Введение в Makefile

2.1 Как выглядит правило

Простой makefile состоит из**(“rules”)** «правил» следующего вида:

цель...: предпосылки... target ... : prerequisites ...

рецепт recipe

... ...

... ...

Целью(target) обычно является имя файла, созданного программой; примеры целейисполняемые или объектные файлы. Цель также может быть названием действия, которое нужно выполнить, например как «чистый»( ‘clean’) (см. Раздел 4.5 [Фальшивые цели] [Phony Targets], стр.29).Необходимым условием является файл, который используется в качестве входных данных для создания цели. Цель часто зависит на несколько файлов.Рецепт — это действие, которое make выполняет. Рецепт может иметь более одной команды,либо в одной строке, либо каждый в своей строке. Обратите внимание: нужно поставить символ табуляциив начале каждой строки рецепта! Это неясность, которая ловит неосторожных. если ты предпочитаете префикс ваших рецептов с символом, отличным от табуляции, вы можете установить .**RECIPEPREFIX** переменной на альтернативный символ (см. Раздел 6.14 [Специальные переменные], с. 73).Обычно рецепт находится в правиле с пререквизитами и служит для создания целевого файла, если какой-либо изпредпосылки меняются. Однако правило, определяющее рецепт для цели, не обязательно иметь предпосылки. Например, правило, содержащее команду удаления, связанную с цель «чистая» не имеет предпосылок.Таким образом, правило объясняет, как и когда переделывать определенные файлы, являющиеся целью конкретное правило. make выполняет рецепт предварительных условий для создания или обновления цель. Правило также может объяснить, как и когда выполнять действие. См. главу 4.[Правила написания], стр. 21.Makefile может содержать другой текст помимо правил, но простой makefile должен содержать толькоправила. Правила могут выглядеть несколько сложнее, чем показано в этом шаблоне, но все подходят рисунок более-менее.Вам нужен файл makefile, чтобы указать make, что делать. Чаще всего makefile говорит make как скомпилировать и скомпоновать программу.В этой главе мы обсудим простой make-файл, описывающий, как компилировать и компоновать текстовый редактор, состоящий из восьми исходных файлов **C** и трех файлов заголовков. Makefile может также сообщите make, как запускать различные команды, когда их явно запрашивают (например, для удалить определенные файлы в качестве операции очистки). Чтобы увидеть более сложный пример make-файла,см. Приложение **C** [Сложный Makefile], стр. 181.Когда make перекомпилирует редактор, каждый измененный исходный файл C должен быть перекомпилирован. Если файл заголовка изменился, каждый исходный файл **C**, который включает файл заголовка, должен быть перекомпилирован в быть безопасным. Каждая компиляция создает объектный файл, соответствующий исходному файлу. Окончательно,если какой-либо исходный файл был перекомпилирован, все объектные файлы, как вновь созданные, так и сохраненные из предыдущих компиляций, должны быть связаны вместе для создания нового исполняемого редактора.

2.2 Простой Makefile

Вот простой **makefile**,описывающий, как исполняемый файл с именем edit зависит от восьми объектных файлов, которые, в свою очередь, зависят от восьми исходных файлов C и трех заголовочных файлов.файлы.В этом примере все файлы **C** включают **defs.h**, но только те, которые определяют команды редактирования.**include command.h**, и только файлы низкого уровня, которые изменяют буфер редактора, включают **buffer.h**.

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

Мы разделяем каждую длинную строку на две строки, используя обратную косую черту(backslash) **/** новую строку; это похоже на использование одного длинного строки, но легче читается. См. Раздел 3.1.1 [Разделение длинных строк], стр. 12.Чтобы использовать этот makefile для создания исполняемого файла(executable file) с именем **edit**, введите: **make** Чтобы использовать этот makefile для удаления исполняемого файла и всех объектных файлов из каталога,тип(type): очистить(make clean).В примере makefile цели включают исполняемый файл(executable file) ‘**edit**’ и **объект** файлы ‘**main.o**’ и ‘**kbd.o**’. Необходимыми условиями являются такие файлы, как ‘**main.c**’ и ‘**defs.h**’. На самом деле, каждый файл **‘.o ‘** является одновременно целью и обязательным условием. Рецепты включают ‘**cc -c main.c**’ и**’cc -c kbd.c’**.Когда целью является файл, его необходимо перекомпилировать или повторно связать, если какое-либо из его предварительных требованийсдача. Кроме того, любые предварительные условия, которые сами по себе генерируются автоматически, должныобновляться в первую очередь. В этом примере редактирование зависит от каждого из восьми объектных файлов; предмет файл **main.o** зависит от исходного файла **main.c** и заголовочного файла **defs.h**.

