Создание библиотек

Содержание

Глава 1. Библиотеки 2

Глава 2. AR 10

Глава 3. RANLIB 12

Глава 4. MAKEFILE 13

1. Обзор make 16

2. Введение в make-файлы 16

3. Написание make-файлов 23

4. Написание правил 29

5. Написание команд в правилах 48

6. Как использовать переменные 61

7. Условные части make-файла 74

8. Функции преобразования текста 80

9. Как запускать make 92

10. Использование неявных правил 103

11. Использование make для обновления архивных файлов 124

12. Особенности GNU-версии программы make 127

13. Несовместимость и недостающие возможности 131

14. Соглашения о make-файлах 133

15. Приложение. Комплексный пример Make-файла. 142

Глава 1. Библиотеки

Библиотеки позволяют использовать разработанный ранее программный код в различных программах. Таким образом, программист может не разрабатывать часть кода для своей программы, а воспользоваться тем, что входит в состав библиотек.

В языке программирования C код библиотек представляет собой функции, размещенные в файлах, которые скомпилированы в объектные файлы, а те, в свою очередь, объединены в библиотеки. В одной библиотеке объединяются функции, решающие определенный тип задач. Например, существует библиотека математических функций.

У каждой библиотеки должен быть свой заголовочный файл, в котором должны быть описаны прототипы (объявления) всех функций, содержащихся в этой библиотеке. С помощью заголовочных файлов вы "сообщаете" вашему программному коду, какие библиотечные функции есть и как их использовать.

При компиляции программы библиотеки подключаются линковщиком, который вызывается gcc. Если программе требуются только стандартные библиотеки, то дополнительных параметров линковщику передавать не надо (есть исключения). Он "знает", где стандартные библиотеки находятся, и подключит их автоматически. Во всех остальных случаях при компиляции программы требуется указать имя библиотеки и ее местоположение.

Библиотеки бывают двух видов — статические и динамические. Код первых при компиляции полностью входит в состав исполняемого файла, что делает программу легко переносимой. Код динамических библиотек не входит в исполняемый файл, последний содержит лишь ссылку на библиотеку. Если динамическая библиотека будет удалена или перемещена в другое место, то программа работать не будет. С другой стороны, использование динамических библиотек позволяет сократить размер исполняемого файла. Также если в памяти находится две программы, использующие одну и туже динамическую библиотеку, то последняя будет загружена в память лишь единожды.

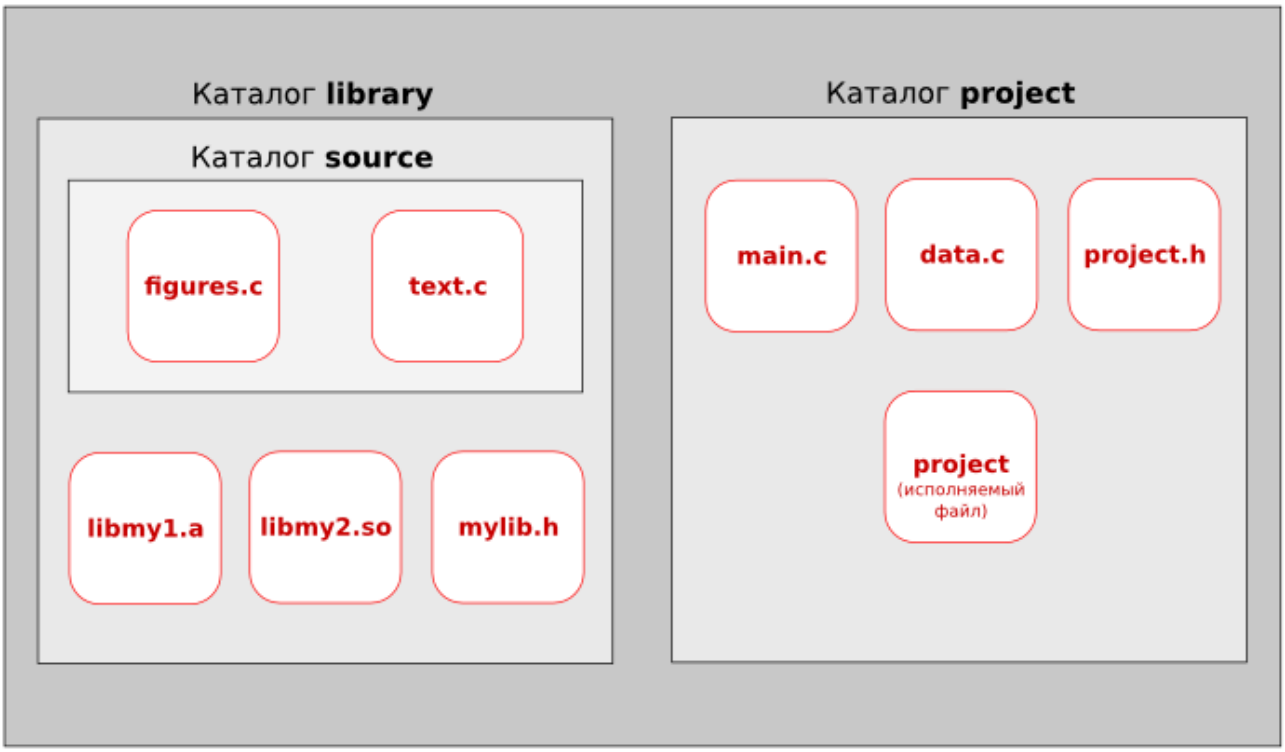
Далее будет описан пример, в котором создается библиотека, после чего используется при создании программы.

Пример создания библиотеки

Допустим, мы хотим создать код, который в дальнейшем планируем использовать в нескольких проектах. Следовательно, нам требуется создать библиотеку. Исходный код для библиотеки было решено разместить в двух файлах исходного кода.

Также на данный момент у нас есть план первого проекта, использующего эту библиотеку. Сам проект также будет включать два файла.

В итоге, когда все будет сделано, схема каталогов и файлов будет выглядеть так:



Пусть каталоги library и project находятся в одном общем каталоге, например, домашнем каталоге пользователя. Каталог library содержит каталог source с файлами исходных кодов библиотеки. Также в library будут находиться заголовочный файл (содержащий описания функций библиотеки), статическая (libmy1.a) и динамическая (libmy2.so) библиотеки. Каталог project будет содержать файлы исходных кодов проекта и заголовочный файл с описанием функций проекта. Также после компиляции с подключением библиотеки здесь будет располагаться исполняемый файл проекта.

В операционных системах GNU/Linux имена файлов библиотек должны иметь префикс "lib", статические библиотеки - расширение *.a, динамические - *.so.

Для компиляции проекта достаточно иметь только одну библиотеку: статическую или динамическую. В образовательных целях мы получим обе и сначала скомпилируем проект со статической библиотекой, потом — с динамической. Статическая и динамическая "разновидности" одной библиотеки по-идее должны называться одинаково (различаются только расширения). Поскольку у нас обе библиотеки будут находиться в одном каталоге, то чтобы быть уверенными, что при компиляции проекта мы используем ту, которую хотим, их названия различны (libmy1 и libmy2).

Исходный код библиотеки

Файл figure.c:

```
void rect(char sign, int w, int h) {
    int i, j;

    for (i=0; i < w; i++)
        putchar(sign);
    putchar('\n');

    for (i=0; i < h-2; i++) {
        for (j=0; j < w; j++) {
            if (j == 0 || j == w-1)
                putchar(sign);
            else
                putchar(' ');
        }
        putchar('\n');
    }

    for (i=0; i < w; i++)
        putchar(sign);
    putchar('\n');
}

void diagonals(char sign, int w) {
    int i, j;

    for (i=0; i < w; i++) {
        for (j=0; j < w; j++) {
            if (i == j || i+j == w-1)
                putchar(sign);
            else
                putchar(' ');
        }
        putchar('\n');
    }
}
```

В файле figure.c содержатся две функции — `rect()` и `diagonals()`. Первая принимает в качестве аргументов символ и два числа и "рисует" на экране с помощью указанного символа прямоугольник заданной ширины и высоты. Вторая функция выводит на экране две диагонали квадрата ("рисует" крестик).

Файл text.c:

```
void text(char *ch) {
    while (*ch++ != '\0')
        putchar('*');
```

```
    putchar('\n');
}
```

В файле text.c определена единственная функция, принимающая указатель на символ строки. Функция выводит на экране звездочки в количестве, соответствующем длине указанной строки.

Файл mylib.h:

```
void rect(char sign, int width, int height);
void diagonals(char sign, int width);
void text(char *ch);
```

Заголовочный файл можно создать в каталоге source, но мы лучше сохраним его там, где будут библиотеки. В данном случае это на уровень выше (каталог library). Тем самым как бы подчеркивается, что файлы исходных кодов после создания из них библиотеки вообще не нужны пользователям библиотек, они нужны лишь разработчику библиотеки. А вот заголовочный файл библиотеки требуется для ее правильного использования.

Создание статической библиотеки

Статическую библиотеку создать проще, поэтому начнем с нее. Она создается из обычных объектных файлов путем их архивации с помощью утилиты ar.

Все действия, которые описаны ниже выполняются в каталоге library (т.е. туда надо перейти командой cd). Просмотр содержимого каталога выполняется с помощью команды ls или ls -l.

Получаем объектные файлы:

```
gcc -c ./source/*.c
```

В итоге в каталоге library должно наблюдаться следующее:

```
figures.o  mylib.h  source  text.o
```

Далее используем утилиту ar для создания статической библиотеки:

```
ar r libmy1.a *.o
```

Параметр r позволяет вставить файлы в архив, если архива нет, то он создается. Далее указывается имя архива, после чего перечисляются файлы, из которых архив создается.

Объектные файлы нам не нужны, поэтому их можно удалить:

```
rm *.o
```

В итоге содержимое каталога library должно выглядеть так:

```
libmy1.a  mylib.h  source
```

, где libmy1.a — это статическая библиотека.

Создание динамической библиотеки

Объектные файлы для динамической библиотеки компилируются особым образом. Они должны содержать так называемый позиционно-независимый код (position independent code). Наличие такого кода позволяет библиотеке подключаться к программе, когда последняя загружается в память. Это связано с тем, что библиотека и программа не являются единой программой, а значит как угодно могут располагаться в памяти относительно друг друга. Компиляция объектных файлов для динамической библиотеки должна выполняться с опцией -fPIC компилятора gcc:

```
gcc -c -fPIC source/*.c
```

В отличие от статической библиотеки динамическую создают при помощи gcc указав опцию -shared:

```
gcc -shared -o libmy2.so *.o
```

Использованные объектные файлы можно удалить:

```
rm *.o
```

В итоге содержимое каталога library:

```
libmy1.a  libmy2.so  mylib.h  source
```

Использование библиотеки в программе

Исходный код программы

Теперь в каталоге project (который у нас находится на одном уровне файловой иерархии с library) создадим файлы проекта, который будет использовать созданную библиотеку. Поскольку сама программа будет состоять не из одного файла, то придется здесь также создать заголовочный файл.

Файл data.c:

```
#include <stdio.h>
#include "../library/mylib.h"

void data (void) {
    char strs[3][30];
    char *prompts[3] = {
        "Ваше имя: ",
        "Местонахождение: ",
        "Пунк прибытия: "};
    int i;

    for (i=0; i<3; i++) {
        printf("%s", prompts[i]);
        gets(strs[i]);
```

```
    }

    diagonals('~', 7);

    for (i=0; i<3; i++) {
        printf("%s", prompts[i]);
        text(strs[i]);
    }
}
```

Функция `data()` запрашивает у пользователя данные, помещая их в массив strs. Далее вызывает библиотечную функцию `diagonals()`, которая выводит на экране "крестик". После этого на каждой итерации цикла вызывается библиотечная функция `text()`, которой передается очередной элемент массива; функция `text()` выводит на экране звездочки в количестве равному длине переданной через указатель строки.

