объектные файлы `.o': каким образом вам задать правильное имя исходного файла для команды `cc'? Вы не можете явно указать имя файла, поскольку это имя будет различным в каждом случае использования данного правила.

Здесь, вам придется воспользоваться так называемыми **автоматическими переменными**. Значения этих переменных автоматически вычисляются заново для каждого исполняемого правила в зависимости от его целей и пререквизитов. В приведенном выше примере вам следует использовать `$@' в качестве имени объектного файла и `$<' в качестве имени исходного файла.

Вот полный перечень имеющихся автоматических переменных:

`$@`
: Имя файла цели правила. Если цель является элементом архива (archive member), то `$@' обозначает имя архивного файла. В шаблонном правиле с несколькими целями (смотрите раздел Введение в шаблонные правила), `$@' обозначает имя цели, которая вызывала запуск команд данного правила.

`$%`
: Для целей, являющихся элементами архива, обозначает имя этого элемента. Смотрите раздел Использование make для обновления архивов. Например, для цели `foo.a(bar.o)' переменная `$%' принимает значение `bar.o', а переменная `$@' - значение `foo.a'. Если цель не является элементом архива, `$%' имеет пустое значение.

`$<`
: Имя первого пререквизита. В случае, если выполняемые команды относятся к неявному правилу, первым пререквизитом является тот, который был указан в неявном правиле. (смотрите раздел Использование неявных правил).

`$?`
: Имена всех пререквизитов (разделенные пробелами), которые

являются "более новыми", чем цель. Для членов архива, используется имя самого элемента (смотрите раздел Использование make для обновления архивов).

`$^`

Имена всех пререквизитов (разделенные пробелами). Для пререквизитов, которые являются элементами архивов, используются только имена элементов (смотрите раздел Использование make для обновления архивов). 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 (смотрите раздел Процедура сопоставления с шаблоном). 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 `$*` вычисляется по-другому. В случае явного правила, если имя цели имеет один из "известных" суффиксов (смотрите раздел Устаревшие суффиксные правила), в переменную `$*` 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.

Переменная `$?` может оказаться полезна даже в явных правилах, когда вы хотите работать только с пререквизитами, которые были изменены. Представьте, например, что архивный файл `lib` содержит копии некоторых объектных файлов. Следующее правило запишет в архив копии только изменившихся объектных файлов:

```
lib: foo.o bar.o lose.o win.o
        ar r lib $?
```

Из всех перечисленных выше переменных, четыре переменные представляют собой одиночные имена файлов, а две переменные - списки имен файлов. 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 (смотрите раздел Функции для обработки имен файлов). 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. (Смотрите раздел Использование элементов архива в качестве целей.)

`$(<D)'
`$(<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

Шаблон цели состоит из символа `%', заключенного между префиксом и суффиксом, каждый из которых (или оба сразу) может быть пустым. Имя файла будет подходить под указанный шаблон только если оно начинается с указанного префикса и заканчивается указанным суффиксом (причем, префикс и суффикс не могут перекрываться). Текст, заключенный между префиксом и суффиксом, называется **основой (stem)**. Таким образом, файл `test.o' удовлетворяет шаблону `%.o' и его основой является строка `test'. Пререквизиты шаблонного правила преобразуются в имена конкретных файлов, путем подстановки основы вместо символа `%'. Так, в приведенном выше примере, из пререквизита `%.c', будет получено имя файла `test.c'.

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.

Конечно, мы знаем, что подобные сценарии бессмыслены, поскольку `foo.c' является исходным текстом на языке Си, а не исполняемым файлом. 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, мы внесли некоторые ограничения в процесс обработки программой `make` шаблонных правил с произвольным соответствием. 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.

Например, встроенные неявные правила для извлечения исходных файлов из файлов RCS и SCCS, являются терминальными. В результате, например, при отсутствии файла `` `foo.c,v' ``, `make` не будет пытаться получить его из файлов `` `foo.c,v.o' `` или `` `RCS/SCCS/s.foo.c,v' ``. Как правило, файлы RCS и SCCS являются действительно "исходными", и не могут быть получены из любых других файлов; таким образом, `make` может сэкономить массу времени, не пытаясь найти способ для их обновления.

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 (смотрите раздел Устаревшие суффиксные правила).

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 (смотрите раздел Шаблонные правила с призвольным соответствием). 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 (смотрите раздел Пустые команды).

Вы можете использовать правило "последнего шанса" для "перекрытия" части другого make-файла. Смотрите раздел"Перекрытие" части 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; смотрите раздел Отмена действия неявных правил). 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.

Алгоритм поиска неявных правил

Ниже приведен алгоритм, которому следует make в процессе поиска подходящего неявного правила для цели *t*. Эта процедура выполняется для всех правил с двойным двоеточием, у которых нет команд. Она также выполняется для каждой цели обычных правил, если ни в одном из этих правил не указано команд. Этот же алгоритм поиска применяется для всех пререквизитов, не являющихся целями ни одного из правил make-файла, а также рекурсивно выполняется для пререквизитов неявных правил в процессе поиска возможных "цепочек" неявных правил.

Суффиксные правила не упоминаются в этом алгоритме, поскольку преобразуются в эквивалентные им шаблонные правила во время чтения make-файла.

Для целей, которые являются элементами архива (имеют форму `*архив*(*элемент*)'), указанный ниже алгоритм применяется дважды; первый раз - с использованием полного имени цели *t*, и во второй раз - используя `(*элемент*)' в качестве цели *t* (если при первом проходе подходящее правило найдено не было).

1. Разделить имя цели *t* на две части - имя каталога (*d*), и остаток строки (без имени каталога) *n* . Например, если имя *t* представляет собой строку `src/foo.o', то частью *d* будет `src/', а частью *n* - фрагмент `foo.o'.
2. Составить список шаблонных правил, чьи цели могут быть сопоставлены с *t* или *n*. Если шаблон цели содержит символ '/', он будет сравниваться с *t*; в противном случае, шаблон будет сравниваться с *n*.
3. Если хотя бы одно правило из этого списка, *не* является правилом с произвольным соответствием, удалить из списка все нетерминальные правила с произвольным соответствием.
4. Удалить из списка все правила, не имеющие команд.
5. Для каждого шаблонного правила из списка:
    1. Ищется основа *s*, которая представляет из себя непустую часть *t* или *n*, соответствующую символу `%' в шаблоне цели.
    2. Путем подстановки основы *s* вместо `%', вычисляются имена пререквизитов; Если шаблон цели не содержит символа '/',

добавить *d* в начало имени каждого из пререквизитов.
    3. Проверить, все ли пререквизиты существуют или должны существовать. (Мы говорим, что файл "должен существовать", если он упоминается в make-файле в качестве цели или явно указанного пререквизита.) Если все пререквизиты существуют или должны существовать, или, если в правиле нет пререквизитов, это правило применяется.
6. Если подходящее шаблонное правило до сих пор не найдено, производится вторая попытка - для каждого шаблонного правила из списка:
    1. Если правило является терминальным, пропустить его и перейти к следующему правилу.
    2. Вычислить имена пререквизитов (также, как это делалось ранее, во время первого прохода).
    3. Проверить, все ли пререквизиты существуют или могут быть созданы.
    4. Рекурсивно используя описываемый алгоритм поиска, для каждого несуществующего пререквизита определить, может ли он быть создан с помощью подходящего неявного правила.
    5. Если все пререквизиты существуют, должны существовать, или могут быть получены с помощью неявных правил, рассматриваемое правило применяется.
7. Если ни одно из неявных правил не может быть применено, используется правило для цели `.DEFAULT` (если такое правило существует). В этом случае, для цели *t* выполняются те же команды, какие указаны для цели `.DEFAULT`. Иначе, считается, что для цели *t* нет команд.

Как только подходящее правило будет найдено, для каждого шаблона цели (кроме того, который был сопоставлен с *t* или *n*), символ `` `%' `` заменяется на *s* и получившееся имя запоминается до тех пор, пока не будут выполнены команды, обновляющие *t*. После выполнения этих команд, все запомненные имена файлов добавляются в "базу данных" и помечаются как обновленные (информация о статусе обновления, при этом, заимствуется из файла *t*).

При выполнении команд шаблонного правила для цели *t*, автоматические переменные получают значения, соответствующие данной цели и пререквизитам. Смотрите раздел Автоматические переменные.

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. Так, следующий пример:

```
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. Смотрите раздел Использование шаблонных символов в именах файлов. 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

Вспомните, что 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. Смотрите раздел "Цепочки" неявных правил.

Подобные неявные правила пишутся с использованием автоматической переменной `$%'. Смотрите раздел Автоматические переменные.

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; смотрите раздел Параллельное исполнение команд) 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.

Возможно, будущие версии make будут иметь какой-либо механизм для обхода этой проблемы, например, с помощью "сериализации" все команд, оперирующих с одним и тем же архивным файлом. В настоящее время, однако, вам либо придется писать свои make-файлы таким образом, чтобы избежать подобной проблемы, либо отказаться от использования опции -j.

Суффиксные правила для архивных файлов

Для работы с архивными файлами, вы можете использовать суффиксное правило специального вида. Смотрите раздел Устаревшие суффиксные правила, где подробно обсуждаются суффиксные правила. Суффиксные правила для архивных файлов можно считать устаревшими, поскольку поддерживаемые в GNU make шаблонные правила для архивов являются более "общим" механизмом (смотрите раздел Неявные правила для целей - элементов архива). 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 првилу:

```
(%.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'.

Возможности GNU make

Ниже приведен обзор возможностей программы GNU make в сравнении с другими версиями make и указано, откуда были "позаимствованы" те или иные функции. В качестве "базовой", мы рассматриваем версию make из состава операционной системы BSD 4.2. Если вы стремитесь к написанию "переносимых" make-файлов, вам следует использовать только те возможности make, которые не перечислены здесь, а также в разделе Несовместимость и нереализованные функции.

Многие возможности были позаимствованы из версии программы make, имевшейся в System V.

- Переменная VPATH и ее специальное значение. Смотрите раздел Поиск пререквизитов по каталогам . Такая возможность существовала в make из System V, однако не была документирована. Она была документирована в make, входившей в состав BSD 4.3 (в документации говорилось, что она имитирует соответствующие возможности make из System V).
- Включаемые make-файлы. Смотрите раздел Подключение других make-файлов. Возможность подключения нескольких файлов в одной директиве, является особенностью GNU make.
- Переменные могут читаться из операционного окружения и передаваться через него. Смотрите раздел Переменные из операционного окружения.
- При рекурсивном вызове, опции передаются через переменную MAKEFLAGS в make "нижнего уровня". Смотрите раздел Передача опций в make "нижнего уровня".
- При работе с архивом, в автоматическую переменную $% заносится имя элемента архива. Смотрите раздел Автоматические переменные.
- Автоматические переменные $@, $*, $<, $%, и $? имеют "родственные" формы вида $(@F) и $(@D). Подобные же две формы имеет и переменная $^ - это было "само собой напрашивающееся" расширение. Смотрите раздел Автоматические переменные.
- Ссылка на переменную с заменой. Смотрите раздел Обращение к переменным.