Глава 2: Введение в файлы **Makefil**

Рецепт может следовать за каждой строкой, содержащей цель и предпосылки. Эти рецепты (recipe)говорят как обновить целевой файл. Символ табуляции**(tab)** (или любой другой символ, указанный **переменная .RECIPEPREFIX**; см. Раздел 6.14 **[Специальные переменные],** с. 73)начало каждой строки в рецепте, чтобы отличать рецепты от других строк в makefile.(Имейте в виду, что make ничего не знает о том, как работают рецепты. Вы сами должны предоставить рецепты, которые будут правильно обновлять целевой файл. *Все, что делает make*, — это выполняет рецепт, который вы указали, когда целевой файл необходимо обновить.)Цель «очистить»( ‘clean’) — это не файл, а просто имя действия. Поскольку вы обычноне хотите выполнять действия, описанные в этом правиле, **‘clean’**не является обязательным условием любого другого правило. Следовательно, make никогда ничего не делает с ним, если вы не укажете это конкретно. Примечание что это правило не только не является обязательным условием, но и не имеет никаких предпосылок, поэтому единственной целью правила является запуск указанного рецепта.Цели, которые не относятся к файлам но только действия называются фальшивыми целями. См. Раздел 4.5 *[Фальшивые цели] [Phony Targets*], стр. 29, для информацию об этом виде цели. См. Раздел 5.5 [Ошибки в рецептах], стр. 49, чтобы увидеть как заставить make игнорировать ошибки от rm или любой другой команды.

2.3 Как make обрабатывает Makefile

По умолчанию **make** начинается с первой цели (не целей, имена которых начинаются с **‘.’** ). Это называется целью по умолчанию. (**Цели(targets)** — *это задачи*, которые в конечном счете стремятся обновить. Вы можно переопределить это поведение с помощью командной строки (см. Раздел 9.2 [Аргументы для указания целей] [**Arguments to Specify the Goals**], с. 99) или с помощью специальной переменной **.DEFAULT\_GOAL** (см. Раздел 6.14 [*Другие Специальные переменные*], стр. 73). В простом примере из предыдущего раздела целью по умолчанию является обновление исполняемого файла.редактирование программы; поэтому мы ставим это правило первым. Таким образом, когда вы даете команду: **make** ,make читает *makefile в* текущем каталоге и начинает с обработки первого правила. В примере это правило предназначено для повторной ссылки редактирования; но прежде чем make сможет полностью обработать это правило, он должен обрабатывать правила для файлов, от которых зависит редактирование, которые в данном случае являются объектными файлами. Каждый из этих файлов обрабатывается по своему правилу. Эти правила говорят об обновлении каждый файл **‘.o ’** скомпилировав его исходный файл. Перекомпиляцию необходимо выполнить, если исходный файл или любой из файлов заголовков, названных в качестве предварительных требований, более поздний, чем объектный файл, или если объектный файл не существует. Другие правила обрабатываются, поскольку их цели отображаются в качестве предварительных условий цель.Если какое-либо другое правило не зависит от цели (или чего-либо еще, от чего оно зависит и т. д.), то это правило не обрабатывается, пока вы не скажете make сделать это (с помощью такой команды, как **make clean**).Прежде чем перекомпилировать объектный файл, **make** может обновить его пререквизиты, исходный файл и файлы заголовков. Этот makefile ничего не указывает для них — файлы **‘.c’** и **‘.h’** не являются объектами каких-либо правил — поэтому **make** ничего не делает для этих файлов. Но **make** будет обновлять автоматически сгенерированные программы **C**, такие как программы, созданные *Bison* или *Yacc*, в соответствии с их собственными правилами в настоящее время. существует, или если какой-либо из объектных файлов новее, чем он. После перекомпиляции необходимых объектных файлов **make** решает, следует ли перекомпоновать**(relink) edit**. Это необходимо сделать, если файл edit не существует или какой-либо из объектных файлов новее, чем он. Если объектный файл был только что перекомпилирован, он теперь новее, чем **edit**, поэтому **edit** повторно связывается. Таким образом, если мы изменим файл **insert.c** и запустим **make,** **make** скомпилирует этот файл, чтобы обновить **insert.o**, а затем связать **edit**. Если мы изменим файл **command.h** и запустим **make**, **make** перекомпилируйте объектные файлы **kbd.o**, **command.o** и **files.o**, а затем свяжите файл **edit.**