Обратите внимание на то, как подключается заголовочный файл библиотеки: через относительный адрес. Две точки обозначают переход в каталог на уровень выше, т.е. родительский по отношению к project, после чего путь продолжается во вложенный в родительский каталог library. Можно было бы указать абсолютный путь, например, "/home/pl/c/les22/library/mylib.h". Однако при перемещении каталогов библиотеки и программы на другой компьютер или в другой каталог адрес был бы уже не верным. В случае с относительным адресом требуется лишь сохранять расположение каталогов project и library относительно друг друга.

Файл main.c:

```
#include <stdio.h>
#include "../library/mylib.h"
#include "project.h"

int main() {
    rect('-',75,4);
    data();
    rect('+',75,3);
}
```

Здесь два раза вызывается библиотечная функция rect() и один раз функция data() из другого файла проекта. Чтобы сообщить функции main() прототип data() также подключается заголовочный файл проекта.

Файл project.h содержит всего одну строчку:

```
void data(void);
```

Из обоих файлов проекта с исходным кодом надо получить объектные файлы для объединения их потом с файлом библиотеки. Сначала мы получим исполняемый файл, содержащий статическую библиотеку, потом — связанный с динамической библиотекой. Однако с какой бы библиотекой мы

не компоновали объектные файлы проекта, компилируются они как для статической, так и динамической библиотеки одинаково:

```
gcc -c *.c
```

При этом не забудьте сделать каталог project текущим!

Компиляция проекта со статической библиотекой

Теперь в каталоге project есть два объектных файла: main.o и data.o. Их надо скомпилировать в исполняемый файл project, объединив со статической библиотекой libmy1.a. Делается это с помощью такой команды:

```
gcc -o project *.o -L../library -lmy1
```

Начало команды должно быть понятно: опция -o указывает на то, что компилируется исполняемый файл project из объектных файлов.

Помимо объектных файлов проекта в компиляции участвует и библиотека. Об этом свидетельствует вторая часть команды: -L../library -lmy1. Здесь опция -L указывает на адрес каталога, где находится библиотека, он и следует сразу за ней. После опции -l записывается имя библиотеки, при этом префикс lib и суффикс (неважно .a или .so) усекаются. Обратите внимание, что после данных опций пробел не ставится.

Опцию -L можно не указывать, если библиотека располагается в стандартных для данной системы каталогах для библиотек. Например, в GNU/Linux это /lib/, /urs/lib/ и др.

Запустив исполняемый файл project и выполнив программу, мы увидим на экране примерно следующее:

Посмотрим размер файла project:

```
pl@desk:~/c/project$ ls -l project
-rwxr-xr-x 1 pl pl 8648 ноя 19 07:46 project
```

Его размер равен 8698 байт.

Компиляция проекта с динамической библиотекой

Теперь удалим исполняемый файл и получим его уже связанным с динамической библиотекой. Команда компиляции с динамической библиотекой выглядит так (одна команда разбита на две строки с помощью обратного слэша и перехода на новую строку):

```
gcc -o project *.o \
> -L../library -lmy2 -Wl,-rpath,../library/
```

Здесь в отличии от команды компиляции со статической библиотеки добавлены опции для линковщика: -Wl,-rpath,../library/. -Wl - это обращение к линковщику, -rpath - опция линковщика, ../library/ - значение опции. Получается, что в команде мы два раза указываем местоположение библиотеки: один раз с опцией -L, а второй раз с опцией -rpath. Видимо для того, чтобы понять, почему так следует делать, потребуется более основательно изучить процесс компиляции и компоновки программ на языке C.

Следует заметить, что если вы скомпилируете программу, используя приведенную команду, то исполняемый файл будет запускаться из командной строки только в том случае, если текущий каталог project. Стоит сменить каталог, будет возникать ошибка из-за того, что динамическая библиотека не будет найдена. Но если скомпилировать программу так:

```
gcc -o project *.o -L../library -lmy2 \
> -Wl,-rpath,/home/pl/c/library
```

, т.е. указать для линковщика абсолютный адрес, то программа в данной системе будет запускаться из любого каталога.

Размер исполняемого файла проекта, связанного с динамической библиотекой, получился равным 8544 байта. Это немного меньше, чем при компиляции проекта со статической библиотекой. Если посмотреть на размеры библиотек:

```
pl@desk:~/c/library$ ls -l libmy*
-rw-r--r-- 1 pl pl 3712 ноя 19 07:35 libmy1.a
-rwxr-xr-x 1 pl pl 7896 ноя 19 07:36 libmy2.so
```

, то видно, что динамическая больше статической, хотя исполняемый файл проекта со статической библиотекой больше. Это доказывает, что в исполняемом файле, связанном с динамической библиотекой, присутствует лишь ссылка на нее.

Глава 2. AR

Программа `ar` создает, модифицирует и извлекает из архивов.

Архив - это отдельный файл, содержащий коллекцию других файлов в структуре, которая позволяет восстановить первоначальные файлы (называются членами архива).

Использование:

ar [параметры эмуляции] [-]{dmpqrstx}[abcDfilMNoPsSTuvV]

[--plugin <имя>] [имя_члена] [счётчик] файл_архива файл…

ar -M [<mri-скрипт]

Команды:

d - удаление файлов из архива

m[ab] - перемещение файлов в архив

p - вывод файлов, найденных в архиве

q[f] - быстрое добавление файлов в архив

r[ab][f][u] - замена существующих или вставка новых файлов в архив

s - действовать как ranlib

t - отображение содержимого архива

x[o] - извлечение файлов из архива

Особые модификаторы для команд:

[a] - размещение файлов после [имени_члена]

[b] - размещение файлов до [имени_члена] (то же, что и [i])

[D] - use zero for timestamps and uids/gids (default)

[U] - use actual timestamps and uids/gids

[N] - использование [счета], как варианта имени

[f] - обрезание имен вставленных файлов

[P] - использование полных путевых имен при сопоставлении

[o] - сохранение исходных дат

[u] - замена только файлов более новых, чем текущее содержимое архива

Обычные модификаторы:

[c] - не предупреждать, если должна быть создана библиотека

[s] - создание индекса архива (cf. ranlib)

[S] - не создавать таблицу символов

[T] - создание полупустого архива

[v] - подробный режим

[V] - вывод номера версии

@<файл> - читать параметры из <файла>

--target=BFDNAME - назначить форматом объекта назначения BFDNAME

необязательные:

--plugin <p> - загрузить указанный модуль

Параметры эмуляции:

Нет параметров эмуляции

Поддерживаемые цели:

elf64-x86-64 elf32-i386 elf32-iamcu elf32-x86-64 a.out-i386-linux
pei-i386 pei-x86-64 elf64-l1om elf64-k1om elf64-little elf64-big
elf32-little elf32-big pe-x86-64 pe-bigobj-x86-64 pe-i386 plugin
srec symbolsrec verilog tekhex binary ibex

Глава 3. RANLIB

Программа `ranlib` генерирует индекс содержимого архива для увеличения скорости доступа к нему и сохраняет его в архиве.

В индексе перечислены все символы, определенные участником архива, который является перемещаемым файлом объекта.

Вы можете использовать `nm -s` или `nm --print-armap` для отображения этого индекса.

Архив с таким индексом ускоряет соединение с библиотекой и позволяет подпрограммам в библиотеке вызывать друг друга независимо от их размещения в архиве.

Программа `ranlib` - это еще одна форма `ar`; запуск `ranlib` полностью эквивалентен выполнению `ar -s`.

Использование:

ranlib [параметры] архив

Параметры:

@<файл> читать параметры из <файла>

--plugin <название> загрузить указанный модуль

-D Use zero for symbol map timestamp (default)

-U Use an actual symbol map timestamp

-t обновить временные отметки карты символов архива

-h --help показать это справочное сообщение

-V --version показать информацию о версии

поддерживаемые цели:

elf64-x86-64 elf32-i386 elf32-iamcu elf32-x86-64 a.out-i386-linux

elf32-little elf32-big pei-i386 pei-x86-64 elf64-l1om elf64-k1om

elf64-little elf64-big pe-x86-64 pe-bigobj-x86-64 pe-i386 plugin

srec symbolsrec verilog tekhex binary ihex

Глава 4. MAKEFILE

Синтаксис make-файла очень простой:
цель: зависимости команда

Например, рассмотрим следующее объявление:

```
hello.o: hello.c hello.h
    gcc -c $< -o $@
```

В этом случае вы можете увидеть использование специальных макросов - автоматических переменных '$@' и '$<'.

hello.o оценивается как $@ - это имя файла цели.
hello.c оценивается как $< - это имя первой зависимости
hello.c hello.h оценивается как $^ - это имена всех зависимостей

т.е переменная '$@' представляет собой желаемую цель,
а переменная '$<' представляет собой предварительное условие выполнения цели, которое необходимо для создания нашей цели - выходного файла(hello.o).

Запустите эту команду, она выведет внутреннюю базу данных команды 'make':

```
make -p
```

NAME := libft.a	Имя нужной нам библиотеки
CC := gcc	Название компилятора
CFLAGS := -Wall -Werror -Wextra	Флаги для компиляции
SRC_DIR := ./srcs/	Папка с исходниками
SRC_FILES := ft_putchar.c ft_putstr.c ft_strcmp.c ft_strlen.c ft_swap.c	Исходные файлы
OBJ_FILES = $(SRC_FILES:.c=.o)	обьектные файлы
SRCS = $(addprefix $(SRC_DIR), $(SRC_FILES))	Полные пути к файлам
OBJS = $(addprefix $(SRC_DIR), $(OBJ_FILES))	
OBJS = $(patsubst %.c, %.o, $(SRCS))	OBJS можно запись вот так
OBJS = $(SRCS:.c=.o)	
HEADER := -I includes/	Флаг и папка с заголовочными файлами
LIBC := ar rc	Программы создания файла libft.a
LIBR := ranlib	Программы создания библиотеки libft.a
RM := rm -f	Программа для зачистки мусора
all: obj $(NAME)	Все правила

obj: $(SRCS) $(SRC_DIR)%.o: $(SRC_DIR)%.c $(CC) $(CFLAGS) -c $< -o $@ $(HEADER)	Компиляция обьектных файлов
FLAGS = -Wall -Wextra -Werror obj: $(SRCS) $(OBJ_DIR)%.o:$(SRC_DIR)%.c $(CC) $(FLAGS) $(HEADER) -o $@ -c $<	Другие варианты компиляции обьектных файлов
.c.o: $(CC) $(CFLAGS) -c $< -o $(<:.c=.o) $(HEADER)	
FLAGS = -Wall -Wextra -Werror %.o: %.c $(CC) -I $(INC) $< $(FLAGS) -o $@ -c %.c : $(SRC)	
FLAGS = -Wall -Wextra -Werror %.o:%.c @gcc $(FLAGS) -o $@ -c $<	
$(NAME): $(OBJS) $(LIBC) $(NAME) $(OBJS) $(LIBR) $(NAME)	Сборка обьектных файлов в один архив-библиотеку
clean: /bin/rm -f $(OBJS)	Правило очистки от обьектных файлов
fclean: clean /bin/rm -f $(NAME) program.out	Правило очистки от обьектных файлов и файла библиотеки libft.a
re: fclean all	Правило зачистки от мусора и перезапуска всех правил заново
.PHONY: all clean fclean re	Правило которое говорит сборщику чтобы он не воспринимал правила из списка 'phony' за файлы

Цвета:

NOC = \033[0m #non color
BOLD = \033[1m
UNDERLINE = \033[4m
BLACK = \033[1;30m
RED = \033[1;31m
GREEN = \033[1;32m
YELLOW = \033[1;33m
BLUE = \033[1;34m
VIOLET = \033[1;35m
CYAN = \033[1;36m
WHITE = \033[1;37m
	Цвета

obj: $(SRCS) @echo "==========…" @echo "$(YELLOW)[0%] (/◕ヮ◕)/* ・゜✧ \tCompiling... Wait a sec.$(NOC)" $(SRC_DIR)%.o: $(SRC_DIR)%.c $(CC) $(CFLAGS) -c $< -o $@ $(HEADER)	Компиляция обьектных файлов
$(NAME): $(OBJS) $(LIBC) $(NAME) $(OBJS) $(LIBR) $(NAME) @echo "$(YELLOW)[100%] (•̀ᴗ•́)و Compilation of '$(NAME)' is done$(NOC)" @echo "==========…" @echo "$(CYAN)>>> CHECK LIBRARY. \033[0m" @echo "$(CYAN)Make sure the '$(NAME)' library works $(NOC)" @echo "$(CYAN)compile the test program using the generated libft.a library $(NOC)" $(CC) $(CFLAGS) -o program.out test.c $(NAME) @echo "==========…" @echo "$(CYAN)>>> CHECK PROGRAM './test' $(NOC)" ./program.out @echo "==========…"	Сборка обьектных файлов в один архив-библиотеку
clean: @echo "==========…" @echo "$(RED)¯\_(ツ)_/¯ Objects remove!$(NOC)" /bin/rm -f $(OBJS) @echo "==========…"	Правило очистки от обьектных файлов
fclean: clean @echo "==========…" @echo "$(RED)(╯°□°) ╯ ︵ ┻━┻ $(NAME) remove!$(NOC)" /bin/rm -f $(NAME) program.out @echo "==========…"	Правило очистки от обьектных файлов и файла библиотеки libft.a
norm : norminette -R CheckForbiddenSourceHeader */*.[ch]	Проверка исходников на соблюдение норм
unzip: ar -x libft.a	Распаковка обьектных файлов из libft.a

https://dimaru.github.io/make-doc/make.html
https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html
https://linoxide.com/learn-linux-makefiles/
http://rus-linux.net/nlib.php?name=/MyLDP/algol/gnu_make/gnu_make_3-79_russian_manual.html#SEC101

1. Обзор make

Утилита make автоматически определяет, какие части большой программы должны быть перекомпилированы и команды для их перекомпиляции, Это руководство описывает GNU make, который был реализован Ричардом Столлманом и Роландом МакГратом, GNU make удовлетворяет разделу 6.2 стандарта IEEE 1003.2-1992 (POSIX.2).

В наших примерах демонстрируются C-программы, поскольку они встречаются наиболее часто, но вы можете использовать make с любым языком программирования, компилятор которого может запускаться из командной строки. На самом деле, применение утилиты make не ограничивается программами. Вы можете использовать его для описания любой задачи, где некоторые файлы должны автоматически порождаться из других всегда, когда те изменяются.

Прежде чем использовать make, вы должны создать файл, называемый make-файлом, который описывает отношения между файлами вашей программы и содержит команды для обновления каждого файла. Обычно в программе исполняемый файл обновляется на основе объектных файлов, которые, в свою очередь, создаются путем компиляции исходных файлов.

Как только существует подходящий make-файл, для выполнения всех необходимых перекомпиляций при изменении исходных файлов достаточно просто набрать в командной строке:

```
make
```

Программа make использует информацию из make-файла и время последнего изменения каждого файла для того, чтобы решить, какие файлы нужно обновить. Для каждого обновляемого файла он вызывает соответствующие команды, указанные в make-файле.

Вы можете при вызове make использовать аргументы командной строки для управления тем, какие файлы следует перекомпилировать и как это делать. Смотрите главу 9 [Как запускать make].

2. Введение в make-файлы

Вам нужен файл, называемый make-файлом, чтобы указать программе make, что делать. Чаще всего, make-файл указывает, как компилировать и компоновать программу.

В этой главе мы обсудим простой make-файл, который описывает, как компилировать и компоновать тектовый редактор, состоящий из восьми исходных C-файлов и трех заголовочных файлов. Этот make-файл может также указывать программе make, как выполнить различные команды, если явно указан запрос на их исполнение (например, удалить определенные файлы в качестве команды clean). Более сложный пример make-файла можно увидеть в приложении Б [Сложный make-файл].

Когда make перекомпилирует редактор, каждый измененный исходный C-файл должен быть перекомпилирован. Если был изменен заголовочный файл, на всякий случай нужно перекомпилировать каждый исходный C-файл, который его включает. Каждая компиляция порождает объектный файл, соответствующий исходному файлу. Наконец, если какой-либо исходный файл был

перекомпилирован, все объектные файлы, как новые, так и оставшиеся от предыдущих компиляций, должны быть скомпонованы вместе для создания нового исполняемого файла редактора.

2.1 На что похоже правило

Простой make-файл состоит из "правил" следующего вида:

```
ЦЕЛЬ ...        : ЗАВИСИМОСТЬ ...
     КОМАНДА
     ...
     ...
```

>

ЦЕЛЬ обычно представляет собой имя файла, генерируемого программой make; примерами целей являются исполняемые или объектные файлы. Цель может также быть именем выполняемого действия, как, например, 'clean' (смотрите раздел 4.4 [Цели-имена действий]).

ЗАВИСИМОСТЬ - это файл, используемый как вход для порождения цели. Часто цель зависит от нескольких файлов.

КОМАНДА - это действие, которое выполняет make. Правило может иметь более, чем одну команду - каждую на своей собственной строке. Важное замечание: вы должны начинать каждую строку, содержащую команды, с символа табуляции. Это является незаметным средством борьбы с неострожностью.

Обычно команда появляется в правиле с зависимостями и служит для создания целевого файла, если какая-либо из зависимостей изменилась. Однако, правило, определяющее команды для цели, не обязательно должно иметь зависимости. Например, правило, содержащее команду удаления, связанную с целью 'clean', не имеет зависимостей.

Правила описывают, как и когда заново порождать определенные файлы, которые являются целями правил. Правило может также описывать, как и когда выполнять действие. Смотрите раздел 4 [Написание правил].

Помимо правил, make-файл, может содержать другой текст, однако простой make-файл содержит только правила. Правила могут выглядеть более сложными, чем показаный шаблон, но все они более или менее соответствуют ему по структуре.

2.2 Простой make-файл

Вот простой make-файл, который описывает, как исполняемый файл, называемый edit, зависит от восьми объектных файлов, которые, в свою очередь, зависят от восьми исходных C-файлов и трех заголовочных файлов.

В этом примере все C-файлы включают 'defs.h', но файл 'command.h' включают только те, которые определяют команды редактирования, а файл 'buffer.h' - только файлы низкого уровня, изменяющие буфер 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
```

Мы разбиваем каждую длинную строку на две строки, используя обратную косую черту, за которой следует перевод строки; это аналогично использованию одной длинной строки, но легче для чтения.

Чтобы использовать этот make-файл для создания исполняемого файла с именем 'edit', наберите в командной строке:

```
make
```

Чтобы использовать этот make-файл для удаления исполняемого файла и всех объектных файлов из текущего каталога, наберите в командной строке:

```
make clean
```

В приведенном примере make-файла к целям относятся, в частности, исполняемый файл 'edit' и объектные файлы 'main.o' и 'kbd.o'. К зависимостям относятся такие файлы, как 'main.c' и 'defs.h'. Фактически, каждый объектный файл является как целью, так и зависимостью. Примерами команд являются 'cc -c main.c' и 'cc -c kbd.c'.

Когда цель является файлом, этот файл должен быть перекомпилирован или перекомпонован, если изменилась любая из его зависимостей. Кроме того, любые зависимости, которые сами автоматически генерируются, должны обновляться первыми. В этом примере, 'edit' зависит от каждого из девяти объектных файлов;

объектный файл 'main.o' зависит от исходного файла 'main.c' и заголовочного файла 'defs.h'.

За каждой строкой, содержащей цель и зависимости, следует команда. Эти команды указывают, как обновлять целевой файл. В начале каждой командой строки должен располагаться символ табуляции, чтобы отличать командные строки от других строк make-файла. (Держите в уме, что make ничего не знает о том, как работают команды. Обеспечить команды, которые корректно обновят целевой файл - целиком ваша забота. Все, что делает make - это выполнение команд из опредeнного вами правила, когда целевой файл должен быть обновлен.)

Цель 'clean' является не файлом, а просто именем действия. Так как обычно вы не хотите выполнять действия из этого правила, 'clean' не является зависимостью какого-либо другого правила. Следовательно, make никогда ничего с ним не сделает, если вы этого специально не укажете. Обратите внимание, что это не только не является зависимостью, оно также не имеет никаких зависимостей, таким образом, единственным предназначением правила является выполнение определенных в правиле команд. Цели, которые не указывают на файлы, а являются просто действиями, называются целями-именами действий. Смотрите раздел 4.4 [Цели-имена действий] для информации об этой разновидности целей. Смотрите раздел 5.4 [Ошибки в командах], где показывается, как заставить make игнорировать ошибки от rm и любых других команд.

2.3 Как make обрабатывает make-файл

По умолчанию, make начинает с первого правила (не считая правил, имена целей у которых начинаются с '.'). Это называется главной целью по умолчанию. (Главной целью называется цель, обновление которой является изначальной задачей программы make. Смотрите раздел 9.2 [Аргументы для определения главных целей].)

В простом примере из предыдущего раздела главной целью по умолчанию является обновление исполняемой программы 'edit'; следовательно, мы располагаем это правило первым.

Таким образом, когда вы даете команду:

```
make
```

make читает make-файл в текущем каталоге и начинает с обработки первого правила. В приведенном примере им является правило для перекомпоновки 'edit'; но, прежде чем make сможет полностью обработать это правило, он должен обработать правила для файлов, от которых зависит 'edit' (в данном случае ими являются объектные файлы). Каждый из этих файлов обрабатывается в соответствии со своими собственным правилом. Эти правила указывают обновить каждый объектный файл путем компиляции его исходного файла. Перекомпиляция должна быть проведена, если исходный файл или любой из заголовочных файлов, упомянутых среди зависимостей, обновлен позднее, чем объектный файл, или если объектный файл не существует.

Другие правила обрабатываются по той причине, что их цели появляются в качестве зависимостей главной цели. Если от какого-либо правила не зависит главная цель (или что-нибудь, отчего она зависит), то это правило не

обрабатывается, если вы не укажете программе make сделать это (с помощью такой команды, как make clean).

Перед перекомпиляцией объектного файла make рассматривает время обновления его зависимостей: исходного файла и заголовочных файлов. Данный make-файл не определяет ничего, что должно делаться для их порождения - файлы, имена которых оканчиваются на '.c' и '.h' не являются целями каких-либо правил - таким образом, make ничего не делает для этих файлов. Однако make мог бы обновить автоматически генерируемые C-программы, например, получаемые с помощью программ Bison или Yacc, если бы для них, в этом случае, были определены свои правила.

После перекомпиляции всех объектных файлов, для которых это необходимо, make решает, перекомпоновывать ли 'edit'. Это должно быть сделано, если файл 'edit' не существует, или какой-либо из объектных файлов обновлен позднее его. Если объектный файл был только что перекомпилирован, то сейчас он новее, чем 'edit', так что 'edit' перекомпоновывается.

Таким образом, если мы изменим файл 'insert.c' и запустим make, make откомпилирует этот файл для обновления 'insert.o', и затем скомпонует 'edit'. Если мы изменим файл 'command.h' и запустим make, make перекомпилирует объектные файлы 'kbd.o', 'command.o', и 'files.o', а затем скомпонует 'edit'.

2.4 Переменные упрощают make-файл

В нашем примере мы вынуждены были дважды перечислять все объектные файлы в правиле для 'edit':

```
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 способствует появлению ошибок - если в систему добавляется новый объектный файл, мы можем добавить его в один список и забыть про другой. Мы можем устранить риск и упростить make-файл при помощи использования переменных. Переменная позволяет один раз определить текстовую строку и затем подставлять ее во многих местах (смотрите главу 6 [Как использовать переменные]).