- Опции командной строки `-b' и `-m', принимаются, но игнорируются (хотя в make из System V они выполняют какие-то действия).
- Выполнение рекурсивных команд для запуска make через переменную MAKE, даже при наличии опций `-n', `-q' или `-t'. Смотрите раздел Рекурсивный вызов make.
- Поддержка суффикса `.a' в суффиксных правилах. Смотрите раздел Суффиксные правила для архивных файлов. В GNU make подобную возможность можно считать устаревшей, поскольку обычного шаблонного правила вполне достаточно: остальную работу берет на себя обобщенный механизм "цепочек" неявных правил (смотрите раздел "Цепочки" неявных правил) и шаблонное правило для занесения элементов в архивы (смотрите раздел Неявные правила для целей - элементов архива).
- "Оригинальный" вид команд (как они записаны в make-файле), включая расположение строк и комбинаций "\, перевод строки", при печати команд сохраняется (за исключением начальных пробелов).

Следующие возможности "навеяны" различными версиями make. В некоторых случаях, мы затрудняемся сказать, откуда именно были "позаимствованы" те или иные возможности.

- Шаблонные правила с использованием шаблонного символа `%'. Это было реализовано в нескольких различных версиях make. Трудно сказать, кто именно является изобретателем подобной конструкции, однако, она распространилась довольно широко. Смотрите раздел Определение и переопределение шаблонных правил.
- Связывание правил в цепочки и промежуточные файлы. Это было реализовано Stu Feldman в его версии make для AT&T Eighth Edition Research Unix, и, позднее, Andrew Hume из AT&T Bell Labs в его программе mk (он использовал термин "transitive closure"). Мы затрудняемся сказать, была ли подобная возможность "позаимствована" нами из этих программ или же мы независимо "изобрели" ее сами. Смотрите раздел "Цепочки" неявных правил.
- Автоматическая переменная $^, содержащая список всех пререквизитов текущей цели. Мы не знаем, чье это "изобретение", но уж точно - не наше. Смотрите раздел Автоматические переменные. Автоматическая переменная $+ является просто "расширением" $^.
- Опция "что, если" (`-W' в GNU make) была (насколько нам известно) придумана Andrew Hume для программы mk. Смотрите раздел Вместо исполнения команд.
- Концепция одновременного выполнения нескольких команд (параллелизм) существует во многих версиях make и ей подобных программ. Однако, в реализациях make из System V и BSD, такой возможности нет. Смотрите раздел Исполнение команд.
- Модефицированный вариант ссылки на переменную с возможностью замены, "пришел" из SunOS 4. Смотрите раздел Обращение к переменным. До тех пор, пока этот альтернативный синтаксис не был поддержан в GNU make (для обеспечения совместимости с SunOS 4), подобная функциональность достигалась с использованием функции patsubst. Сложно сказать, что здесь явилось "первоисточником", поскольку функция patsubst существовала в GNU make еще до того, как

была реализована система SunOS 4.
- Специальное значение символа `+' в начале строки, содержащей команды (смотрите раздел Вместо исполнения команд), определено стандартом *IEEE Standard 1003.2-1992* (POSIX.2).
- Синтаксис оператора `+=', добавляющего значение к переменной, "позаимствован" из make системы SunOS 4. Смотрите раздел Добавление текста к переменной.
- Синтаксис `*архив*(*элемент1 элемент2*...)' для указания нескольких элементов одного архивного файла, взят из make, входящей в SunOS 4. Смотрите раздел Использование элементов архива в качестве целей.
- Директива -include, позволяющая подключать make-файлы с игнорированием несуществующих, была взята из программы make операционной системы SunOS 4. (Заметьте, однако, что SunOS 4 make не допускала перечисления нескольких файлов в одной директиве -include.) Аналогичная возможность (под именем sinclude) присутствует в версиях make от SGI и, возможно, других системах.

Остальные возможности являются нововведениями GNU make:

- Использование опций `-v' и `--version' для вывода информации о версии и авторских правах.
- Использование опций `-h' и `--help' для вывода подсказки о воспринимаемых программой make опциях.
- Упрощенно вычисляемые переменные. Смотрите раздел Две разновидности переменных.
- При рекурсивном использовании make, определение переменной, производимое с помощью командной строки, автоматически передается в make "нижнего уровня" через переменную MAKE. Смотрите раздел Рекурсивный вызов make.
- Использование опций `-C' и `--directory' для смены рабочего каталога. Смотрите раздел Обзор опций.
- Определение многострочных переменных с помощью конструкции define. Смотрите раздел Многострочные переменные.
- Явное объявление абстрактных целей с использование специальной цели .PHONY. Подобная возможность (но с другим синтаксисом) была реализована также Andrew Hume из AT&T Bell Labs в его программе mk. По-видимому, это может считаться примером независимого "параллельного" изобретения. Смотрите раздел Абстрактные цели.
- Манипуляции с текстом путем вызова функций. Смотрите раздел Функции преобразования текста.
- Использование опций `-o' и `--old-file' для имитации того, что время модефикации файла является "старым". Смотрите раздел Предотвращение перекомпиляции некоторых файлов.
- Условные конструкции. Подобная возможность реализовывалась многократно в различных версиях make; она, скорее, выглядит логичным расширением "в духе" препроцессора языка Си и аналогичных макроязыков, нежели "революционной" концепцией. Смотрите раздел Условные части make-файла.
- Возможность задания путей поиска для подключаемых файлов. Смотрите раздел Подключение других make-файлов.
- Возможность указания дополнительных make-файлов с помощью

переменной среды. Смотрите раздел Переменная MAKEFILES.
- Удаление начальной последовательности `./' из имен файлов, дабы `./*файл*' и `*файл*' рассматривались как один и тот же файл.
- Использование специального метода поиска для пререквизитов, которые являются библиотеками и записаны в форме `-l*имя*'. Смотрите раздел Поиск в каталогах для подключаемых библиотек.
- Суффиксы (для суффиксных правил) могут содержать произвольные символы (смотрите раздел Устаревшие суффиксные правила). Во многих других версиях make, суффиксы обязательно должны начинаться с `.' и не должны содержать символов `/'.
- Отслеживание текущего "уровня вложенности" при рекурсивном вызове make, с помощью переменной MAKELEVEL. Смотрите раздел Рекурсивный вызов make.
- Занесение имен всех целей, заданных через командную строку, в переменную MAKECMDGOALS. Смотрите раздел Аргументы для задания главной цели.
- Статические шаблонные правила. Смотрите раздел Статические шаблонные правила.
- "Выборочный" поиск с помощью директивы vpath. Смотрите раздел Поиск пререквизитов по каталогам.
- Вычисляемые имена переменных. Смотрите раздел Обращение к переменным.
- Автоматическое обновление make-файлов. Смотрите раздел Автоматическое обновление make-файлов. Программа make из состава System V make, в сильно "усеченном" виде имела подобную функциональность (умела обновлять make-файлы, получая их новые версии из системы SCCS.
- Новые встроенные неявные правила. Смотрите раздел Перечень имеющихся неявных правил.
- Встроенная переменная `MAKE_VERSION', содержащая номер версии программы make.

Несовместимость и нереализованные функции

Не все возможности, имеющиеся в различных версиях make реализованы в GNU make. Однако, среди нет таких, которые требовались бы стандартом POSIX.2 (*IEEE Standard 1003.2-1992*).

- Цель, записанная как `*файл*(*имя_символа*)', означает элемент архивного файла *файл*. Однако, будучи объектным файлом, элемент выбирается не по своему имени, а по наличию определенного внутри него символьного имени *имя_символа*. Мы не стали добавлять такую возможность в программу GNU make, поскольку это потребовало бы от нее знания формата "внутреннего" представления таблицы символов, хранящейся внутри архивного файла, что противоречило бы принципам модульности. Смотрите раздел Обновление каталога символов архивного файла.
- Суффиксы (используемые в суффиксных правилах), которые

оканчиваются символом `~', имеют специальное значение в make из System V; они ссылаются на файл SCCS, соответствующий файлу, чьим именем является имя суффикса без конечного `~'. Так, например, суффиксное правило `.c~.o' будет описывать процесс получения файлов `*n*.o' из SCCS-файлов `s.*n*.c'. Для полного охвата всех возможных случаев, потребуется целый набор подобных суффиксных правил. Смотрите раздел Устаревшие суффиксные правила. В GNU make, все подобные ситуации обрабатываются с помощью двух шаблонных правил для извлечения файлов из SCCS и "обобщенного" механизма связывания правил в цепочки. Смотрите раздел "Цепочки" неявных правил.

- В make из System V, запись `$$@' имеет достаточно странный смысл: в пререквизитах правила с несколькими целями, она означает конкретную цель, которая в данный момент обрабатывается. Такая возможность не предусмотрена в GNU make, поскольку `$$' всегда означает просто `$'. Подобную функциональность можно получить с помощью статических шаблонных правил (смотрите раздел Статические шаблонные правила). Например, правило для make из System V:

  ```
  $(targets): $$@.o lib.a
  ```

  может быть заменено на статическое шаблонное правило для GNU make:

  ```
  $(targets): %: %.o lib.a
  ```

- В реализациях make для System V и BSD 4.3, файлы, найденные в процессе поиска по каталогам с использованием VPATH (смотрите раздел Поиск пререквизитов по каталогам), меняют свои имена внутри командных строк. Нам кажется, что намного проще всегда использовать автоматические переменные, а подобную возможность считать "устаревшей".
- В реализациях make для некоторых вариантов Unix, автоматическая переменная $*, будучи указана в качестве пререквизита правила, получает весьма странное значение, а именно - полное имя *цели данного правила*. Мы затрудняемся сказать, зачем это могло понадобиться разработчикам этих вариантов make; в действительности, это совершенно не согласуется с "нормальным" определением $*.
- В некоторых реализациях make, поиск неявных правил (смотрите раздел Использование неявных правил), по-видимому, производится для *всех* целей, а не только для тех, которые не имеют команд. Это, например, означает, что при наличии одного лишь правила:

  ```
  foo.o:
          cc -c foo.c
  ```

  make "догадается" о зависимости файла `foo.o' от `foo.c'. Нам это кажется неправильным. Свойства пререквизитов в make определены достаточно хорошо (по крайней мере, в GNU make), а подобные вещи плохо "укладываются" в общую модель.
- GNU make не содержит встроенных неявных правил для компиляции или препроцессорной обработки программ на языке EFL. Если мы узнаем, что кто-то использует этот язык, мы с удовольствием добавим

нужные правила.

- По всей видимости, реализация `make` из системы SVR4 поддерживает суффиксные правила без команд, но рассматривает их как правила с пустыми командами (смотрите раздел Пустые команды). Так, например, правило:

  ```
  .c.a:
  ```

  "перекроет" встроенное суффиксное правило для `` `.c.a' ``. Нам кажется более "естественным", чтобы правила без команд просто добавляли к цели указанные в них пререквизиты. Для получения аналогичного эффекта, приведенный выше пример для GNU `make` можно переписать так:

  ```
  .c.a: ;
  ```

- Некоторые версии `make` всегда вызывают командный интерпретатор с опцией `` `-e' ``, за исключением тех случаев, когда `make` была запущена с опцией `` `-k' `` (смотрите раздел Проверка компиляции программы). Опция `` `-e' `` инструктирует командный интерпретатор завершать работу сразу же, как только любая из запущенных им программ вернет ненулевой код возврата. Нам кажется более естественным помещать каждую командную строку для оболочки на отдельную строку make-файла и не использовать такую специальную возможность.