**2.4 Переменные делают Makefile проще**

В нашем примере нам пришлось дважды перечислить все объектные файлы в правиле для редактирования (повторяется здесь)

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

Такое дублирование подвержено ошибкам; если в систему добавляется новый объектный файл, мы можем добавить его в один список и забыть о другом. Мы можем устранить риск и упростить **makefile** с помощью переменной. Переменные позволяют определить текстовую строку один раз и заменить ее в несколько раз позже (см. Глава 6 [Как использовать переменные] [How to Use Variables], с. 59). Стандартной практикой для каждого **makefile** является наличие переменной с именем objects, OBJECTS, objs, OBJS, obj, OBJ который представляет собой список всех имен объектных файлов. Мы бы определили такую переменную objectsс такой строкой в **makefile**:

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

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

Затем, в каждом месте, где мы хотим разместить список имен объектных файлов, мы можем заменить значение переменной, написав **‘$(objects)’** (см. Глава 6 [How to Use Variables], стр. 59).Вот как выглядит полный простой makefile, когда вы используете переменную для **объекта**

файлы:

**Глава 2: Введение в файлы Makefile**

**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)**

2.5 Letting make Deduce the Recipes 2.5 Вывод рецептов

Нет необходимости подробно излагать рецепты компиляции отдельных исходных файлов **C**,

потому что **make** может их вычислить: у него есть неявное правило для обновления файла **‘.o’** из

файл с соответствующим именем **‘.c’** с помощью команды **‘cc -c’**.

Например, он будет использовать **рецепт** **‘cc -c main.c -o main.o’** для компиляции **main.c** в **main.o**. Поэтому мы можем опустить рецепты из правил для объектных файлов. См. Глава 10 [Использование неявных правил], стр. 111.Когда файл **‘.c’** используется таким образом автоматически, он также автоматически добавляется всписок предпосылок. Поэтому мы можем опустить файлы **‘.c’** из предварительных условий, если мы опускаем рецепт.

Вот весь пример с обоими этими изменениями и переменными объектами как

предложено выше:

**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)**

Вот как бы мы написали makefile на практике. (осложнения, связанные

с **‘clean’** описаны в другом месте. См. Раздел 4.5 [Фальшивые цели], стр. 29, и

Раздел 5.5 [Ошибки в рецептах], стр. 49.)

Поскольку неявные правила настолько удобны, они важны. Вы увидите, как они используются

Часто

**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)**

Вот как бы мы написали **makefile** на практике. (Осложнения, связанные с **‘clean’**, описаны в другом месте. См. Раздел 4.5 [**Phony Targets**], с. 29, и Раздел 5.5 [**Errors in Recipes**], стр. 49.)

Поскольку неявные правила настолько удобны, они важны. Вы увидите, как они используются

часто.

2**.6 Another Style of Makefile 2.6 Другой стиль Makefile**

Когда объекты makefile создаются только по неявным правилам, альтернативный стиль

makefile возможен. В этом стиле makefile вы группируете записи по их предпосылкам вместо этого.по своим целям. Вот как он выглядит:

**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 **defs.h** указывается как необходимое условие для всех объектных файлов; **command.h** и **buffer.h** являются предварительными требованиями для определенных объектных файлов, перечисленных для них.Лучше ли это — дело вкуса: компактнее, но кому-то не нравится

это потому, что они считают более понятным разместить всю информацию о каждой цели в одном месте.

Глава 2: Введение в файлы **Makefile**

2.7 Правила очистки каталога

Компиляция программы — не единственное, для чего вам может понадобиться написать правила. **Makefiles**

обычно рассказывают, как сделать несколько других вещей помимо компиляции программы: например, как чтобы удалить все объектные файлы и исполняемые файлы, чтобы каталог был **‘clean’** Вот как мы могли бы написать правило **make** для очистки нашего редактора примеров:

**clean:**

**rm edit $(objects)**

На практике мы можем написать правило несколько более сложным образом.