Для любого make-файла стандортной практикой является наличие переменной, называемой objects, OBJECTS, objs, OBJS, obj или OBJ, которая представляет собой список имен всех объектных файлов. Мы могли бы определить такую переменную objects со значением, являющимся списком объектных файлов из приведенного make-файла:

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

Тогда везде, где мы хотим поместить список имен объектных файлов, мы можем подставить значение переменной, написав '$(objects)' (смотрите главу 6 [Как использовать переменные]).

Вот как полностью выглядит make-файл, когда вы используете переменную для 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)
```

2.5 Возможность использования неявных команд

Писать команды для компиляции отдельных исходных файлов не является необходимым, поскольку make может сам их определить: он имеет неявное правило для обновления '.o'-файла из соответствующего '.c'-файла, используя команду 'cc -c'. Например, он будет использовать команду 'cc -c main.c -o main.o' для компиляции 'main.c' в 'main.o'. Следовательно, мы можем опустить команды из правил для объектных файлов. Смотрите главу 10 [Использование неявных правил].

Когда '.c'-файл автоматически используется таким способом, он также автоматически добавляется в список зависимостей. Поэтому мы можем опустить '.c'-файлы в зависимостях, позволяющих нам опустить команды для компиляции.

Вот пример, использующий оба этих изменения, а также переменную objects, о которой говорится 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)
```

Именно так нам следует писать make-файл в реальной практике. (Усложнения, связанные с 'clean', опиываются в другом месте. Смотрите раздел 4.4 [Цели-имена действий] и раздел 5.4 [Ошибки в командах].)

Неявные правила важны из-за их удобства. Вы увидите, что они используются часто.

2.6 Еще один стиль make-файла

Когда объекты make-файла создаются только при помощи правил по умолчанию, возможен альтернативный стиль построения make-файла. При его использовании вы группируете записи по их зависимостям, а не по их целям. Вот как это 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
```

В данном случае 'defs.h' является зависимостью для всех объектных файлов; 'command.h' и 'buffer.h' является зависимостью для определенных файлов, перечисленных в соответствующих правилах.

Является ли это лучшим способом - дело вкуса; этот способ более компактен, тем не менее некоторые недолюбливают его, поскольку считают более удобным располагать информацию о каждой цели в одном месте.

2.7 Правила для очистки каталога

Вы могли бы захотеть написать правила не только для компиляции программ. Make-файлы часто указывают, как делать некоторые другие действия, отличные от компиляции: например, как удалить все объектные и исполняемые файлы в текущем каталоге (очистить каталог).

Вот как мы могли бы написать правило make для очистки нашего редактора из примера:

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

На практике, мы могли бы захотеть написать правило несколько более сложным способом, чтобы обработать непредвидельные ситуации. Сделаем следующее:

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

Это не дает программе make нарушить логику работы, когда используемый файл называется 'clean' и заставляет ее продолжаться вопреки ошибкам со стороны rm. (смотрите раздел 4.4 [Цели-имена действий]) 5.4 [Ошибки в командах]).

Правило, подобное этому, не следует размещать в начале make-файла, поскольку мы не хотим выполнять его по умолчанию! Таким образом, в примере make-файла мы хотим, чтобы правило для 'edit', которое перекомпилирует редактор, оставалось главной целью по умолчанию.

Так как 'clean' не является зависимостью 'edit', это правило вообще не будет выполняться, если вы дали команду 'make' без аргументов. Для того, чтобы заставить правило выполниться, мы должны набрать 'make clean'. Смотрите главу 9 [Как запускать make].

3. Написание make-файлов

Информация, указывающая программе make, как перекомпилировать систему, получается из специального make-файла.

3.1 Что содержат make-файлы

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

- Явное правило указывает, указывает, когда и как заново порождать один или более файлов, называемых целями правила. В нем перечисляются файлы, от которых зависят цели и могут также быть даны команды,

используемые для создания или обновления целей. Смотрите главу 4 "Написание правил".

- Неявное правило указывает, когда и как заново порождать класс файлов на основе их имен. Оно описывает, как цель может зависеть от файла с именем, похожим на имя цели и давать команды для создания или обновления таких целей. Смотрите главу 10 "Использование неявных правил".
- Определение переменной - это строка make-файла, которая определяет значение переменной, представляющее собой текстовую строку, которая в дальнейшем может быть подставлена в текст.
- Директива является указанием программе make сделать что-либо при чтении make-файла. Возможны следующие указания:
    - Чтение другого make-файла (смотрите раздел 3.3 "Включение других make-файлов").
    - Принятие решения (на основе значений переменных), использовать или игнорировать часть make-файла (смотрите главу 7 "Условные части make-файла").
    - Определение многостроковой переменной на основе нескольких строк make-файла (смотрите раздел 6.8 "Определение многостроковых переменных").
- Символ '#' в make-файле является началом комментария. Он сам и остаток строки игнорируются, за исключение символа '\' (если ему не предшествует такой же символ, это означает продолжение комментария на следующую строку). Комментарии могут появиться в любом месте make-файла. Исключениями являются определения с использованием директивы define и, возможно, команды в правилах (здесь уже командная оболочка решает, что является комментарием). Строка, состоящая только из комментария (возможно, с пробелами перед ним) рассматривается как пустая и игнорируется.

3.2 Как назвать make-файл

По умолчанию, когда программа make ищет make-файл, она пытается использовать следующие имена: 'GNUmakefile', 'makefile' и 'Makefile' (в указанном порядке).

Обычно вам имеет смысл назвать ваш make-файл либо 'makefile', либо 'Makefile'. (Мы рекомендуем 'Makefile', поскольку в этом случае при выводе содержимого каталога он будет выделяться, появляясь в начале списка, рядом с такими важными файлами, как 'README'.) Первое из указанных имен, 'GNUmakefile', не рекомендуется для большинства make-файлов. Вам следует использовать это имя, если ваш make-файл специфичен для GNU make и не будет воспринят другими версиями программы make. Другие версии программы make ищут файлы с именами 'makefile' и 'Makefile', но 'GNUmakefile'.

Если make не находит файла ни с одним из этих имен, он не использует никакого make-файла. Затем вы должны главную цель как аргумент командной строки, и make попытается выяснить, как заново породить ее, используя только встроенные неявные правила. Смотрите главу 10 [Использование неявных правил].

Если вы хотите использовать нестандартное имя для вашего make-файла, вы можете определить имя make-файла с помощью опций '-f' или '--file'. Аргументы '-f <имя файла>' или '--file=<имя файла>'. указывают программе make читать файл с именем <имя файла> в качестве make-файла. При использовании более, чем одной опции '-f <имя файла>' или '--file=<имя файла>', вы можете определить несколько make-файлов. Все указанные make-файлы просто присоединяются друг за другом в указанном порядке. Имена make-файла по умолчанию ('GNUmakefile', 'makefile' и 'Makefile') не проверяются автоматически, если вы определяете опции '-f' или '--file'.

3.3 Включение других make-файлов

Директива include указывает программе make приостановить чтение текущего make-файла и, прежде чем продолжать, прочитать один или более других make-файлов. Эта директива представляет собой строку make-файла, которая выглядит так:

```
include <имя файла> ...
```

В качестве имени файла может использоваться шаблон имени файла, используемый в командной оболочке.

В начале строки допустимы пробелы, которые игнорируются, однако символы табуляции недопустимы (если строка начинается с табуляции, она будет рассматриваться как командная строка). Между словом 'include' и именами файлов, а также между именами файлов необходим пробел, а лишние пробелы здесь и в конце директивы игнорируются. В конце строки допустим комментарий, начинающийся с символа '#'. Если имена фалов содержат какие-либо ссылки на переменные или функции, подставляются их значения. Смотрите главу 6 [Как использовать переменные].

Например, если у вас есть три '.mk'-файла , 'a.mk', 'b.mk', и 'c.mk', а вместо $(bar) подставляется 'bish bash', то строка make-файла

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

эквивалентна строке

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

Когда make обрабатывает директиву include, он приостанавливает чтение текущего make-файла и считывает по очереди каждый файл, перечисленный в списке. По завершении этого, make продолжает чтение make-файла, в котором появилась директива.

Одной из причин использования директивы include является наличие нескольких программ, которые обрабатываются индивидуальными make-файлами в различных каталогах, и которым требуется общий набор определений переменных (смотрите раздел 6.5 [Установка переменных]) или шаблонных правил (смотрите раздел 10.5 [Определение и переопределение шаблонных правил]).

Еще одной такой причиной является желание использовать автоматическую генерации зависимостей из исходного файла; зависимости могут быть помещены в

файл, который включается в основной make-файл. Такая практика является, вообще говоря, более ясной, чем просто добавление зависимостей в конец основного make-файла, как традиционно делалось в других версиях make. Смотрите раздел 4.12 [Автоматические зависимости].

Если указанное имя не начинается с символа '/' и файл не найден в текущем каталоге, производится поиск еще в нескольких каталогах. Во-первых, поиск производится во всех каталогах, определенных вами с помощью опции '-I' или '--include-dir' (смотрите раздел 9.7 [Обзор опций]). Затем поиск ведется в следующих каталогах: '/usr/local/include' (вместо '/usr/local' может быть другой префикс), '/usr/gnu/include', '/usr/local/include', '/usr/include' (именно в таком порядке).

Если включаемый make-файл не может быть найден ни в одном из этих каталогов, порождается предепреждающее сообщение, но оно само по себе не является фатальной ошибкой - обработка make-файла, содержащего include, продолжается. Завершив чтение make-файлов, make попытается переделать все make-файлы, которые устарели или не существуют. Смотрите раздел 3.5 [Как переделываются make-файлы]. Только после неудачной попытки найти способ переделать make-файл make сообщит об отсутствии файла как о фатальной ошибке.

Если вы хотите, чтобы make просто игнорировал, не сообщая об ошибке, make-файл, который не существует и не может быть переделан, используйте директиву -include вместо include, как показано ниже:

```
-include <имя файла> ...
```

Эта директива действует так, что от нее не будет сообщений об ошибке (даже предупреждений, если любой из указанных файлов не существует).

3.4 Переменная MAKEFILES

Если определена переменная окружения MAKEFILES, make рассматривает ее значение как список имен (разделенных пробелами) дополнительных make-файлов, которые считываются перед другими. Это работает во многом так же, как и директива include: поиск этих файлов производится в различных каталогах (смотрите раздел 3.3 [Включение других make-файлов]). При этом главная цель по умолчанию никогда не берется из этих make-файлов, а при неудачном поиске файлов, указанных в MAKEFILES сообщение об ошибке не порождается.

Основное использование переменной MAINFILES - связь между рекурсивными вызовами make (смотрите раздел 5.6 [Рекурсивное использование программы make]). Обычно нежелательно устанавливать переменные окружения перед вызовом make на верхнем уровне, поскольку лучше не беспокоиться о внутренностях make-файла извне. Однако, если вы запускаете make, не определяя make-файл, make-файл, указанный в переменной окружения $MAKEFILES, может сделать что-нибудь полезное, чтобы улучшить работу встроенных неявных правил, например определить пути поиска (смотрите раздел 4.3 [Поиск по каталогам]).

Некоторые пользователь соблазняются возможностью автоматически устанавливать переменную окружения MAKEFILES при входе в систему, и создают make-файлы в расчете на это. Это очень плохая идея, поскольку такие make-

файлы не смогут работать где-либо еще. Намного лучше явно написать директиву include в make-файле. Смотрите раздел 3.3 [Включение других make-файлов])

3.5 Как переделываются make-файлы

Иногда make-файлы могут быть переделаны из других файлов, таких как RCS- или SCCS-файлы. Если make-файл может быть переделан из других файлов, вы, вероятно, захотите, чтобы make получил свежую версию make-файла для считывания.

Для этого после считывания всех make-файлов make будет рассматривать каждый из них в качестве главной цели и попытается произвести обновление. Если make-файл имеет правило, указывающее, как обновлять его (найденное в том же самом make-файле или в каком-либо другом) или если имеется неявное правило, применимое к нему (смотрите главу 10 [Использование неявных правил]), то он, при необходимости, будет обновлен. После того, как были проверены все make-файлы, в том случае, если какой-нибудь из них на самом деле был изменен, make начинает все с нуля и заново считывает все make-файлы. (Он снова попытается обновить каждый из них, но обычно они уже не изменятся, так как обновление только что было произведено).

Если make-файлы определяют для порождения файла заново правило с двойным двоеточием, у которого есть команды, но нет зависимостей, этот файл всегда будет переделываться (смотрите раздел 4.11 [Двойное двоеточие]). В случае make-файла, make-файл, для которого существует правило с двойным двоеточием, в котором есть команды, но нет зависимостей будет переделываться каждый раз при запуске make, затем - еще раз, когда make начнет считывание с нуля и прочитает заново все make-файлы. Это приведет к бесконечному циклу: make будет постоянно переделавать make-файл, и никогда не займется ничем другим. Таким образом, чтобы избежать этого, make не будет пытаться переделать make-файлы, которые определены как цели правил с двойным двоеточием, но не имеют зависимостей.

Если вы не определяете никакой из make-файлов для считывания при помощи опций '-f' или '--file', make попробует использовать имена make-файлов по умолчанию - смотрите раздел 3.2 [Как назвать make-файл]. В отличие от make-файлов, явно указанных с использованием опций '-f' или '--file', make не уверен, что эти файлы должны существовать. Однако, если make-файл по умолчанию не существует, но может быть создан выполнением правил make, вы , вероятно захотите, чтобы эти правила выполнились, и полученный make-файл мог быть использован.

Следовательно, если ни один из make-файлов по умолчанию не существует, make попробует породить каждый из них в том же порядке, в котором они ищутся (смотрите раздел 3.2 [Как назвать make-файл]), до тех пор пор пока его попытки порождения make-файла не увенчаются успехом или он не переберет все возможные имена. Заметьте, что невозможность найти или породить make-файл не является ошибкой - make-файл не всегда необходим.

При использовании опции '-t' или '--touch', (смотрите раздел 9.3 [Вместо исполнения команд]), вы бы не хотели использовать устаревший make-файл для определения того, какие цели необходимо пометить как обновленные. Таким образом, опция '-t' не оказывает влияния на обновление make-файлов - они обновляются даже тогда, когда она указана. Аналогично, опции '-q' (или '--question') и '-n' (или '--just-print') не

предотвращают обновления make-файлов, поскольку устаревший make-файл привел бы к некорректному результату для других целей. Таким образом, 'make -f mfile -n foo' обновит 'mfile', считает его, и затем напечатает команды для обновления 'foo' и его зависимости без исполнения команд. Команды, печатаемые для 'foo' будут браться из обновленного файла 'mfile'.

Однако, в определенной ситуации, вы действительно могли бы захотеть избежать обновления даже make-файлов. Вы можете сделать это, определив в командной строке в качестве главных целей эти make-файлы, а в качестве make-файлов - их же. Когда имя make-файла явно определено как главная цель, опция '-t' и аналогичные с ней опции будут применены к нему.

Таким образом, 'make -f mfile -n mfile foo' прочитает make-файл 'mfile', напечатает команды, необходимые для его обновления без их выполнения, а затем напечатает команды, необходимые для обновления 'foo' без их выполнения. Команды, печатаемые для 'foo' будут браться из существующего файла 'mfile'.

3.6 Перекрытие части другого make-файла

Иногда полезно иметь make-файл, который большей частью похож на другой make-файл. Часто вы можете использовать директиву 'include' для включения одного файла в другой, и добавлять дополнительные цели или определения переменных. Однако, если два make-файла предложат различные команды для одной и той же цели, make не позволит вам сделать этого. Тем не менее есть еще один способ.

Во включающем make-файле (в make-файле, в который включается другой make-файл) вы можете использовать шаблонное правило произвольного соответствия, чтобы указать программе make, что для порождения целей, которые не могут быть созданы на основе информации из включающего файла, необходимо использовать другой make-файл. Смотрите раздел 10.5 [Шаблонные правила] для дополнительной информации о шаблонных правилах.

Например, если у вас есть make-файл с именем 'Makefile', который указывает, как порождать 'foo' (и другие цели), вы можете написать make-файл с именем 'GNUMakefile', который содержит:

```
foo:
     frobnicate > foo

%: force
          @       $(MAKE) -f Makefile $@
force: ;
```

Если вы напишете в командной строке 'make foo', make найдет 'GNUMakefile', прочитает его и выяснит, что для порождения 'foo' необходимо выполнить команду 'frobnicate > foo'.Если вы напишете в командной строке 'make bar', make не найдет в 'GNUMakefile' способа для порождения 'bar'. Таким образом, он будет использовать команду из шаблонного правила: 'make -f Makefile bar'. Если в 'Makefile' имеется правило для порождения bar, оно будет применено. Аналогично будет обработана любая другая цель, для которой правило порождения не указано в 'GNUMakefile'.

В данном примере осуществляется обработка шаблонного правила с шаблоном '%', который соответствует любой цели. Правило определяет зависимость 'force', чтобы

гарантировать, что команда будет выполнена, даже если цель уже существует. Мы даем цели 'force' пустой набор команд, чтобы предотвратить поиск программой make неявного правила для ее построения - в противном случае все то же правило произвольного соответствия было бы применено к самой цели 'force' и был бы создан цикл зависимости.

4. Написание правил

Правило содержится в make-файле. Оно указывает, когда и как заново порождать определенные файлы, называемые целями правила (чаще всего правилу соответствует только одна цель). В нем перечисляются другие файлы, которые являются зависимостями цели, и команды, используемые для создания или обновления цели.

Порядок правил несущественнен, за исключением определения главной цели по умолчанию: цели, с которой начинается работа make, если вы ее не определили. По умолчанию главной целью make является цель первого правила в первом make-файле. Если в первом правиле есть несколько целей, то только первая цель берется в качестве главной цели по умолчанию. Есть два исключения: цель, начинающаяся с точки, не является главной целью по умолчанию, если она не содержит при этом один или более символа '/'; кроме того, цель, определяющая шаблонное правило, не воздействует на определение главной цели по умолчанию. Смотрите раздел 10.5 [Определение и переопределение шаблонных правил].

Поэтому обычно make-файл пишется так, чтобы первое правило было правилом для компиляции всей программы или всех программ, описываемых make-файлом (часто с именем цели 'all'). Смотрите раздел 9.2 [Аргументы для определения целей].

4.1 Синтаксис правила

В общем виде, правило имеет следующий вид:

```
ЦЕЛИ : ЗАВИСИМОСТИ
     КОМАНДА
     ...
```

или такой:

```
ЦЕЛИ : ЗАВИСИМОСТИ ; КОМАНДА
     КОМАНДА
     ...
```

Слева от двоеточия необходимо указать список имен файлов, разделенных пробелами. При этом могут использоваться шаблонные символы (смотрите раздел 4.2 [Использование шаблонных символов в именах файлов]), а имя в форме 'a(m)' представляет элемент m архивного файла a (Смотрите раздел 11.1 [Элементы архива в качестве целей]). Обычно в правиле присутствует только одна цель, но

иногда есть смысл сделать больше (смотрите раздел 4.8 [Несколько целей в правиле]).

Командные строки начинаются с символа табуляции. Первая команда может появится после строки зависимостей, предваренная символом табуляции, или на той же строке, что и зависимости, предваренная символом ';'. Оба способа имеют одинаковый эффект. Смотрите главу 5 [Написание команд в правилах].

Поскольку знак доллара используется в начале переменных, в том случае, если вы хотите поместить сам по себе знак доллара, напишите '$$' (смотрите главу 6 [Как использовать переменные]). Вы можете разбивать длинную строку путем вставки обратной косой чертой, за которой следует перевод строки, однако это не является обязательным требованием, потому что make не устанавливает ограничения на длину строк в make-файле.

Правило несет два вида информации: когда цели находятся в неактуальном состоянии и как, при необходимости, обновить их.

Критерий неактуального состояния определяется в терминах зависимостей, которые состоят из имен файлов, разделенных пробелами. (Допустимы также шаблонные символы и элементы архивов (Смотрите главу 11 [Элементы архива в качестве целей])). Цель считается неактуальной, если она не существует, или она более старая, чем одна из зависимостей (по результатам сревнения времен последних изменений). Идея состоит в том, что содержимое целевого файла вычисляется на основе информации, содержащейся в зависимостях; таким образом, как только любая из зависимостей изменяется, содержимое существующего целевого файла необязательно будет корректнм.

То, как надо обновлять цели, определяется набором команд. Команды представляют собой строки, которые будут выполнены командной оболочкой, но с некоторыми дополнительными возможностями. (Смотрите главу 5 [Написание команд в правилах]).

4.2 Использование шаблонных символов в именах файлов

При использовании шаблонных символов одно имя файла может определять несколько файлов. Шаблонными символами для make являются '*', '?' и '[...]', как и в командной оболочке Bourne shell. Например, '*.c' определяет список всех файлов (в рабочем каталоге), чьи имена заканчиваются на '.c'.

Символ '~' в начале имени файла также имеет специальное значение. Если он один или за ним следует символ '/', он представляет ваш домашний каталог. Например, '~/bin' означает 'home/you/bin'. Если за символом '~' следует слово, строка представляет домашний каталог пользователя, именем которого является это слово. Например, '~john/bin' означает 'home/john/bin'.

Обработка шаблонов автоматически осуществляется в целях, в зависимостях или в командах (где обработку шаблонов осуществляет командная оболочка). В других ситуациях обработка шаблонов производится только тогда, когда явно ее закажете путем использования функции wildcard.

Специальное значение шаблонного символа выключается предшествующей ему обратной косой чертой. Таким образом, 'foo*bar' ссылается на особый файл, чье имя состоит из 'foo', звездочки и 'bar'.

Примеры шаблонов

Шаблоны могут быть использованы в командах правила, где они обрабатываются командной оболочкой. Вот, например, правило для удаления всех объектных файлов:

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

Шаблоны также полезны в зависимостях правила. При использовании следующего правила в make-файле, 'make print' напечает все '.c'-файлы, которые были изменены с моменты последней их печати:

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

Это правило использует 'print' как пустой целевой файл; смотрите раздел 4.6 [Пустые целевые файлы для фиксации событий]. (Автоматическая переменная '$?' используется для печати тех файлов, которые были изменены; смотрите раздел 10.5.3 [Автоматические переменные].)

Обработка шаблонов не осуществляется в момент определения переменной. Таким образом, если вы напишете:

```
objects = *.o
```

то значением переменной objects будет именно строка '*.o'. Однако, если вы используете значение этой переменной в цели, зависимости или команде, то при обработке соответствующего правила произойдет обработка шаблона. Чтобы установить в objects результат обработки шаблона, используйте следующую конструкцию:

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

Ловушки в использовании шаблонов

Здесь приводится пример наивного использования обработки шаблонов, при которой делается не то, что вы могли бы предположить. Предположим, вы хотели бы указать, что исполняемый файл 'foo' создается из всех объектных файлов каталога, и вы пишете следующее:

```
objects = *.o

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

Значение переменной objects - строка '*.o'. Обработка шаблона происходит в правиле для 'foo', поэтому каждый существующий '.o'-файл становится зависимостью для 'foo' и будет, при необходимости, перекомпилироваться.