Принятые соглашения для make-файлов

В этом разделе описаны соглашения, принятые для make-файлов программ GNU. Пакет Automake может помочь вам в создании make-файлов, следующих этим соглашениям.

Общие соглашения для make-файлов

Каждый make-файл должен содержать строку:

```
SHELL = /bin/sh
```

во избежании проблем при работе в системах, где переменная `SHELL` может быть "унаследована" из операционного окружения. (Для GNU `make`, однако, это не составляет проблемы.)

Различные варианты программы `make` имеют разный набор суффиксов и суффиксных правил - иногда это может служить источником недоразумений и ошибок. Поэтому, "хорошей идеей" является явное задание списка суффиксов, содержащего только те суффиксы, которые используются в данном конкретном make-файле, например:

```
.SUFFIXES:
.SUFFIXES: .c .o
```

Первая строка очищает используемый список суффиксов, вторая строка - добавляет в него нужные суффиксы, используемые в данном make-файле.

Не следует предполагать, что `.' находится в пути поиска исполняемых команд. Когда вам нужно запустить программу, являющуюся частью вашего пакета, используйте `./' если эта программа строится в ходе обработки make-файла или `$(srcdir)/' если запускаемый файл является "неизменной" частью и его можно отнести к исходному коду. Если нет ни одного из этих префиксов, будет использован текущий путь поиска.

Разница между `./' (директория, где происходит сборка проекта) и `$(srcdir)/' (директория с *исходными текстами*) существенна, поскольку пользователь может запустить компиляцию в отдельной директории, указав опцию `--srcdir' для `configure'. Например, правило вида:

```
foo.1 : foo.man sedscript
        sed -e sedscript foo.man > foo.1
```

не будет работать, если сборка проекта происходит не в директории с исходными файлами, поскольку файлы `foo.man' и `sedscript' находятся именно там.

При использовании GNU `make', можно полагаться на `VPATH' при наличии единственного файла пререквизита, поскольку автоматическая переменная `$<' укажет на исходный файл, где бы он ни находился. (Многие версии `make' устанавливают `$<' только для неявных правил.) Таким образом, для корректной работы `VPATH', вместо

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c bar.c -o foo.o
```

следует писать

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c $< -o $@
```

При наличии нескольких пререквизитов, проще всего, явно указывать префикс `$(srcdir)'. Так, приведенный выше пример для `foo.1', лучше всего записать следующим образом:

```
foo.1 : foo.man sedscript
        sed -e $(srcdir)/sedscript $(srcdir)/foo.man > $@
```

При использовании GNU `make', можно полагаться на `VPATH' при наличии единственного файла пререквизита, поскольку автоматическая переменная `$<' укажет на исходный файл, где бы он ни находился. (Многие версии `make' устанавливают `$<' только для неявных правил.) Таким образом, для корректной работы `VPATH', вместо

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c bar.c -o foo.o
```

следует писать

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c $< -o $@
```

При наличии нескольких пререквизитов, проще всего, явно указывать префикс `$(srcdir)'. Так, приведенный выше пример для `foo.1', лучше всего записать следующим образом:

```
foo.1 : foo.man sedscript
        sed -e $(srcdir)/sedscript $(srcdir)/foo.man > $@
```

Пакеты GNU, обычно, содержат некоторые файлы, не являющиеся исходными, например, файлы в формате Info, выходные файлы программ Autoconf, Automake, Bison или Flex. Поскольку, как правило, эти файлы располагаются в директории с исходными текстами, они и должны всегда располагаться там, а не в директории, где происходит сборка проекта. Таким образом, правила make-файла, обновляющие эти файлы, должны помещать их в директорию с исходными текстами.

Однако, если файл не входит в состав дистрибутива, make-файле не должен помещать его в директорию с исходными текстами, поскольку, в обычных условиях, процесс сборки не должен модефицировать каталог с исходными текстами.

Желательно, также, попробовать добиться корректной работы make-файла (по крайней мере, для целей сборки и инсталляции) при запуске `make` в параллельном режиме.

Использование утилит

Команды, указываемые в make-файле (включая скрипты, такие как `configure`), следует писать в рассчете на интерпретатор `sh`, а не `csh`. Не используйте специфические возможности оболочек `ksh` или `bash`.

Скрипт `configure` и указанные в make-файле правила для сборки и инсталляции не должны напрямую использовать никаких утилита, за исключением:

```
cat cmp cp diff echo egrep expr false grep install-info
ln ls mkdir mv pwd rm rmdir sed sleep sort tar test touch true
```

В правиле `dist`, для компрессирования можно использовать программу `gzip`.

Для этих программ следует пользоваться только "общепринятыми" опциями. Например, не используйте `mkdir -p', поскольку эта опция поддерживается лишь в немногих системах.

По возможности, следует избегать создания make-файлом символических связей, поскольку в некоторых системах они не поддерживаются.

Для сборки и инсталляции, make-файл может пользоваться компиляторами и другими необходимыми программами, однако, это следует делать через определенные в `make` переменные, дабы пользователь мог указать свою

альтернативу. Вот некоторые из программ, которые мы имели ввиду:

```
ar bison cc flex install ld ldconfig lex
make makeinfo ranlib texi2dvi yacc
```

Для запуска этих программ следует использовать следующие переменные `make`:

```
$(AR) $(BISON) $(CC) $(FLEX) $(INSTALL) $(LD) $(LDCONFIG) $(LEX)
$(MAKE) $(MAKEINFO) $(RANLIB) $(TEXI2DVI) $(YACC)
```

Если вы используете программы `ranlib` или `ldconfig`, убедитесь, что не произойдет ничего плохо при отсутствии этих программ в системе. Перед выполнением такой команды, выдайте сообщение для пользователя, что отсутствие этих команд не является проблемой, а ошибку, которая может возникнуть при попытке выполнения, проигнорируйте. (Здесь, может помочь макрос `` `AC_PROG_RANLIB' `` программы Autoconf.)

При использовании символических ссылок, вы должны предусмотреть "запасной вариант" для систем, где они не поддерживаются.

Вот список дополнительных утилит, которые можно использовать через соответствующие переменные `make`:

```
chgrp chmod chown mknod
```

Все прочие утилиты возможно использовать во фрагментах make-файлов (или скриптах), используемых только для определенных систем, где эти утилиты действительно существуют.

Переменные для имен команд

Make-файлы должны предоставлять переменные, с помощью которых можно изменять определенные опции, команды и тому подобное.

В частности, большинство утилит нужно запускать с помощью соответствующих переменных. Так, при использовании программы Bison, следует использовать переменную `BISON`, чье значение, по умолчанию, устанавливается как `` `BISON = bison' ``, и использовать переменную `$(BISON)` при каждом запуске Bison.

Вызов утилит, манипулирующих с файлами (таких, как `ln`, `rm`, `mv` и им подобных), может производится без использования переменных, поскольку пользователям нет необходимости заменять эти утилиты другими программами.

Для каждой переменной, содержащая имя программы, должна иметься соответствующая переменная для хранения опций, которые будут передаваться этой программе. Для получения имени этой переменной, добавьте `` `FLAGS' `` к имени перенной, ссылающейся на соответствующую программу. Так, например, опции для `BISON` должны храниться в переменной `BISONFLAGS`. (Имена `CFLAGS` для компилятора Си, `YFLAGS` для программы yacc и `LFLAGS` для lex, являются исключениями, используемыми только в силу их

"стандартности".) Используйте CPPFLAGS для любых команд, использующих препроцессор и переменную LDFLAGS во всех командах, производящих компоновку или компиляцию с последующей компоновкой.

Если имеются какие-то опции компилятора Си, которые *должны* быть использованы для правильной компиляции определенных файлов, не следует включать их в CFLAGS. Пользователи вправе ожидать, что они самостоятельно могут указать любое желаемое значение CFLAGS. Вместо этого, передавайте необходимые опции "в обход" CFLAGS, например, "прямо" указывая их в команде компиляции или используя какую-нибудь другую переменную, наподобие:

```
CFLAGS = -g
ALL_CFLAGS = -I. $(CFLAGS)
.c.o:
        $(CC) -c $(CPPFLAGS) $(ALL_CFLAGS) $<
```

Напротив, опцию `-g' вполне можно включить в CFLAGS, поскольку она *не требуется* для успешной компиляции. Ее можно рассматривать как значение по умолчанию, которое лишь рекомендуется. Если, по умолчанию, предполагается, что пакет будет компилироваться с помощью компилятора GCC, в начальное значение CFLAGS можно также включить опцию `-O'.

Помещайте ссылку на CFLAGS в конец строки с командой компиляции, после других переменных, содержащих опции компиляции, дабы пользователь мог использовать CFLAGS для "перекрытия" этих опций.

Переменная CFLAGS должна использоваться при любом вызове компилятора Си (как для компиляции, так и для компоновки).

Каждый make-файл должен определять переменную INSTALL, которая является базовой командой для инсталляции файлов в систему.

Каждый make-файл должен определять переменные INSTALL_PROGRAM и INSTALL_DATA. (По умолчанию, они должны иметь значение $(INSTALL).) Далее, эти переменные должны использоваться для инсталляции, соответственно, исполняемых и неисполняемых файлов. Эти переменные используются примерно таким способом:

```
$(INSTALL_PROGRAM) foo $(bindir)/foo
$(INSTALL_DATA) libfoo.a $(libdir)/libfoo.a
```

К имени целевого файла вы можете, дополнительно, добавить значение переменной DESTDIR. Сделав это, вы дадите возможность инсталлятору получить "слепок" инсталляции, который, в дальнейшем, может быть использован для реальной инсталляции в систему. Не пытайтесь установить значение DESTDIR самостоятельно, из своего make-файла. С поддержкой DESTDIR, приведенный выше пример можно переписать так:

```
$(INSTALL_PROGRAM) foo $(DESTDIR)$(bindir)/foo
$(INSTALL_DATA) libfoo.a $(DESTDIR)$(libdir)/libfoo.a
```

Никогда не используйте имя каталога в качестве второго аргумента

команды инсталляции - используйте имя конкретного файла. Используйте отдельную команду для каждого инсталлируемого файла.

Переменные для имен каталогов инсталляции

Каталоги, куда происходит инсталляция всегда должны именоваться с использованием переменных, дабы пакет легко мог быть установлен в "нестандартное" место. Ниже приведены стандартные имена для таких переменных. Они базируются на "общепринятой" (в Unix) структуре файловой системы; ее варианты используются в SVR4, 4.4BSD, Linux, Ultrix v4 и других современных операционных системах.

Следующие две переменные определяют корневые каталоги для всего процесса инсталляции. Все прочие каталоги, куда будет происходить инсталляция, должны быть подкаталогами одного из них. Процесс инсталляции не должен помещать каких-либо файлов непосредственно в эти корневые каталоги.