для обработки непредвиденных ситуаций. Мы бы сделали это:

**.PHONY : clean**

**clean :**

**-rm edit $(objects)**

Это предотвратит путаницу с реальным файлом с именем **clean** и заставит его

продолжать, несмотря на ошибки от **rm**. (См. Раздел 4.5 [Фальшивые цели], стр. 29, и Раздел 5.5.

[Ошибки в рецептах], стр. 49.)

Подобное правило не следует размещать в начале make-файла, потому что мы

не хочу, чтобы он работал по умолчанию! Таким образом, в make-файле примера нам нужно правило редактирования,

который перекомпилирует редактор, чтобы он оставался целью по умолчанию.

Поскольку очистка не является обязательным условием для редактирования, это правило вообще не будет выполняться, если мы дадим

команда **make** без аргументов. Чтобы правило заработало, мы должны набрать ‘make

clean’. См. Глава 9 [Как запустить make], стр. 99.

Chapter 3: Writing Makefiles Глава 3: Написание Makefiles

**3 Написание makefile**

Информация, которая говорит make, как перекомпилировать систему, получена из базы данных.

называется makefile

**3.1 Что содержат файлы Makefile**

Файлы Makefile содержат пять типов вещей: **явные правила(explicit rules),**

**неявные правила(implicit rules),**

**определения переменных(variable definitions),**

**директивы(directives)** и **комментарии(comments).** Правила, переменные и директивы подробно описаны в последующих разделах главах.

**Явное правило говорит(explicit rules)**, когда и как переделывать один или несколько файлов, называемых целями правила.

В нем перечислены другие файлы, от которых зависят цели, называемые предпосылками цели,а также может дать рецепт для создания или обновления целей. См. главу 4.

[Правила написания], стр. 21.

**Неявное правило(implicit rules)** говорит, когда и как переделывать класс файлов на основе их имен.

Он описывает, как цель может зависеть от файла с именем, похожим на цель, и

дает рецепт для создания или обновления такой цели. См. главу 10 [Использование неявных правил] стр. 111.

**Определение переменной** — это строка, задающая значение текстовой строки для переменной,могут быть заменены в тексте позже. Простой пример makefile показывает переменную определение объектов в виде списка всех **объектных** файлов (см. раздел 2.4 [Переменные делают Variables Make Makefiles Simpler], стр. 6).

**Директива** — это указание make делать что-то особенное при чтении makefile.

Это включает:

Чтение другого **makefile** (см. Раздел 3.3 [Включение других Makefiles] Including Other Makefiles], с. 13).Принятие решения (на основе значений переменных) использовать или игнорировать часть **makefile** (см. Глава 7 [Условные части файлов Makefile] Conditional Parts of Makefiles, с. 77).Определение переменной из дословной строки, содержащей несколько строк (см. Раздел 6.8).[Определение многострочных переменных], стр. 69).

**«#»** в строке **makefile** начинает комментарий. Это и остальная часть строки игнорируются,

за исключением того, что завершающая обратная косая черта(backslash), не экранированная другой обратной косой чертой(backslash), будет продолжать

комментарий в несколько строк. Строка, содержащая только комментарий (возможно, с пробелами перед ним) фактически пуст и игнорируется. Если вам нужен литерал **#** ,

экранируйте его обратной косой чертой (например, **\#**). Комментарии могут появляться в любой строке makefile, хотя они в определенных ситуациях обращаются специально. Вы не можете использовать комментарии внутри ссылок на переменные или вызовов функций: любой экземпляр # будет рассматриваться буквально (а не как начало комментария) внутри ссылки на переменную или вызова функции. Комментарии внутри рецепта передаются оболочке, как и любой другой текст рецепта. Оболочка решает, как его интерпретировать: является ли это комментарием, зависит от оболочки. В директиве определения комментарии не игнорируются во время выполнения. определение переменной, а скорее сохраняется в значении переменной. Когда переменная развернутые, они будут рассматриваться либо как комментарии, либо как текст рецепта, в зависимости от в контексте, в котором оценивается переменная.