Но что будет, если вы удалите все '.o'-файлы? Когда шаблону не соответствует ни один файл, он остается в первозданном виде, и, таким образом, 'foo' будет зависеть от файла со странным именем '*.o'. Поскольку, вероятнее всего, такого файла не существует, make выдаст вам ошибку, говорящую о том, что он не может выяснить, как породить '*.o'. Это не то, чего вы хотите!

На самом деле, достичь желаемого результата при помощи обработки шаблонов возможно, но для этого нужны более развитые методы, включающие в себя функцию wildcard и строковые подстановки. Они описываются в следующем разделе.

Функция wildcard

Обработка шаблонов автоматически осуществляется в правилах. При этом она обычно не производится при установке значения переменной или внутри аргумента функции. Если вы хотите, чтобы в таких ситуациях шаблон был обработан, вам нужно использовать функцию wildcard, например:

```
$(wildcard ШАБЛОН...)
```

Эта строка, будучи использованной в любом месте make-файла, заменяется на разделенный пробелами список имен существующих файлов, соответствующих одному из данных шаблонов имени файла. Если ни один существующий файл не удовлетворяет шаблону, то шаблон не включается в вывод функции wildcard. Обратите внимание, что это отличается от того, как обрабатываются шаблоны без соответствий в правилах make-файла, где они не игнорируются, а используются в первоначальном виде (смотрите раздел 4.2.2 [Ловушки в использовании шаблонов]).

Одно из использований функции wildcard - получение списка всех исходных C-файлов каталога, что делается следующим образом:

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

Мы может заменить список исходных C-файлов на список объектных файлов путем замены в результате функции суффикса '.c' на '.o', как показано ниже:

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

(Здесь мы использовали еще одну функцию, patsubst. Смотрите раздел 8.2 [Функции подстановки и анализа строк].)

Таким образом, make-файл для компиляции всех исходных C-файлов в каталоге и последующей их компоновки мог бы быть написан следующим образом:

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

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

(Здесь используются преимущества неявного правила для компиляции C-программ, поэтому нет необходимости писать явные правила для компиляции файлов. Смотрите раздел 6.2 [Две разновидности переменных] для объяснения знака ':=', являющегося вариантом знака '='.)

4.3 Поиск зависимостей по каталогам

Для больших систем часто является желательным располагать исходные файлы в отдельных каталогах от двоичных файлов. Возможности поиска по каталогам программы make способствуют этому посредством автоматического поиска в некоторых каталогах для нахождения файла зависимости. Когда вы перераспределяете файлы по каталогам, вам не требуется изменять отдельные правила, достаточно изменить пути поиска.

VPATH: Путь поиска для всех зависимостей

Значение переменной программы make VPATH определяет список каталогов, в которых следует осуществлять поиск. Чаще всего предполагается, что в этих каталогах содержатся файлы зависимостей, которых нет в текущем каталоге, однако, VPATH определяет список путей поиска, который make применяет ко всем файлам, включая файлы, являющиеся целями правил.

Таким образом, если файл, упомянутый как цель или зависимость, не существует в текущем каталоге, make ищет файл с таким именем в каталогах, перечисленных в VPATH. Если в одном из них такой файл найден, то он становится зависимостью. Таким образом, правила могут указывать среди зависимостей имена исходных файлов, как если бы они они все существовали в текущем каталоге. Смотрите раздел 4.3.3 [Написание команд командной оболочки с учетом поиска по каталогам].

В переменной VPATH имена каталогов разделяются двоеточиями или пробелами. При поиске make перебирает каталоги в том порядке, в котором они перечислены.

Например,

```
VPATH = src:../headers
```

определяет пути поиска, включающие два каталога, 'src' и '../headers', которые make будет в таком порядке перебирать при поиске.

При этом значении VPATH следующее правило:

```
foo.o : foo.c
```

интерпретируется так, как будто оно написано так:

```
foo.o : src/foo.c
```

при условии, что файл 'foo' не существует в текущем каталоге, но найден в каталоге 'src'.

Директива vpath

Средством, аналогичным переменной VPATH, но более гибким, является директива vpath (обратите внимание на маленькие буквы), которая позволяет определить путь поиска для определенного класса имен файлов, удовлетворяющих определенному шаблону. Таким образом, вы можете выделить некоторые каталоги поиска для одного класса имен файлов, а другие (или никаких) - для других имен файлов.

Есть три формы директивы VPATH:

vpath ШАБЛОН КАТАЛОГИ

Определяет пути поиска для имен файлов, соответствующих ШАБЛОНу. КАТАЛОГИ представляют собой список каталогов для поиска, разделенный двоеточиями или пробелами, по аналогии с путями поиска, используемыми в переменной VPATH

vpath ШАБЛОН

Очищает пути поиска, связанные с ШАБЛОНом

vpath

Очищает все пути поиска, ранее назначенные директивами vpath

Шаблон vpath является строкой, содержащей символ '%'. Имя файла зависимости, поиск которого осуществляется, должно соответствовать этой строке, причем символ '%' соответствует любой последовательности, содержащей нуль или более символов (как в шаблонных правилах; смотрите раздел 10.5 [Определение и переопределение шаблонных правил]). Например, '%.h' соответствует файлам, которые заканчиваются на .h. (Если нет символа '%', то зависимость должна точно соответствовать шаблону, что бывает нужным не очень часто).

Специальное назначение символа '%' в шаблоне директивы vpath может быть отменено предшествующим символом '\'. Специальное назначение символа '\', который в противном случае отменял бы специальное назначение последующего символа '%', может быть отменено еще одним символом '\'. Символы '\', отменяющие специальное назначение символов '%' или других символов '\', удаляются из шаблона перед тем, как ему будут сопоставляться имена файлов. Символы '\', которые заведемо не влияют на трактовку символа '%', остаются нетронутыми.

Если файл зависимости в текущем каталоге не существует, то в том случае, когда его имя соответствует шаблону из директивы vpath, поиск осуществляется в каталогах, указанных в той директиве, как если бы они были упомянуты в переменной VPATH (причем, поиск в этих каталогах осуществляется перед поиском в тех каталогах, которые на самом деле указаны в переменной VPATH).

Например, строка

```
vpath %.h ../headers
```

указывает программе make искать любой файл зависимости, чье имя заканчивается на '.h' в каталоге '../headers', если такой файл не найден в текущем каталоге.

Если имя файла зависимости удовлетворяет нескольким шаблонам vpath, make обрабатывает одну за другой каждую подходящую директиву vpath, осуществляя поиск во всех каталогах, упомянутых в каждой такой директиве. make обрабатывает несколько директив vpath в том порядке, в котором они появляются в make-файле; несколько директив с одинаковым шаблоном не влияют друг на друга.

Таким образом, этот фрагмент

```
vpath %.c foo
vpath %   blish
vpath %.c bar
```

означает поиск файла, оканчивающегося на '.c' в каталоге 'foo', затем 'blish', затем 'bar', в то время как этот фрагмент

```
vpath %.c foo:bar
vpath %   blish
```

означает поиск файла, оканчивающегося на '.c' в каталоге 'foo', затем 'bar', затем 'blish'.

Написание команд командной оболочки с учетом поиска по каталогам

Когда зависимость найдена в результате поиска по каталогам в каталоге, отличном от текущего, команды в правиле не изменяются - они будут исполнены так, как они написаны. Поэтому вам следует внимательно писать команды с тем, чтобы они искали зависимости в тех же каталогах, где их находит make.

Это делается с помощью автоматических переменных, таких как '$^' (смотрите раздел 10.5.3 [Автоматические переменные]). Например, значением '$^' является список всех зависимостей правила, включая имена каталогов, в которых они были найдены, а значением '$@ ' - цель. Например:

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

(Переменная CFLAGS существует для тог, чтобы вы могли определить флаги для C-компиляции посредством неявных правил. Мы используем ее из соображений последовательности, в результате чего она будет одинаково влиять на C-компиляцию. Смотрите раздел 10.3 [Переменные, используемые неявными правилами].)

Часто зависимости также включают в себя заголовочные файлы, которые вы не хотите упоминать в команде. Автоматическая переменная '$<' является просто первой зависимостью.

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

Поиск по каталогам и неявные правила

Поиск в каталогах, определенных в переменной VPATH или при помощи директивы vpath происходит также в случае неявных правил (смотрите главу 10 [Использование неявных правил])

Например, если файл 'foo.o' не имеет явных правил, make рассматривает неявные правила, такие как встроенное правило для компиляции 'foo.c', если такой файл существует. Если такого файла нет в текущем каталоге, то он ищется в соответствующих каталогах для поиска. Если файл 'foo.c' существует (или упоминается в make-файле) в любом из каталогов, применяется неявное правило для C-компиляции.

Командам из неявных правил обычно необходимо пользоваться автоматическими переменными; следовательно, они будут использовать имена файлов, найденных в результате поиска по каталогам без каких-либо дополнительных усилий с вашей стороны.

Поиск по каталогам библиотек для компоновки

Поиск по каталогам библиотек, используемых компоновщиком, применяется особым образом. Эта специфическая особенность вступает в силу, когда вы пишете зависимость, имя которой имеет форму '-l<имя файла>'. (Вы можете сказать, что здесь происходит что-то странное, поскольку зависимость обычно является именем файла, а имя библиотечного файла имеет вид 'lib<имя файла>.a', а не '-l<имя файла>'.)

Когда имя зависимости имеет форму '-l<имя файла>', make специально обрабатывает его, устраивая поиска файла 'lib<имя файла>.a' в текущем каталоге, в каталогах, определенных путями поиска, соответствующими шаблонам директивы vpath и путями поиска из переменной VPATH, а затем в каталогах '/lib', '/usr/lib' и <префикс>/lib (обычно '/usr/local/lib').

Например, правило

```
foo : foo.c -lcurses
      cc $^ -o $@
```

вызовет исполнение команды 'cc foo.c /usr/lib/libcurses.a -o foo', если 'foo' более старый, чем 'foo.c' или '/usr/lib/libcurses.a'.

4.4 Цели-имена действий

Цель-имя действия представляет собой цель, которая на самом деле не является именем файла. Это просто наименование некоторых команд, которые будут исполняться при явном запросе. Есть две причины использования целей-имен действий: для избежания конфликта с файлом, имеющим такое же имя, и для улучшения производительности.

Если вы пишете правило, команды которого не будут создавать целевой файл, команды будут выполняться каждый раз, когда придет время порождать цель. Вот пример:

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

Поскольку команда rm не создает файл с именем 'clean', вероятно такой файл никогда не будет существовать.

Цель-имя действия прекратит работу, если кто-нибудь когда-нибудь создаст в этом каталоге файл 'clean'. По причине отсутствия зависимостей, файл 'clean' непременно будет считаться свежим, и команды из соответствующего правила не выполняться не будут. Чтобы избежать этой проблемы, вы можете явно объявить цель как имя действия, используя специальную цель .PHONY (смотрите раздел 4.7 [Специальные встроенные имена целей]), как показано ниже:

```
.PHONY : clean
```

Как только это сделано, 'make clean' выполнит команды, независимо от того, есть ли файл с именем 'clean'.