`prefix'
: Префикс, используемый для конструирования значений по умолчанию для перечисленных ниже переменных. По умолчанию, значением `prefix` должно быть `/usr/local'. При построении полной GNU-системы "с нуля", этот префикс будет содержать пустое значение, а `/usr' будет символической ссылкой на `/'. (При использовании Autoconf, записывается как `@prefix@'.) Запуск `make install' со значением `prefix`, отличным от того, который использовался для сборки пакета, *не* должен приводить к перекомпиляции этого пакета.

`exec_prefix'
: Префикс, используемый при конструировании значений по умолчанию для некоторых из перечисленных ниже переменных. По умолчанию, значением `exec_prefix` должно быть `$(prefix)`. (При использовании Autoconf, записывается как `@exec_prefix@'.) В основном, `$(exec_prefix)` используется для каталогов, содержащих машинно-зависимые файлы (например, исполняемые файлы и файлы библиотек), в то время как для остальных каталогов используется `$(prefix)`. Запуск `make install' со значением `exec_prefix`, отличным от того, который использовался для сборки пакета, *не* должен приводить к перекомпиляции этого пакета.

Исполняемые файлы должны инсталлироваться в одну из следующих директорий.

`bindir'
: Каталог для инсталляции исполняемых программ, которые могут быть запущены пользователем. Обычно, это `/usr/local/bin', но вам следует использовать запись `$(exec_prefix)/bin'. (При использовании Autoconf, пишется как `@bindir@'.)

`sbindir'
: Каталог для инсталляции исполняемых программ, которые могут быть запущены из командного интерпретатора, но полезны, в основном,

только для администраторов системы. Обычно, это директория `/usr/local/sbin', но вам следует использовать запись `$(exec_prefix)/sbin'. (При использовании Autoconf, пишется как `@sbindir@'.)

`libexecdir'
: Каталог для инсталляции исполняемых программ, которые запускаются не пользователями, а другими программами. Обычно, это каталог `/usr/local/libexec', но вам следует использовать запись `$(exec_prefix)/libexec'. (При использовании Autoconf, записывается как `@libexecdir@'.)

Файлы данных, используемые программами, можно классифицировать двумя различными способами.

- В ходе "обычной" работы пакета, некоторые файлы могут модефицироваться, другие же никогда не модефицируются (хотя некоторые из них и могут редактироваться пользователем).
- Одни файлы являются архитектурно-независимыми и могут использоваться на любых машинах; другие файлы являются архитектурно-зависимыми и могут использоваться только на машинах одной архитектуры; третьи файлы могут являться специфичными для конкретной машины.

Таким образом, получается шесть возможных случаев. Однако, мы не рекомендуем вам использовать каких-либо архитектурно-зависимых файлов кроме объектных и библиотечных файлов. Все прочие файлы данных, лучше всего, сохранять архитектурно-независимыми и это, в общем, не так уж и сложно.

Вот список переменных, которые должны использоваться в make-файлах для задания каталогов:

`datadir'
: Директория, куда инсталлируются файлы данных, используемые только для чтения и независимые от архитектуры. Обычно, это каталог `/usr/local/share', но вы должны использовать запись `$(prefix)/share'. (При использовании Autoconf, записывается как `@datadir@'.) Ниже рассматривается специальное исключение - каталоги `$(infodir)' и `$(includedir)'.

`sysconfdir'
: Каталог, куда инсталлируются файлы с неизменяемыми данных, относящиеся только к конкретной машине (грубо говоря, файлы конфигурации хоста). Примерами могут служить конфигурационные файлы почтовой программы и сетевой поддержки, `/etc/passwd' и тому подобное. Все файлы, помещаемые в эту директорию, должны быть обычными текстовыми ASCII файлами. Как правило, это каталог `/usr/local/etc', но вам следует использовать запись `$(prefix)/etc'. (При использовании Autoconf, пишется как `@sysconfdir@'.) Не помещайте в эту директорию исполняемых файлов (их, скорее всего, следует поместить в `$(libexecdir)' или `$(sbindir)'). Также, не помещайте в этот каталог файлов, которые модефицируются в ходе "обычной" работы (разумеется, мы не рассматриваем случаи использования программ, предназначенных для изменения конфигурационных файлов).

Изменяемые файлы следует помещать в `$(localstatedir)'.

`sharedstatedir'
: Каталог для инсталляции архитектурно-независимых файлов данных, которые модефицируются в ходе работы пакета. Обычно, это будет `/usr/local/com', но вам следует использовать запись `$(prefix)/com'. (При использовании Autoconf, пишется как `@sharedstatedir@'.)

`localstatedir'
: Каталог для инсталляции файлов с модефицируемыми данными, относящимися к конкретной машине. У пользователей не должно возникать необходимости в модефикации файлов из этого каталога для конфигурирования пакета; информацию о конфигурации следует помещать в отдельные файлы, расположенные в каталогах `$(datadir)' или `$(sysconfdir)'. Переменная `$(localstatedir)' обычно будет содержать значение `/usr/local/var', но вам следует использовать запись `$(prefix)/var'. (При использовании Autoconf, записывается как `@localstatedir@'.)

`libdir'
: Каталог для размещения объектных файлов и библиотек объектных файлов. Не помещайте сюда исполняемые файлы - инсталлируйте их в директорию `$(libexecdir)'. Как правило, значением `libdir` будет `/usr/local/lib', но вам следует использовать запись `$(exec_prefix)/lib'. (При использовании Autoconf, пишется как `@libdir@'.)

`infodir'
: Каталог, куда будут помещаться Info файлы вашего пакета. По умолчанию, это будет `/usr/local/info', но вам следует использовать запись `$(prefix)/info'. (При использовании Autoconf, записывается как `@infodir@'.)

`lispdir'
: Директория, куда помещаются все Lisp-файлы для редактора Emacs, входящие в состав вашего пакета. Как правило, это `/usr/local/share/emacs/site-lisp', но вам следует использовать запись `$(prefix)/share/emacs/site-lisp'. При использовании Autoconf, это записывается как `@lispdir@'. Для того, чтобы запись `@lispdir@' сработала, в вашем `configure.in' должна быть следующая строка:

```
lispdir='${datadir}/emacs/site-lisp'
AC_SUBST(lispdir)
```

`includedir'
: Каталог, куда будут помещаться заголовочные файлы, предназначенные для включения в пользовательские программы с помощью директивы `#include' препроцессора Си. Обычно, это каталог `/usr/local/include', но вам следует использовать запись `$(prefix)/include'. (При использовании Autoconf, записывается как `@includedir@'.) Кроме GCC, большинство других компиляторов не выполняют поиск заголовочных файлов в каталоге `/usr/local/include', так что помещение их в данный каталог имеет смысл только при использовании компилятора GCC. Зачастую, это не является проблемой, поскольку многие библиотеки ориентированы на работу только с GCC. Однако, некоторые библиотеки могут работать и с другими компиляторами, поэтому их заголовочные файлы следует помещать сразу в два места - каталоги `includedir` и `oldincludedir`.

`oldincludedir'
: Каталог для заголовочных файлов `#include', предназначенных для компиляторов, отличных от GCC. Обычно, это каталог `/usr/include'. (При использовании Autoconf, записывается как `@oldincludedir@'.) Указанные в make-файле команды инсталляции должны проверять значение `oldincludedir'. Если это значение пусто, процесс инсталляции не должен пытаться его использовать, а должен отменить "вторую" инсталляцию заголовочных файлов. Пакет не должен замещать заголовочных файлов, находящихся в этой директории, если только они не взяты из этого же пакета. Так, если в пакете Foo имеется заголовочный файл `foo.h', то при инсталляции он должен помещаться в каталог `oldincludedir' только если (1) там нет файла `foo.h' или (2) файл `foo.h' существует и был ранее установлен пакетом Foo. Для проверки того, что `foo.h' был установлен именно пакетом Foo, поместите в этот файл (как часть комментария) строку с определенной сигнатурой, по которой файл можно будет "опознать" с помощью программы `grep'.

Файлы помощи в стиле Unix (man-страницы, man pages) должны размещаться в одном из следующих каталогов:

`mandir'
: Корневая директория для размещения файлов помощи (если такие имеются) вашего пакета. Обычно, это будет `/usr/local/man', но вам следует использовать запись it as `$(prefix)/man'. (При использовании Autoconf, записывается как `@mandir@'.)

`man1dir'
: Каталог для установки man-страниц, относящихся к разделу 1. Следует записывать как `$(mandir)/man1'.

`man2dir'
: Директория для man-страниц, относящихся к разделу 2. Следует записывать как `$(mandir)/man2'

`...'
: **Не используйте формат man-страничек как основной формат документации для программ GNU. Пишите документацию в формате Texinfo. Man-странички пригодны только для людей, использующих программы GNU в Unix-подобных операционных системах.**

`manext'
: Расширение имени файла для инсталлируемых man-страничек. Оно должно начинаться с точки за которой следует соответствующая цифра; обычно, это будет `.1'.

`man1ext'
: Расширение имени файла для инсталляции man-страничек, относящихся к секции 1.

`man2ext'
: Расширение имени файла для инсталляции man-страничек, относящихся к секции 2.

`...'
: Используйте эти имена вместо `manext', если пакет нуждается в инсталляции man-страничек, относящихся к нескольким секциям.

И, наконец, вы должны установить следующую переменную:

`srcdir'
: Каталог с исходными файлами компилируемого пакета. Значение этой переменной, обычно, устанавливается скриптом `configure`. (При использовании Autconf, используйте запись `srcdir = @srcdir@'.)

Пример:

```
# Общий префикс для инсталляционных каталогов.
# Примечание: Этот каталог должен существовать еще до начала инсталляции.
prefix = /usr/local
exec_prefix = $(prefix)
# Сюда будет помещаться исполняемый файл команды `gcc'.
bindir = $(exec_prefix)/bin
# Сюда будут помещаться каталоги, используемые компилятором.
libexecdir = $(exec_prefix)/libexec
# Сюда будут помещаться Info-файлы.
infodir = $(prefix)/info
```

Если ваша программа устанавливает большое количество файлов в один из "стандартных" каталогов, определенных пользователем, можно сгруппировать их в отдельный подкаталог, относящийся к данному пакету. Такие подкаталоги должны создаваться правилом `install`.

Не следует ожидать, что устанавливаемые пользователем значения указанных выше переменных, будут содержать имена конкретных подкаталогов (соответствующих данному пакету). Смысл наличия "унифицированного" набора имен переменных состоит в том, чтобы пользователь мог задать одинаковые значения для нескольких разных пакетов GNU-программ. Пакеты должны быть спроектированы таким образом, чтобы правильно использовать задаваемые пользователем значения.

Стандартные имена целей для пользователей

Все GNU-программы должны иметь в своем make-файле следующие цели:

`all'
: Компиляция всей программы. По умолчанию, именно эта цель должна являться главной целью. Эта цель не должна пересобирать файлы документации; Info-файлы должны иметься в составе дистрибутива, а файлы DVI должны создаваться только по специальному запросу. По умолчанию, компиляция и компоновка должна выполняться с опцией `-g', дабы исполняемые файлы имели отладочную информацию. Пользователи, которые не бояться оказаться в трудной ситуации, могут, при желании, потом убрать эту отладочную информацию.

`install'
: Компиляция программы и копирование исполняемых файлов, файлов библиотек и прочих файлов в те места, где они должны, в действительности, располагаться в системе. При наличии какого-нибудь простого теста, проверяющего правильность инсталляции

пакета, такой тест должен запускаться. Не убирайте отладочную информацию из исполняемых файлов во время их инсталляции. Озабоченные наличием такой информации пользователи, могут использовать `install-strip`. Будучи запущенной сразу после `` `make all' ``, `install`, по возможности, не должна ничего модефицировать в каталоге, где собиралась программа. Дело в том, что сборка программного пакета и его инсталляция очень часто выполняются под разными именами пользователей. Если каталогов, куда инсталлируются файлы пакета, не существует, они должны создаваться (в том числе и каталоги, указанные в `prefix` и `exec_prefix`). Для этого можно использовать правило `installdirs` (описано ниже). Используйте `` `-' `` во всех командах, инсталлирующих man-странички, дабы игнорировать любые ошибки, которые могут возникать в системах, не имеющих документации в формате man-страничек. Файлы Info инсталлируются путем копирования их в каталог `` `$(infodir)' `` при помощи `$(INSTALL_DATA)` (смотрите раздел Переменные для имен команд), с последующим запуском `install-info` (если эта программа задана). Переменная `install-info` содержит имя программы, которая редактирует файл `` `dir' `` (каталог Info-файлов), добавляя или обновляя записи об инсталлированных Info-файлах; эта программа является частью пакета Texinfo. Вот пример правила для инсталляции Info-файла:

```
$(DESTDIR)$(infodir)/foo.info: foo.info
        $(POST_INSTALL)
# В каталоге . может находится более новая версия info-файла, чем в srcdir.
        -if test -f foo.info; then d=.; \
         else d=$(srcdir); fi; \
        $(INSTALL_DATA) $$d/foo.info $(DESTDIR)$@; \
# Запускаем install-info только при его наличии.
# Используем `if' вместо начального `-'
# чтобы не пропустить ошибку, которую может вернуть install-info.
# Мы используем `$(SHELL) -c', поскольку некоторые оболочки
# завершаются не вполне корректно, встретив неизвестную команду.
        if $(SHELL) -c 'install-info --version' \
           >/dev/null 2>&1; then \
          install-info --dir-file=$(DESTDIR)$(infodir)/dir \
                       $(DESTDIR)$(infodir)/foo.info; \
        else true; fi
```

При описании цели `install`, вы должны отнести все ее команды к одной из трех категорий: обычные (normal), команды **пред-инсталляции (pre-installation)** и команды **пост-инсталляции (post-installation)**. Смотрите раздел "Категории" команд инсталляции.

`` `uninstall' ``
: Удалить все инсталлированные файлы - копии файлов, созданные при выполнении `` `install' ``. Это правило не должно модефицировать каталоги, где осуществлялась сборка пакета - можно модефицировать только каталоги с инсталлированными файлами. Команды деинсталляции также, как и команды инсталляции, делятся на три категории. Смотрите раздел "Категории" команд инсталляции.

`` `install-strip' ``
: Аналогично `install`, но в процессе инсталляции из исполняемых файлов удаляется отладочная информация. Во многих случаях, определение этой цели может быть очень простым:

```
install-strip:
```

```
$(MAKE) INSTALL_PROGRAM='$(INSTALL_PROGRAM) -s' \
        install
```

Обычно, мы не рекомендуем удалять из исполняемых файлов отладочную информацию, если только вы не уверены, что программа не содержит ошибок. Однако, вполне разумно инсталлировать исполняемые файлы без отладочной информации, сохранив где-нибудь (на случай ошибки) их "оригинальные" версии с отладочной информацией.

`clean'
: Удалить из текущей директории все файлы, созданные в ходе сборки программы. Файлы с конфигурацией, при этом, удаляться не должны. Не должны также удаляться файлы, которые, хотя и могут быть созданы заново, но, как правило, не создаются, так как имеются в пакете уже в "готовом виде". Напротив, файлы `.dvi' следует удалить, если только они не в входят в дистрибутив пакета.

`distclean'
: Удаляет все файлы из текущей директории, созданные в процессе конфигурировании или сборки пакета. Если вы распаковали исходные файлы и собрали программу, не создавая никаких дополнительных файлов, после команды `make distclean', в каталоге должны остаться только файлы, входящие в исходный дистрибутив пакета.

`mostlyclean'
: Подобно `clean', но может воздержаться от удаления некоторых файлов, которые, обычно, "нежелательно" перекомпилировать. Так, например, `mostlyclean' для компилятора GCC не удаляет файл `libgcc.a', поскольку необходимость в его перекомпиляции возникает редко, а занимает она много времени.

`maintainer-clean'
: Удаляет из текущей директории практически все, что может быть построено с помощью make-файла. В этот список, обычно, входят все файлы, удаляемые с помощью distclean, а также: Си-программы, полученные с помощью Bison, таблицы тегов, Info-файлы и тому подобное. Мы сказали "практически все", потому что команда `make maintainer-clean' не должна удалять скрипт `configure', даже если если для него имеется правило в make-файле, с помощью которого `configure' может быть создан заново. В общем, команда `make maintainer-clean' не должна удалять ничего, что необходимо для запуска `configure' и последующей сборки пакет. Это и является единственным исключением; все остальные файлы, которые могут быть построены заново, команда maintainer-clean должна удалять. Предполагается, что `maintainer-clean' будет использоваться только лицом, сопровождающим данный пакет, а не обычными пользователями. Вполне возможно, что для получения некоторых файлов, удаленных при помощи `make maintainer-clean', могут потребоваться какие-то специальные программы, не включенные в состав дистрибутива. Поскольку, все нужные для сборки пакета файлы входят в дистрибутив, нам нет смысла беспокоиться о том, насколько прост процесс их создания. Так что, если после команды `make maintainer-clean' окажется, что для сборки пакета вам придется заново распаковывать дистрибутив - не жалуйтесь. Для того, чтобы предупредить пользователя о возможных проблемах, правило для цели maintainer-clean должно начинаться

следующими командами:

```
@echo 'This command is intended for maintainers to use; it'
@echo 'deletes files that may need special tools to rebuild.'
```

('Эта команда предназначена для использования только лицом, сопровождающим данный пакет; она удаляет некоторые файлы, для создания которых требуются специальные программы.')

`TAGS'
Обновляет таблицу тегов (tags table) для программы.

`info'
Сгенерировать все необходимые Info-файлы. Один из лучших способов определить подобное правило - использовать запись, наподобие:

```
info: foo.info

foo.info: foo.texi chap1.texi chap2.texi
        $(MAKEINFO) $(srcdir)/foo.texi
```

В make-файле вы должны определить переменную `MAKEINFO`. Она должна запускать программу `makeinfo`, входящую в состав пакета Texinfo. Как правило, пакеты программ GNU имеют в своем дистрибутиве все необходимые Info-файлы, а это означает, что Info-файлы находятся в каталоге с исходными файлами. Таким образом, команды, обновляющие info-файлы, должны обновлять их в директории с исходными файлами. Обычно, при сборке пользователем пакета программ, обновления Info-файлов не происходит, поскольку в дистрибутиве они находятся в "готовом виде" и не нуждаются в обновлении.

`dvi'
Сгенерировать DVI-файлы для всей документации в формате Texinfo. Например:

```
dvi: foo.dvi

foo.dvi: foo.texi chap1.texi chap2.texi
        $(TEXI2DVI) $(srcdir)/foo.texi
```

В make-файле вы должны определить переменную `TEXI2DVI`. Она должна запускать программу `texi2dvi`, которая входит в состав пакета Texinfo. (3) Как альтернатива, в правиле можно перечислить только пререквизиты, предоставив GNU `make` возможность самостоятельно выполнить нужные команды.

`dist'
Создать дистрибутивный tar-файл для этой программы. Tar-файл должен быть построен таким образом, чтобы имена содержащихся в нем файлов начинались с имени каталога, отражающим название пакета, для которого предназначается данный дистрибутив. Это имя может включать в себя номер версии пакета. Например, дистрибутивный tar-файл компилятора GCC версии 1.40 распаковывается в подкаталог с именем `gcc-1.40'. Проще всего это сделать, создав подкаталог с нужным именем и поместить туда нужные файлы с помощью `ln` или `cp`. Далее, для этого каталога надо выполнить команду `tar` и сжать полученный архивный файл программой `gzip`. Например, дистрибутивный файл для GCC версии

1.40 называется `gcc-1.40.tar.gz'. Цель dist должна иметь явные зависимости от всех файлов, которые не являются исходными, но, вместе с тем, должны входить в дистрибутив. В этом случае, будет уверенность, что все такие файлы имеют самую свежую версию и попадут в состав дистрибутива. Смотрите раздел `Making Releases' документа *GNU Coding Standards*.

`check'
: Выполнить самотестирование программы (если оно предусмотрено). Перед запуском тестов, пользователь должен осуществить сборку программы, но не обязан ее инсталлировать; вы должны писать тесты таким образом, чтобы для своей работы они не требовали инсталляции проверяемого пакета.

Там, где это уместно, рекомендуется использовать цели со следующими именами:

`installcheck`
: Проверить правильность инсталляции (если подобный тест предусмотрен). Перед запуском теста, пользователь должен собрать и инсталлировать пакет. Не следует предполагать, что каталог `$(bindir)' находится в пути поиска.

`installdirs`
: В make-файле полезно иметь цель `installdirs', создающую все каталоги (включая и родительские), которые используются для инсталляции файлов пакета. В составе пакета Texinfo имеется скрипт `mkinstalldirs', который удобно использовать для подобной цели. Используя этот скрипт, соответствующее правило можно записать так:

```
# Убедиться в том, что все требуемые для инсталляции каталоги (например, $(bindir))
# действительно существуют и, при необходимости, создать их.
installdirs: mkinstalldirs
        $(srcdir)/mkinstalldirs $(bindir) $(datadir) \
                                $(libdir) $(infodir) \
                                $(mandir)
```

: Подобное правило не должно модефицировать каталогов, где осуществлялась сборка пакета. Это правило не должно делать ничего, кроме создания каталогов для инсталляции.

"Категории" команд инсталляции

При описании цели `install`, вы должны отнести указываемые команды к одной из трех категорий: обычные (normal) команды, команды **пред-инсталляции (pre-installation)** и команды **пост-инсталляции (post-installation)**.

"Обычные" команды помещают файлы в нужное место и устанавливают для них необходимые аттрибуты доступа. Они не должны модефицировать никаких файлов, кроме тех, которые взяты из данного пакет.

Команды пред-инсталляции и пост-инсталляции могут изменять другие файлы; в частности, модефицировать "глобальные" конфигурационные файлы или файлы записей.

Обычно, команды пред-инсталляции исполняются до "обычных" команд инсталляции, а команды пост-инсталляции - после "обычных" команд.

Чаще всего, команды пост-инсталляции используются для запуска `install-info`. Для этого не могут быть использованы обычные команды, поскольку при этом изменяется содержимое файла (Info-директории), который не является частью инсталлируемого пакета. Для этого нужно использовать команды пост-инсталляции, потому что подобное действие должно выполняться после завершения работы "обычных" команд, которые инсталлируют необходимые Info-файлы пакета.