Зная, что цель-имя действия не именует настоящий файл, который может быть переделан из других файлов, make пропускает поиск неявного правила для цели-имени действия (смотрите главу 10 [Использование неявных правил]). Именно поэтому объявление цели-имени действия положительно сказывается на производительности, даже если вас не беспокоит возможное существование настоящего файла с таким именем.

Таким образом, вы сначала пишете строку, которая устанавливает clean в качестве цели-имени действия, а затем пишете правило, как показано ниже:

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

Цель-имя дествия не должна быть зависимостью реального целевого файла; если это так, ее команды выполняются каждый раз, когда make обновляет этой файл. До тех пор, пока цель-имя действия не является зависимостью никакой реальной цели, ее команды будут исполняться только тогда, когда цель-имя действия определена как главная цель (смотрите раздел 9.2 [Аргументы для определения целей]).

Цели-имена действий могут иметь зависимости. Когда в одном каталоге содержится много программ, наиболее удобно описать все программы в одном

make-файле './Makefile'. Поскольку заново порождаемой целью по умолчанию станет первая цель в make-файле, удобно сделать ею цель-имя действия под названием 'all' и дать ей в качестве зависимостей все отдельные 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 вы можете просто набрать в командной строке 'make', чтобы переделать все три программы или определить в качестве аргументов те, что необходимо переделать (например, 'make prog1 prog2').

Когда одна цель-имя действия является зависимостью другой, она служит в качестве ее подпрогораммы. Например, здесь 'make cleanall' удалит объектные файлы, файлы различий и файл 'program'.

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm program

cleanobj :
        rm *.o

cleandiff :
        rm *.diff
```

4.5 Правила без команд и зависимостей

Если правило не имеет зависимостей или команд, и цель правила - несуществующий файл, то make работает в предположении, что эта цель обновляется всегда, когда правило исполняется. Это подразумевает, что для всех целей, зависящих от нее, всегда будут выполняться команды из соответствующих правил.

Проиллюстрируем это примером:

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

Здесь правило для цели 'FORCE' удовлетворяет указанным условиям, поэтому цель 'clean', зависящая от нее, вынуждена выполнять свои команды. В имени 'FORCE' нет ничего специального, однако это имя часто используется в таких случаях.

Как можно видеть, использование 'FORCE' таким способом дает такой же результат, как и использование '.PHONY : clean'.

Использование '.PHONY' более наглядно и более эффективно. Однако, другие версии make не поддерживают '.PHONY', поэтому 'FORCE' появляется во многих make-файлах. Смотрите раздел 4.4 [Цели-имена действий].

4.6 Пустые целевые файлы для фиксации событий

Пустая цель является вариантом цели-имени действия; она используется для того, чтобы хранить набор команд для действий, которые вы время от времени явно запрашиваете. В отличие от цели-имени действия, этот целевой файл может реально существовать, но его содержимое несущественно, и обычно он пуст.

Предназначение пустого целевого файла - зафиксировать, с помощью времени его последней модификации, когда в последний раз исполнялись команды из правила. Это делается при помощи включения в набор команд команды touch для обновления целевого файла.

Пустой целевой файл должен иметь несколько зависимостей. Когда вы заказываете очередное порождение пустой цели, команды будут выполняться, если какая-нибудь из зависимостей более актуальна, чем цель; другими словами, если зависимость была изменена с того момента, как вы в последний раз породили цель. Вот пример:

```
print: foo.c bar.c
        lpr -p $?
        touch print
```

Согласно этому правилу, 'make print' выполнит команду lpr, если какой-нибудь исходный файл был изменен с момента последнего 'make print'. Автоматическая переменная '$?' используется для того, чтобы печатать только те файлы, которые изменились (смотрите раздел 10.5.3 [Автоматические переменные]).

4.7 Специальные встроенные имена целей

Некоторые имена, появляясь в качестве целей, имеют специальное значение.

.PHONY

Зависимости специальной цели .PHONY рассматриваются как цели-имена действий. Когда придет время рассматривать такую цель, make выполнит команды в безусловном режиме, независимо от того, существует ли такой файл и от того,

какого время его последней модификации. Смотрите раздел 4.4 [Цели-имена действий].

.SUFFIXES

Зависимости специальной цели .SUFFIXES представляют собой список суффиксов, которые будут использоваться при проверке суффиксных правил. Смотрите раздел 10.7 [Устаревшие суффиксные правила].

.DEFAULT

Команды, определенные для .DEFAULT, используются с любыми целями, для которых не найдено правил (как явных, так и неявных). Смотрите раздел 10.6 [Последняя возможность]. Если определены команды для .DEFAULT, то они будут исполняться для каждого файла, упомянутого в качестве зависимости, но не являющегося целью какого-либо правила. Смотрите раздел 10.8 [Алгоритм поиска неявного правила].

.PRECIOUS

Цели, от которых зависит .PRECIOUS, подвергаются специальной обработке: если make уничтожается или прерывается при выполнении соответствующих им команд, цель не удаляется. Смотрите раздел 5.5 [Прерывание или уничтожение программы make]. Кроме того, если цель представляет собой промежуточный файл, он не будет удален после того, как необходимость в нем отпала, как это обычно делается. Смотрите раздел 10.4 [Цепочки неявных правил].

Вы можете также указать шаблон цели неявного правила (как, например, '%.o') в качестве файла зависимости специальной цели .PRECIOUS, чтобы сохранить промежуточные файлы, созданные посредством правил, шаблонам целей которых соответствуют имена этих файлов.

.IGNORE

Если вы определяете зависимости для .IGNORE, то make будет игнорировать ошибки при выполнении команд, запускаемых для этих особых файлов. Команды для .IGNORE роли не играют.

Если .IGNORE определяется как цель без зависимостей, это значит, что необходимо игнорировать ошибки при выполнении команд для всех файлов. Такое использование .IGNORE поддерживается только для исторической совместимости. Этот прием не очень полезен, поскольку он воздействует на любую команду в make-файле; мы рекомендуем вам использовать более гибкие способы игнорирования ошибок в отдельных командах. Смотрите раздел 5.4 [Ошибки в командах].

.SILENT

Если вы определяете зависимости для .SILENT, то make не будет перед выполнением команд, используемых для переделывания этих особых файлов, печатать соответствующие команды. Команды для .SILENT роли не играют.

Если .IGNORE определяется как цель без зависимостей, это значит, что необходимо подавлять печать любой команды перед ее выполнением. Такое использование .SILENT поддерживается только для исторической совместимости. Мы рекомендуем вам использовать более гибкие способы подавления печати перед выполнением отдельных команд. Смотрите раздел 5.1 [Отображение команды]. Если вы хотите подавить печать всех команд при определенном запуске make, используйте опцию '-s' или '--silent' (смотрите раздел 9.7 [Обзор опций]).

.EXPORT_ALL_VARIABLES

Будучи просто упомянутой в качестве цели, указывает программе make по умолчанию экспортировать порожденным процессам все переменные. Смотрите раздел 5.6.2 [Связь порожденным процессом make через переменные].

В качестве специальной цели рассматривается также любой суффикс из определения неявного правила, так же как и конкатенация двух суффиксов, как, например, '.c.o'. Эти цели представляют собой суффиксные правила, устаревший способ определения неявных правил (однако, все еще широко распространенный). В принципе, любое имя цели могло бы таким образом стать специальным, если бы вы разбили его на две части и добавили обе к списку суффиксов. На практике же суффиксы обычно начинаются с '.', поэтому эти специальные имена цели также начинаются с '.'. Смотрите раздел 10.7 [Устаревшие суффиксные правила].

4.8 Несколько целей в правиле

Правило с несколькими целями эквивалентно написанию нескольких правил, каждое из которых имеет одну цель, и идентичных во всем остальном. Ко всем целям применяются одни и те же команды, но их действия могут меняться, поскольку вы можете подставлять в команду конкретное имя цели, используя '$@ '. Правило также распространяет действие всех зависимостей на все цели этого правила.

Это полезно в двух случаях.

- Вам нужны только зависимости, а не команды. Например, строка

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

  дает дополнительную зависимость для каждого из трех упомянутых объектных файлов

- Для получения всех целей используются похожие команды. От команд не требуется быть абсолютно идентичными, поскольку можно использовать автоматическую переменную '$@ ' для подстановки в команду конкретной заново порождаемой цели (смотрите раздел 10.5.3 "Автоматические переменные"). Например, правило

  ```
  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 предполагается, что гипотетическая программа generate осуществляет два типа вывода, один при использовании параметра '-big', другой - при использовании '-little'. Для объяснения функции subst смотрите раздел 8.2 "Функции подстановки и анализа строк"
Допустим, вы хотели бы изменять зависимости в соответствии с целью аналогично тому, как переменная '$@ ' позволяет вам изменять команды. Вы не можете сделать этого при использовании обычного правила с несколькими целями, но это можно сделать при помощи статического шаблонного правила. Смотрите раздел 4.10 "Статические шаблонные правила".

4.9 Несколько правил для одной цели

Один файл может быть целью нескольких правил. Все зависимости, упомянутые во всех таких правилах, образуют общий список зависимости для данной цели. Если цель обновлялась в последний раз раньше, чем какая-либо из зависимостей какого-либо правила, выполняются команды.

Для файла может исполнен только один набор команд. Если более, чем одно правило дает команды для одного и того же файла, make использует последний встретившийся набор команд и выдает сообщение об ошибке. (Как исключение, если имя файла начинается с точки, сообщение об ошибке не печатается. Такое странное поведение поддерживается только для совместимости с другими реализациями make). У вас нет причин писать make-файлы таким образом, поэтому make выдает вам сообщение об ошибке.

Дополнительное правило, содержащее только зависимости, может быть использовано для добавления нескольких дополнительных зависимостей одновременно к нескольким файлам. Например, обычно имеется переменная с именем objects, содержащая список всех файлов, являющихся выходом компилятора в порождаемой системе. Легкий способ указать, что все они должны быть перекомпилированы, если изменился файл 'config.h' - написать следующее :

```
objects = foo.o bar.o
foo.o : defs.h
bar.o : defs.h test.h
$(objects) : config.h
```

Это могло быть вставлено или убрано без изменения правил, которые действительно определяют, как порождать объектные файлы, что является удобным для использования способом, если вы хотите в произвольном месте добавлять дополнительные зависимости.

Еще один полезный совет состоит в том, что дополнительные зависимости могут быть определены при помощи переменной, которую вы устанавливаете в аргументе командной строки для make. (смотрите раздел 9.5 [Перекрывающиеся переменные]). Например, правило

```
extradeps=
$(objects) : $(extradeps)
```

означает, что команда 'make extradeps=foo.h' будет рассматривать 'foo.h' в качестве зависимости для каждого объектного файла, а просто 'make' - не будет.

Если ни одно из явных правил для цели не имеет команд, то make организует поиск применимого неявного правила, чтобы найти какие-нибудь команды (смотрите главу 10 [Использование неявных правил]).

4.10 Статические шаблонные правила

Статические шаблонные правила - это правила, которые определяют несколько целей и создают имена зависимостей для каждой цели на основе имени цели. Они являются более общими, чем обычные правила с несколькими целями, поскольку цели не должны иметь одинаковые зависимости. Их цели должны быть похожими, но не обязательно одинаковыми.

Синтаксис статических шаблонных правил

Здесь приведен синтаксис статического шаблонного правила:

```
ЦЕЛИ ...: ШАБЛОН ЦЕЛИ: ШАБЛОНЫ ЗАВИСИМОСТЕЙ ...
        КОМАНДЫ
        ...
```

Список ЦЕЛЕЙ определяет цели, к которым применяется правило. Цели могут содержать шаблонные символы, так же как и цели обычных правил (смотрите раздел 4.2 [Использование шаблонных символов в именах файлов]).