Большинство программ не нуждается в командах пред-инсталляции, но, на всякий случай, мы предусмотрели и подобную возможность.

Для того, чтобы отнести команды правила `install` к одной из трех возможных категорий, перед ними надо вставить соответствующую строку с именем категории (**category lines**). Эта строка задаст категорию для следующих за нею команд.

Строка с указанием категории состоит из символа табуляции, за которым следует ссылка на специальную переменную; в конце строки может находится комментарий. Всего существует три таких специальных переменных - по одной на каждую категорию команд; каждое имя указывает свою категорию. При "обычном" исполнении, строки с категориями не выполняют никаких операций, поскольку соответствующие переменные, обычно, не определены (и вы *не должны* определять их в своем make-файле).

Ниже перечислены все три возможных категории:

```
$(PRE_INSTALL)     # Далее следуют команды пред-инсталляции.
$(POST_INSTALL)    # Далее следуют команды пост-инсталляции.
$(NORMAL_INSTALL)  # Далее следуют "обычные" команды.
```

При отсутствии строки с категорией в начале правила `install`, все командные строки от начала правила до первой строки с категорией, будут рассматриваться как "обычные". При отсутствии каких-либо строк с категориями, все команды правила будут рассматриваться как "обычные".

Аналогично, имеется три категории команд для правила `uninstall`:

```
$(PRE_UNINSTALL)     # Далее следуют команды пред-деинсталляции.
$(POST_UNINSTALL)    # Далее следуют команды пост-деинсталляции.
$(NORMAL_UNINSTALL)  # Далее следуют "обычные" команды.
```

Обычно, команды пред-деинсталляции используются для удаления записей (о деинсталлируемых Info-файлах) из Info-директории.

При наличии у целей `install` или `uninstall` любых пререквизитов, работающих как "подпрограммы" инсталляции, для *каждой* строки с командой в этих пререквизитах должна быть указана ее категория. Команды для "главной" цели, также должны начинаться со строки категории. Только в этом случае можно быть уверенным, что все команды попадут в нужную категорию,

независимо от того, какие из пререквизитов обрабатывались.

Команды пред-инсталляции и пост-инсталляции не должны запускать каких-либо программ, кроме:

```
[ basename bash cat chgrp chmod chown cmp cp dd diff echo
egrep expand expr false fgrep find getopt grep gunzip gzip
hostname install install-info kill ldconfig ln ls md5sum
mkdir mkfifo mknod mv printenv pwd rm rmdir sed sort tee
test touch true uname xargs yes
```

Необходимость разделения команд на категории возникает из-за необходимости уметь инсталлировать "бинарные" дистрибутивы пакетов. Как правило, такой пакет содержит все необходимые файлы (в том числе и исполняемые) в готовом виде и использует свой собственный метод их инсталляции. Таким образом, ему не требуется запускать "обычные" команды инсталляции. Однако, при инсталляции бинарного пакета, нужно выполнить команды пред-инсталляции и пост-инсталляции.

Программы для сборки бинарных пакетов работают, "извлекая" команды пред-инсталляции и пост-инсталляции. Вот один из возможных способов извлечения команд пред-инсталляции:

```
make -n install -o all \
      PRE_INSTALL=pre-install \
      POST_INSTALL=post-install \
      NORMAL_INSTALL=normal-install \
  | gawk -f pre-install.awk
```

Здесь, файл `pre-install.awk' может содержать что-нибудь, наподобие:

```
$0 ~ /^\t[ \t]*(normal_install|post_install)[ \t]*$/ {on = 0}
on {print $0}
$0 ~ /^\t[ \t]*pre_install[ \t]*$/ {on = 1}
```

В результате получается файл с командами пред-инсталляции, который, затем, выполняется командным интерпретатором в процессе инсталляции бинарного пакета.

Справочник

В данном приложении перечислены все директивы, функции и специальные переменные, распознаваемые программой GNU make. Смотрите также разделы Имена специальных целей, Перечень имеющихся неявных правил, и Обзор опций.

Вот перечень директив, распознаваемых GNU make:

`define` *имя_переменной*
`endef`

Определить многострочную, рекурсивно-вычисляемую переменную. Смотрите раздел Именованные командные последовательности.

`ifdef` *переменная*
`ifndef` *переменная*

`ifeq (`*a*`,`*b*`)`
`ifeq "`*a*`" "`*b*`"`
`ifeq '`*a*`' '`*b*`'`
`ifneq (`*a*`,`*b*`)`
`ifneq "`*a*`" "`*b*`"`
`ifneq '`*a*`' '`*b*`'`
`else`
`endif`

Условно обработать часть make-файла.
Смотрите раздел Условные части make-файла.

`include` *файл*
`-include` *файл*
`sinclude` *файл*

Подключить другой make-файл.
Смотрите раздел Подключение других make-файлов.

`override` *имя_переменной* `=` *значение*
`override` *имя_переменной* `:=` *значение*
`override` *имя_переменной* `+=` *значение*
`override` *имя_переменной* `?=` *значение*
`override define` *имя_переменной*
`endef`

Определить переменную с "перекрытием" предыдущего ее определения (в том числе и заданного с помощью командной строки).
Смотрите раздел Директива `override`.

`export`

Инструктирует `make` экспортировать, по умолчанию, все переменные в дочерние процессы.
Смотрите раздел Связь с `make` "нижнего уровня" через переменные.

`export` *переменная*
`export` *переменная* `=` *значение*
`export` *переменная* `:=` *значение*
`export` *переменная* `+=` *значение*
`export` *переменная* `?=` *значение*
`unexport` *переменная*

Включить или отключить экспорт конкретной переменной в дочерние процессы.
Смотрите раздел Связь с `make` "нижнего уровня" через переменные

`vpath` *шаблон путь*

Задать путь поиска для файлов, подходящих под указанный шаблон (с символом `%`).
Смотрите раздел Директива `vpath`.

`vpath` *шаблон*

Удалить все пути поиска, заданные ранее для указанного шаблона.

`vpath`

Удалить все пути поиска, заданные ранее с помощью директив `vpath`.

Далее приведен список функций, манипулирующих с текстом (смотрите раздел Функции преобразования текста):

`$(subst` *исходная_строка*`,`*конечная_строка*`,`*текст*`)`

Заменить *исходную_строку* на *конечную_строку* в указанном *тексте*.

Смотрите раздел Функции анализа и подстановки строк.

`$(patsubst шаблон,замена,текст)`

Заменить все слова, подходящие под *шаблон*, на *замену* в указанном *тексте*.

Смотрите раздел Функции анализа и подстановки строк.

`$(strip строка)`

Удалить из указанной *строки* избыточные пробелы.

Смотрите раздел Функции анализа и подстановки строк.

`$(findstring искомая_строка,текст)`

Найти *искомую_строку* в указанном *тексте*.

Смотрите раздел Функции анализа и подстановки строк.

`$(filter шаблон...,текст)`

Выбрать из указанного *текста* все слова, подходящие под *шаблон(ы)*.

Смотрите раздел Функции анализа и подстановки строк.

`$(filter-out шаблон...,текст)`

Выбрать из *текста* слова, *не подходящие* ни под один из указанных *шаблонов*.

Смотрите раздел Функции анализа и подстановки строк.

`$(sort список)`

Отсортировать слова из *списка* в лексикографическом порядке, удалив дубликаты.

Смотрите раздел Функции анализа и подстановки строк.

`$(dir имена...)`

Из указанных имен файлов выделяет части, определяющие имена каталогов.

Смотрите раздел Функции для обработки имен файлов.

`$(notdir имена...)`

Из каждого указанного имени файла выделить часть, которая не является именем каталога.

Смотрите раздел Функции для обработки имен файлов.

`$(suffix имена...)`

Из каждого имени файла выделить суффикс (последний `.' и следующие за ним символы).

Смотрите раздел Функции для обработки имен файлов.

`$(basename имена...)`

Выделяет "базовое имя" (часть имени без суффикса) из каждого имени файла.

Смотрите раздел Функции для обработки имен файлов.

`$(addsuffix суффикс,имена...)`

Добавляет *суффикс* к каждому из указанных *имен*.

Смотрите раздел Функции для обработки имен файлов.

`$(addprefix префикс,имена...)`

Предваряет *префиксом* каждое из указанных *имен*.

Смотрите раздел Функции для обработки имен файлов.

`$(join список1,список2)`

Объединяет два "параллельных" списка слов.

Смотрите раздел Функции для обработки имен файлов.

`$(word n,текст)`

Выделяет *n*-ное слово (счет начинается с единицы) из указанного *текста*.

Смотрите раздел Функции для обработки имен файлов.

`$(words` *текст*`)`
: Подсчитывает количество слов в указанном *тексте*.
Смотрите раздел Функции для обработки имен файлов.

`$(wordlist` *от*`,`*до*`,`*текст*`)`
: Возвращает список слов из указанного *текста* с номерами *от* и *до*.
Смотрите раздел Функции для обработки имен файлов.

`$(firstword` *имена*`...)`
: Выделяет первое слово из указанного списка *имен*.
Смотрите раздел Функции для обработки имен файлов.

`$(wildcard` *шаблон*`...)`
: Получить список файлов, чьи имена подходят под указанный шаблон (шаблон записывает в формате интерпретатора командной строки, а *не* `%'-шаблона).
Смотрите раздел Функция `wildcard`.

`$(error` *сообщение*`...)`
: При вычислении этой функции, `make` генерирует фатальную ошибку с указанным *сообщением*.
Смотрите раздел Функции управления сборкой.

`$(warning` *сообщение*`...)`
: При вычислении этой функции, `make` выдает предупреждение (warning) с указанным *сообщением*.
Смотрите раздел Функции управления сборкой.

`$(shell` *команда*`)`
: Выполнить команду оболочки и возвратить выведенную ей информацию.
Смотрите раздел Функция `shell`.

`$(origin` *переменная*`)`
: Вернуть строку, описывающую, каким образом была определена переменная `make` *переменная*.
Смотрите раздел Функция `origin`.

`$(foreach` *переменная-аргумент*`,`*слова*`,`*текст*`)`
: Вычислить значение *текста*, поочередно подставляя в *переменную-аргумент* слова из списка *слова*, и объединить полученные результаты.
Смотрите раздел Функция `foreach`.

`$(call` *переменная*`,`*параметр*`,...)`
: Вычислить и вернуть значение *переменной*, заменяя ссылки `$(1)`, `$(2)` на значения первого, второго и так далее *параметров*.
Смотрите раздел Функция `call`.

Вот список имеющихся автоматических переменных. Смотрите раздел Автоматические переменные, где они описаны подробно.

`$@`
: Имя файла цели.

`$%`
: Имя элемента для целей, являющихся элементами архива.

`$<`
: Имя первого пререквизита из списка пререквизитов

`$?`
: Имена всех пререквизитов (разделенные пробелами), которые

являются более новыми, чем цель. Для пререквизитов, являющихся элементами архивов, используются только имена элементов (смотрите раздел Использование make для обновления архивов).

`$^`

`$+`

Имена всех пререквизитов, разделенные пробелами. Для пререквизитов, являющихся элементами архивов, используются только имена элементов (смотрите раздел Использование make для обновления архивов). Значение переменной `$^` не содержит дубликатов пререквизитов, в то время как `$+` сохраняет дубликаты и "оригинальный" порядок следования пререквизитов.

`$*`

Основа имени (stem), с которой было сопоставлено неявное правило (смотрите раздел Процедура сопоставления с шаблоном).

`$(@D)`

`$(@F)`

Части имени `$@`, которые представляют собой имя каталога и имя файла внутри каталога, соответственно.

`$(*D)`

`$(*F)`

Части имени `$*`, которые представляют собой имя каталога и имя файла внутри каталога, соответственно.

`$(%D)`

`$(%F)`

Части имени `$%`, которые представляют собой имя каталога и имя файла внутри каталога, соответственно.

`$(<D)`

`$(<F)`

Части имени `$<`, которые представляют собой имя каталога и имя файла внутри каталога, соответственно.

`$(^D)`

`$(^F)`

Части имени `$^`, которые представляют собой имя каталога и имя файла внутри каталога, соответственно.

`$(+D)`

`$(+F)`

Части имени `$+`, которые представляют собой имя каталога и имя файла внутри каталога, соответственно.

`$(?D)`

`$(?F)`

Части имени `$?`, которые представляют собой имя каталога и имя файла внутри каталога, соответственно.

Следующие переменные GNU `make` использует специальным образом:

`MAKEFILES`

Make-файлы, которые будут считываться при каждом вызове `make`. Смотрите раздел Переменная `MAKEFILES`.

`VPATH`

Путь поиска для файлов, которые не могут быть найдены в текущем каталоге.

Смотрите раздел Переменная VPATH: список каталогов для поиска пререквизитов.

SHELL
: Имя используемого по умолчанию командного интерпретатора, обычно `/bin/sh'. В своем make-файле вы можете установить для переменной SHELL новое значение, изменив, тем самым, командный интерпретатор, который будет использоваться для вызова команд. Смотрите раздел Исполнение команд.

MAKESHELL
: Используется только при работе в системе MS-DOS и содержит имя командного интерпретатора, который будет использоваться make. Это значение имеет приоритет перед значением переменной SHELL. Смотрите раздел Исполнение команд.

MAKE
: Имя, с помощью которого была вызвана make. Использование этой переменной в командах имеет специальное значение. Смотрите раздел Как работает переменная MAKE.

MAKELEVEL
: Текущий "уровень вложенности" make при рекурсивном вызове. Смотрите раздел Связь с make "нижнего уровня" через переменные.

MAKEFLAGS
: Опции, заданные для make. Вы можете установить эту переменную из операционной среды или присвоить ей нужное значение внутри make-файла. Смотрите раздел Передача опций в make "нижнего уровня". *Никогда* не задавайте значение MAKEFLAGS непосредственно в командной строке: в командный интерпретатор это значение может быть передано некорректно. Никогда не препятствуйте рекурсивно вызванным экземплярам make в получении значения этой переменной через операционную среду.

MAKECMDGOALS
: Цели, заданные make в командной строке. Присваивание этой переменной другого значения не влияет на работу make. Смотрите раздел Аргументы для задания главной цели.

CURDIR
: Имя текущего рабочего каталога (после того, как были обработаны все опции -C, если такие были заданы). Присваивание этой переменной другого значения, не влияет на работу make. Смотрите раздел Рекурсивный вызов make.

SUFFIXES
: Список используемых по умолчанию (до того, как make начнет интерпретировать make-файлы) суффиксов для суффиксных правил.

.LIBPATTERNS
: Определяет способ именования и порядок следования библиотек, поиск которых проводит make. Смотрите раздел Поиск в каталогах для подключаемых библиотек.

Сообщения об ошибках

Далее, приведен список наиболее распространенных ошибок, выдаваемых

программой make, с информации о том, что они означают, и как их можно исправить.

Иногда, ошибки make не являются фатальными, особенно при наличии префикса - в командах или опции -k в аргументах командной строки. Ошибки, являющиеся фатальными, предваряются строчкой ***.

Сообщения об ошибки также предваряются именем программы (обычно, `make'), или, если ошибка произошла в конкретном make-файле, то именем этого файла и номером строки, где была обнаружена ошибка.

В приведенной ниже таблице такого рода общие "префиксы" опущены.

`[*foo*] Error *NN*'

`[*foo*] *описание сигнала*'

Эти ошибки не являются ошибками make. Они означают, что программа, запущенная make в качестве команды, вернула ненулевой код возврата (`Error *NN*'), который интерпретируется make как ошибка, либо эта программа была аварийно завершена (например, получив соответствующий сигнал). Смотрите раздел Ошибки при исполнении команд. Отсутствие в начале сообщения об ошибке строки *** означает, что make игнорировала данную ошибку из-за наличия в команде префикса -.

`missing separator. Stop.'

`missing separator (did you mean TAB instead of 8 spaces?). Stop.'

Данные сообщения означают, что make не смогла разобраться в "типе" очередной считанное строки. Для определения типа очередной строки, GNU make проверяет наличие в ней нескольких возможных разделителей (:, =, символ табуляции и так далее). Данное сообщение означает, что не найден ни один из возможных разделителей. Зачастую, появление таких ошибок связано с использованием текстовых редакторов, которые, при выравнивании строк, вместо символа табуляции используют последовательность пробелов (этим отличаются многие текстовые редакторы для MS-Windows). В таких случаях make будет использовать второй вариант сообщения об ошибке. Помните, что каждая команда в make-файле должна начинаться с символа табуляции. Восемь пробелов не заменяют один символ табуляции. Смотрите раздел Синтаксис правил.

`commands commence before first target. Stop.'

`missing rule before commands. Stop.'

Это означает, что make-файл, похоже, начинается с командных строк: в начале идет символ табуляции, а следующая за ним конструкция не является допустимой командой make (такой, например, как оператор присваивания). Командные строки всегда должны относиться к какой-то цели. Второй вариант сообщения об ошибке используется в том случае, когда первым непробельным символом в строке является двоеточие; make интерпретирует эту ситуацию как отсутствие левой части правила "цель: пререквизит". Смотрите раздел Синтаксис правил.

`No rule to make target `*xxx*'.'

`No rule to make target `*xxx*', needed by `*yyy*'.'

Это означает, что программа make решила обновить указанную цель, но

не может найти никаких подходящих для этого правил (ни явных, ни неявных, включая встроенные неявные правила). Если вы хотите, чтобы этот файл был создан, вам нужно добавить в make-файл соответствующее правило, описывающее процесс достижения подобной цели. Зачастую, такая ошибка является следствием простой описки (неправильно записанного имени файла) или повреждения каталога с исходными файлами (когда `make` попытается построить недостающие исходные файлы).

`` `No targets specified and no makefile found. Stop.' ``

`` `No targets. Stop.' ``

Первое сообщение означает, что вы не задали никаких целей в командной строке и `make` не может найти ни одного make-файла для обработки. Второе сообщение означает, что был найден некоторый make-файл, однако он не содержит ни одной цели, которую можно было бы выбрать по умолчанию, а в командной строке вы тоже не указали ни одной цели. В такой ситуации GNU `make` не может сделать ничего полезного. Смотрите раздел Аргументы для задания make-файла.

`` `Makefile `xxx' was not found.' ``

`` `Included makefile `xxx' was not found.' ``

Программа `make` не может найти make-файл, указанный в командной строке (сообщение первого вида), или подключаемый make-файл (сообщение второго вида).

`` `warning: overriding commands for target `xxx'' ``

`` `warning: ignoring old commands for target `xxx'' ``

Для каждой цели, GNU `make` позволяет задать лишь один набор команд (исключением являются правила с двойным двоеточием). Такое предупреждающее сообщение выдается при попытке задать команды для цели, которая ранее уже была определена как имеющая команды; при этом, первый набор команд будет "перекрыт" вторым заданным набором. Смотрите раздел Несколько правил с одной целью.

`` `Circular xxx <- yyy dependency dropped.' ``

Это означает, что `make` обнаружила циклическую зависимость: пререквизит *yyy* цели *xxx* (возможно, через цепочку других пререквизитов) опять зависит от цели *xxx*.

`` `Recursive variable `xxx' references itself (eventually). Stop.' ``

Означает, что вы определили обычную (рекурсивно вычисляемую) переменную *xxx*, которая, при попытке ее вычисления, ссылается на саму себя (*xxx*). Это является ошибкой; вам нужно использовать либо упрощенно-вычисляемую переменную (`:=`), либо оператор добавления (`+=`). Смотрите раздел Использование переменных.

`` `Unterminated variable reference. Stop.' ``

Означает, что вы забыли указать закрывающиеся круглые или фигурные скобки при ссылке на переменную или функцию.

`` `insufficient arguments to function `xxx'. Stop.' ``

Это означает, что вы не задали необходимого числа параметров для указанной функции. Смотрите документацию на эту функцию, где указан список требуемых аргументов. Смотрите раздел Функции преобразования текста.

`` `missing target pattern. Stop.' ``

`` `multiple target patterns. Stop.' ``

`target pattern contains no `%'. Stop.'
: Эти ошибки возникают при неправильном составлении статических шаблонных правил. Первая ошибка означает, что в правиле нет шаблона цели. Вторая ошибка означает, что в разделе целей указано сразу несколько шаблонов. Третья ошибка означает, что имя цели не содержит шаблонного символа (%). Смотрите раздел Синтаксис статических шаблонных правил.

`warning: -jN forced in submake: disabling jobserver mode.'
: Это и описанное ниже сообщение генерируется в том случае, если make обнаруживает проблему, связанную с "параллельным" режимом работы в системе, где такой режим поддерживается и make "нижнего уровня" могут "общаться" между собой (смотрите раздел Передача опций в make "нижнего уровня"). Данное предупреждение генерируется в том случае, если make "нижнего уровня" была запущена с аргументом `-jN' (где *N* больше единицы). Такое может произойти, например, если вы самостоятельно установите переменную среды MAKE и присвоите ей значение `make -j2'. В этом случае, рекурсивно вызванные копии make не будут пытаться "связаться" с другими копиями make, а будут работать исходя из того, что они могут запускать не более двух заданий одновременно.

`warning: jobserver unavailable: using -j1. Add `+' to parent make rule.'
: Для того, чтобы запущенные копии make могли "общаться" между собой, родительская копия передает в дочернюю копию некоторую дополнительную информацию. Это может вызвать проблемы, если дочерний процесс не является, на самом деле, программой make. Поэтому, родительский процесс make передает эту дополнительную информацию в дочерний процесс, если только он уверен, что дочерний процесс действительно является программой make. Для этого используется "обычный" алгоритм (смотрите раздел Как работает переменная MAKE). Если make-файл построен таким образом, что родительская копия make не уверена, что дочерний процесс является программой make, в дочерний процесс будет передана лишь часть необходимой информации. В этом случае, дочерний процесс выдает предупреждающее сообщение и выполняет работу в "последовательном" режиме.

Пример "сложного" make-файла

Вот пример make-файла, используемого для сборки программы GNU tar. Это - относительно сложный make-файл.

По умолчанию, главной целью становится первая цель `all'. Интересной особенностью данного make-файла является то, что исходный файл `testpad.h' автоматически создается программой testpad, которая, в свою очередь, компилируется из `testpad.c'.

При выполнении команды `make' или `make all', будет создан исполняемый файл `tar', демон `rmt', обеспечивающий удаленный доступ и Info-файл `tar.info'.

При выполнении команды `make install', make не только создаст файлы `tar', `rmt', и `tar.info', но и инсталлирует их в систему.

При выполнении `make clean', make удаляет все файлы `.o', а также файлы `tar', `rmt', `testpad', `testpad.h', и `core'.

При выполнении команды `make distclean', помимо файлов, удаляемых в `make clean', make также удалит файлы `TAGS', `Makefile', и `config.status'. (В данном случае, make-файл (и файл `config.status') генерируется пользователем с помощью программы configure, которая входит в дистрибутив tar. Эта программа здесь не показана.)

При выполнении `make realclean', наряду с файлами, удаляемыми по команде `make distclean', make также удаляет Info-файлы, полученные из `tar.texinfo'.

В make-файле также имеются две цели для создания дистрибутивов: shar и dist.

```
# Сгенерирован автоматически из Makefile.in с помощью configure.
# Un*x make-файл для программы GNU tar.
# Copyright (C) 1991 Free Software Foundation, Inc.

# This program is free software; you can redistribute
# it and/or modify it under the terms of the GNU
# General Public License ...
...
...

SHELL = /bin/sh

#### Начало раздела конфигурации. ####

srcdir = .

# При использовании gcc, вы должны либо запустить скрипт
# fixincludes, поставляемый с этим компилятором, либо использовать
# gcc с опцией -traditional.  Иначе, вызов ioctl
# может быть неправильно скомпилирован в некоторых системах.
CC = gcc -O
YACC = bison -y
INSTALL = /usr/local/bin/install -c
INSTALLDATA = /usr/local/bin/install -c -m 644

#  Опции, которые можно задать для DEFS:
# -DSTDC_HEADERS        Если у вас имеются стандартные ANSI C заголовочные
#                       файлы и библиотеки.
# -DPOSIX               Если у вас имеются стандартные POSIX.1 заголовочные
#                       файлы и библиотеке.
# -DBSD42               Если у вас имеется sys/dir.h (кроме случаев,
#                       когда вы используете -DPOSIX), sys/file.h,
#                       и st_blocks в `struct stat'.
# -DUSG                 Если у имеются System V/ANSI C
#                       функции для работы со строками и памятью,
#                       соответствующие заголовочные файлы, sys/sysmacros.h,
#                       fcntl.h, getcwd, нет valloc
#                       и есть ndir.h (если вы только не
#                       используете -DDIRENT).
# -DNO_MEMORY_H         В случаях, когда USG или STDC_HEADERS, но
#                       не включать файл memory.h.
# -DDIRENT              В случае USG, но вместо ndir.h у вас
```

```
#                       имеется dirent.h.
# -DSIGTYPE=int         Если ваши обработчики сигналов
#                       возвращают int, а не void.
# -DNO_MTIO             Если у вас нет sys/mtio.h
#                       (ioctl для магнитной ленты).
# -DNO_REMOTE           Если у вас нет удаленной оболочки или
#                       rexec.
# -DUSE_REXEC           Использовать rexec для удаленных операций с
#                       лентой вместо запуска
#                       rsh или remsh через fork.
# -DVPRINTF_MISSING     Если ваша система не имеет функции vprintf
#                       (но имеет _doprnt).
# -DDOPRNT_MISSING      Если ваша система не имеет функции _doprnt.
#                       Надо также определить
#                       -DVPRINTF_MISSING.
# -DFTIME_MISSING       Если ваша система не поддерживает системный вызов ftime.
# -DSTRSTR_MISSING      Если ваша система не имеет функции strstr.
# -DVALLOC_MISSING      Если ваша система не имеет функции valloc.
# -DMKDIR_MISSING       Если ваша система не поддерживает системых вызовов mkdir
#                       и rmdir.
# -DRENAME_MISSING      Если ваша система не поддерживает системный вызов rename.
# -DFTRUNCATE_MISSING   Если система не поддерживает системный вызов ftruncate.
# -DV7                  Для Version 7 Unix (давно не
#                       тестировалось).
# -DEMUL_OPEN3          Если у вас нет версии open с тремя аргументами
#                       и вы хотите, чтобы эта функция эмулировалась
#                       с помощью имеющихся у вас системных вызовов.
# -DNO_OPEN3            Если у вас нет версии open с тремя аргументами
#                       и вы хотите отключить опцию tar -k
#                       вместо того, чтобы эмулировать нужную функцию open.
# -DXENIX               Если у вас имеется sys/inode.h
#                       и нужно, чтобы он был 94 для своего подключения.

DEFS =  -DSIGTYPE=int -DDIRENT -DSTRSTR_MISSING \
        -DVPRINTF_MISSING -DBSD42
# Установите для этой переменной значение rtapelib.o или, если вы определили NO_REMOTE,
# установите для нее пустое значение.
RTAPELIB = rtapelib.o
LIBS =
DEF_AR_FILE = /dev/rmt8
DEFBLOCKING = 20

CDEBUG = -g
CFLAGS = $(CDEBUG) -I. -I$(srcdir) $(DEFS) \
        -DDEF_AR_FILE=\"$(DEF_AR_FILE)\" \
        -DDEFBLOCKING=$(DEFBLOCKING)
LDFLAGS = -g

prefix = /usr/local
# Префикс для каждой инсталлируемой программы,
# обычно пустой или содержит `g'.
binprefix =

# Каталог, куда будет инсталлироваться программа tar.
bindir = $(prefix)/bin

# Каталог, куда будут инсталлироваться Info-файлы.
infodir = $(prefix)/info

#### Конец раздела конфигурации. ####

SRC1 =  tar.c create.c extract.c buffer.c \
        getoldopt.c update.c gnu.c mangle.c
SRC2 =  version.c list.c names.c diffarch.c \
        port.c wildmat.c getopt.c
```

```
SRC3 =  getopt1.c regex.c getdate.y
SRCS =  $(SRC1) $(SRC2) $(SRC3)
OBJ1 =  tar.o create.o extract.o buffer.o \
        getoldopt.o update.o gnu.o mangle.o
OBJ2 =  version.o list.o names.o diffarch.o \
        port.o wildmat.o getopt.o
OBJ3 =  getopt1.o regex.o getdate.o $(RTAPELIB)
OBJS =  $(OBJ1) $(OBJ2) $(OBJ3)
AUX =   README COPYING ChangeLog Makefile.in  \
        makefile.pc configure configure.in \
        tar.texinfo tar.info* texinfo.tex \
        tar.h port.h open3.h getopt.h regex.h \
        rmt.h rmt.c rtapelib.c alloca.c \
        msd_dir.h msd_dir.c tcexparg.c \
        level-0 level-1 backup-specs testpad.c

all:    tar rmt tar.info

tar:    $(OBJS)
        $(CC) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)

rmt:    rmt.c
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ rmt.c

tar.info: tar.texinfo
        makeinfo tar.texinfo

install: all
        $(INSTALL) tar $(bindir)/$(binprefix)tar
        -test ! -f rmt || $(INSTALL) rmt /etc/rmt
        $(INSTALLDATA) $(srcdir)/tar.info* $(infodir)

$(OBJS): tar.h port.h testpad.h
regex.o buffer.o tar.o: regex.h
# getdate.y has 8 shift/reduce conflicts.

testpad.h: testpad
        ./testpad

testpad: testpad.o
        $(CC) -o $@ testpad.o

TAGS:   $(SRCS)
        etags $(SRCS)

clean:
        rm -f *.o tar rmt testpad testpad.h core

distclean: clean
        rm -f TAGS Makefile config.status

realclean: distclean
        rm -f tar.info*

shar: $(SRCS) $(AUX)
        shar $(SRCS) $(AUX) | compress \
          > tar-`sed -e '/version_string/!d' \
                     -e 's/[^0-9.]*\([0-9.]*\).*/\1/' \
                     -e q
                     version.c`.shar.Z

dist: $(SRCS) $(AUX)
        echo tar-`sed \
             -e '/version_string/!d' \
             -e 's/[^0-9.]*\([0-9.]*\).*/\1/' \
```

```
             -e q
             version.c` > .fname
        -rm -rf `cat .fname`
        mkdir `cat .fname`
        ln $(SRCS) $(AUX) `cat .fname`
        tar chZf `cat .fname`.tar.Z `cat .fname`
        -rm -rf `cat .fname` .fname

tar.zoo: $(SRCS) $(AUX)
        -rm -rf tmp.dir
        -mkdir tmp.dir
        -rm tar.zoo
        for X in $(SRCS) $(AUX) ; do \
            echo $$X ; \
            sed 's/$$/^M/' $$X \
            > tmp.dir/$$X ; done
        cd tmp.dir ; zoo aM ../tar.zoo *
        -rm -rf tmp.dir
```

Индекс

#

$

%

*

+

,

-

:

=

?

@

[

\

_

a

b

C

d

e

f

g

h

i

j

k

l

m

n

o

p

q

r

S

t

u

v

w

y

~

Индекс: функции, переменные и директивы

$

%

*

+

.

/

<

- < (автоматическая переменная)
- <D (автоматическая переменная)
- <F (автоматическая переменная)

?

- ? (автоматическая переменная)
- ?D (автоматическая переменная)
- ?F (автоматическая переменная)

@

- @ (автоматическая переменная)
- @D (автоматическая переменная)
- @F (автоматическая переменная)

^

- ^ (автоматическая переменная)
- ^D (автоматическая переменная)
- ^F (автоматическая переменная)

a

- addprefix
- addsuffix
- AR
- ARFLAGS
- AS
- ASFLAGS

b

- basename

c

- call
- CC
- CFLAGS
- CO
- COFLAGS
- COMSPEC
- CPP
- CPPFLAGS
- CTANGLE

d

e

f

g

i

j

l

- LDFLAGS
- LEX
- LFLAGS

m

- MAKE, MAKE
- MAKECMDGOALS
- Makefile
- makefile
- MAKEFILES, MAKEFILES
- MAKEFLAGS
- MAKEINFO
- MAKELEVEL, MAKELEVEL
- MAKEOVERRIDES
- MFLAGS

n

- notdir

o

- origin
- OUTPUT_OPTION
- override

p

- patsubst, patsubst
- PC
- PFLAGS

r

- RFLAGS
- RM

s

- shell
- SHELL
- SHELL (исполнение команд)

t

u

v

w

y

Примечания

(1)

Программа GNU Make, скомпилированная для работы в системах MS-DOS и MS-Windows, ведет себя так, как если бы *prefix* являлся корневым каталогом компилятора DJGPP.

(2)

В системе MS-DOS, значение текущей рабочей директории является **глобальным** и его изменение *будет* влиять на выполнение последующих командных строк.

(3)

Для выполнения "реальной" работы по форматированию, программа `texi2dvi` использует TeX. TeX не входит в состав дистрибутива Texinfo.

This document was generated on 19 July 2000 using the texi2html translator version 1.54.