ШАБЛОН ЦЕЛИ и ШАБЛОНЫ ЗАВИСИМОСТЕЙ указывают, как вычислять зависимости каждой цели. К каждой цели применяется шаблон цели для получения части имени цели, называемой основой. Эта основа подставляется в каждый из шаблонов зависимостей, в результате чего порождаются имена зависимостей (по одному из каждого шаблона зависимости).

Обычно каждый шаблон содержит ровно один символ '%'. Когда цель сопоставляется шаблону цели, символ '%' соответствует незафиксированной части имени цели - эта часть называется основой. Оставшаяся часть шаблона должна точно соответствовать имени цели. Например, цель 'foo.o' удовлетворяет шаблону '%.o', при этом 'foo' является основой. Цели 'foo.c' и 'foo.out' не удовлетворяют шаблону.

Имена зависимостей для каждой цели соэдаются путем подстановки основы вместо символа '%' в каждый шаблон зависимости. Например, если единственный шаблон зависимости - '%.c', то подстановка основы 'foo' дает имя зависимости

'foo.c'. Является законным написание шаблона зависимости, не содержащего '%' - в таком случае данная зависимость одинакова для всех целей.

Специальное назначение символа '%' в шаблоне правила может быть отменено предшествующим символом '\'. Специальное назначение символа '\', который в противном случае отменял бы специальное назначение последующего символа '%', может быть отменено еще одним символом '\'. Символы '\', отменяющие специальное назначение символов '%' или других символов '\', удаляются из шаблона перед тем, как он будет сравниваться с именами файлов или в него будет подставляться основа. Символы '\', которые заведомо не влияют на трактовку символа '%', остаются нетронутыми. Например, в шаблоне 'the\%weird\\%pattern\\', фрагмент 'the%weird\' предшествует действующему символу '%', а фрагмент 'pattern\\' следует за ним. Последние два символа '\' остаются на месте, поскольку они не могут воздействовать ни на какой символ '%'.

Вот пример, который компилирует или 'foo.o' или 'bar.o' из соответствующего '.c'-файла:

```
objects = foo.o bar.o

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

Здесь '$<' является автоматической переменной, которая содержит имя зависимости, а '$@ ' - автоматической переменной которая содержит имя цели; смотрите раздел 10.5.3 [Автоматические переменные].

Каждая специфицированная цель должна соответствовать шаблону цели - для каждой цели, которая не соответствует, выдается предупреждение. Если у вас есть список файлов, из которого только некоторые будут соответствовать шаблону, вы можете использовать функцию filter для отсечения несоответствующих имен файлов (смотрите раздел 8.2 [Функции подстановки и анализа 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 шаблонное правило вызывает обновление каждого из этих объектных файлов путем компиляции соответствующих исходных 'C'-файлов. Результатом '$(filter %.elc,$(files))' является 'foo.elc', поэтому этот файл порождается из 'foo.el'.

Еще один пример показывает, как использовать '$*' в статических шаблонных правилах:

```
bigoutput littleoutput : %output : text.g
        generate text.g -$* > $@
```

Статические шаблонные правила в сравнении с неявными правилами

Статические шаблонные правила имеют много общего с неявными правилами, определенными как шаблонные правила (смотрите раздел 10.5 [Определение и переопределение шаблонных правил]). В обоих случаях имеется шаблон для цели и шаблоны для построения имен зависимостей. Различие заключается в том, как make определяет, когда применяются правила.

Неявное правило может быть применено к любой цели, которая соответствует его шаблону, но оно применяется только тогда, когда цель не имеет команд, определенных иным способом и когда могут быть найдены зависимости. Если применяемым оказывается более одного неявного правила, применяется только одно - его выбор зависит от порядка правил.

Напротив, статическое шаблонное правило применяется именно к тому списку целей, который вы определяете в правиле. Оно не может применяться ни к какой другой цели и неизменно применяется к каждой из специфицированных целей. Если применимы два конфликтующих правила, и оба имеют команды, это является ошибкой.

Статическое шаблонное правило может быть предпочтительнее неявного правила по следующим причинам:

- Вы можете захотеть перекрыть обычное неявное правило для нескольких файлов, чьи имена не могут быть синтакисически выделены в отдельную категорию, но могут быть представлены явным списком.
- Если не можете быть уверены в точном содержимом каталога, который вы используете, вы не можете быть уверены, в том, что какие-нибудь не относящиеся к вашей деятельности файлы, не приведут make к использованию некорректных неявных правил. Выбор может зависеть от порядка, в котором осуществляется поиск неявного правила. При использовании статических шаблонных правил ненадежность отсутствует: каждое правило применяется именно к определенным целям.

4.11 Правила с двумя двоеточиями

Правила с двойным двоеточием представляют собой правила, записываемые при помощи '::', а не ':' после имени цели. Их обработка отличается от обработки обработки обычных правил в том случае, когда одна и та же цель появляется более, чем в одном правиле.

Когда цель появляется в нескольких правилах, все правила должны быть одного типа: все обычные или все с двойным двоеточием. Если они с двойным двоеточием, то каждое из них не зависит от других. Команды каждого правила с двойным двоеточием выполняются, если цель обновлялась раньше, чем какая-либо зависимость из этого правила. Это может привести к выполнению любого или всех из правил с двойным двоеточием, а также к невыполнению ни одного такого правила.

Правила с двойным двоеточием, имеющие одну и ту же цель фактически полностью отделены одно от другого. Каждое правило с двойным двоеточием, обрабатывается индивидуально, так же, как обрабатываются правила с с различными целями.

Правила с двойным двоеточием для цели выполняются в том порядке, в котором они появляются в make-файле. Однако, в тех случаях, когда действительно есть смысл в правилах с двойным двоеточием, порядок выполнения команд не играет роли.

Правила с двойным двоеточием являются несколько неяными и нечасто приносят большую пользу - они обеспечивают механизм для тех случаев, в которых метод, используемый для обновления цели, отличается в зависимости от того, какие файлы зависимости вызвали обновление, а такие случаи являются редкими.

Каждое правило с двойным двоеточием должно определять команды - если они не определены, будет использовано неявное правило, в случае его применимости. Смотрите главу 10 [Использование неявных правил].

4.12 Автоматическая генерация зависимостей

В make-файле для программы, многие из правил, которые вам приходится писать, указывают только на то, что некоторый объектный файл зависит от некоторого заголовочного файлов. Например, если 'main.c' использует 'defs.h' посредством директивы #include, вы бы написали:

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

Вам это правило нужно для того, чтобы make знал, что он должен обновлять 'main.o' всегда, когда изменяется 'defs.h'. Можно заметить, что для большой программы вы бы написали в вашем make-файле десятки таких правил. Кроме того, вы всегда должны быть очень внимательны в отношении обновления make-файла каждый раз, когда вы добавляете или удаляете директиву #include.

Чтобы избавиться от этого неудобства, большинство современных C-компиляторов могут написать для вас эти правила, просмотрев в исходном файле строки с директивой #include. Обычно это делается при помощи опции компилятора '-M'. Например, команда:

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

генерирует на выходе:

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

Таким образом, вам больше не требуется самим писать все такие правила. Компилятор сделает это за вас.

Обратите внимание, что такая зависимость порождает упоминание 'main.o' в make-файле, таким образом он впоследствии не может рассматриваться как промежуточный файл при поиске неявного правила. Это означает, что make никогда не будет удалять файл после его использования - смотрите раздел 10.4 [Цепочки неявных правил].

В старых программах make традиционной практикой было использование возможности компилятора генерировать зависимости с помощью команды вида 'make depend'. Эта команда создавала файл 'depend', содержащий все

автоматически генерируемые зависимости - в таком случае make-файл мог использовать директиву include для их чтения (смотрите раздел 3.3 [Включение]).

В GNU-версии программы make из-за возможности обновления make-файлов такая практика становится устаревшей - вам никогда не требуется явно указывать программе make перегенерировать зависимости, поскольку она всегда перегенерирует любой make-файл, который является необновленым. Смотрите раздел 3.5 [Как переделываются make-файлы].

Рекомендуемая нами практика автоматической генерации зависимостей заключается в том, чтобы иметь для каждого исходного файла соответствующий make-файл. Для каждого исходного файла '<имя файла>.c' заводится make-файл '<имя файла>.d', в котором перечисляются файлы, от которых зависит объектный файл '<имя файла>.o'. В таком случае для порождения новых зависимостей требуется заново просмотреть только измененный файл.

Вот шаблонное правило для порождения файла зависимостей (т.е. make-файла) с именем '<имя файла>.d' из исходного C-файла с именем '<имя файла>.c':

```
%.d: %.c
        $(SHELL) -ec '$(CC) -M $(CPPFLAGS) $< \
                      | sed '\''s/$*\\.o[ :]*/& $@/
g'\'' > $@'
```

Смотрите раздел 10.5 [Шаблонные правила] для информации об определении шаблонных правил. Благодаря флагу командной оболочки '-e', непосредственно после неудачного выполнения команды $(CC) (завершения с ненулевым результатом) происходит завершение работы оболочки. В противном случае командная оболочка завершалась бы с результатом последней команды в конвейере (в данном случае sed), поэтому программа make не замечала бы ненулевой результат компилятора.

При использовании компилятора GNU C вы можете захотеть вместо флага '-M' использовать флаг '-MM'. Его использование приводит к пропуску зависимостей от системных заголовочных файлов. Подробности смотрите в разделе "Опции управления препроцессором" руководства по использованию GNU CC.

Предназначением программы sed является преобразование (например):

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

в:

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

Это делает каждый '.d'-файл зависимым от всех исходных и заголовочных файлов, от которых зависит соответствующий '.o'-файл. В этом случае make знает, что всегда, когда изменяется любой из исходных или заголовочных файлов, требуется перегенерировать зависимости.

После того, как вы определили правило для обновления '.d'-файлов, вы используете директиву include для чтения их всех. Смотрите раздел 3.3 [Включение]. Например:

```
sources = foo.c bar.c

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

(В этом примере для преобразования списка исходных файлов 'foo.c bar.c' в список make-файлов с зависимостями 'foo.d bar.d' используется ссылка на переменную с заменой. Смотрите раздел 6.3.1 [Ссылки с заменой] для полной информации о подстановочных ссылках). Так как '.d'-являются такими же make-файлами, как и любые другие, make при необходимости обновит их без какой-либо дополнительной работы с вашей стороны. Смотрите раздел 3.5 [Как переделываются make-файлы].

5. Написание команд в правилах

Команды правила состоят из командых строк командной оболочки, предназначенных для выполнения их одной за другой. Каждая командная строка должна начинаться с символа табуляции, за исключением того, что первая командная строка может быть присоединена к строке целей и зависимостей через точку с запятой. Среди командных строк могут появляться пробельные строки и строки, состоящие только из комментария - они игнорируются. (Но будьте осторожны - кажущаяся пробельной, сторока, начинающаяся с символа табуляции, не является пробельной! Это пустая команда - смотрите раздел 5.8 [Пустые команды]).

Пользователи используют в качестве программных оболочек различные программы, но команды из make-файла всегда интерпретируются программой '/bin/sh', если в make-файле не определена другая. Смотрите раздел 5.2 [Выполнение команд].

Используемая командная оболочка определяет, могут ли быть написаны комментарии в командных строках, и какой синтаксис они используют. Когда командной оболочкой является '/bin/sh', символ '#' начинает комментарий, который продолжается до конца строки. Символ '#' не обязательно должен быть в начале строки. Текст строки перед символом '#' не является частью комментария.

5.1 Отображение команды

Обычно make печатает каждую командную строку перед ее выполнением. Мы называем это отображением, поскольку создается впечатление, что вы сами набираете команду.

Когда строка начинается с '@', отображение этой строки подавляется. Символ '@' отбрасывается, прежде чем команда передается командной оболочке. Обычно вам следует использовать это для тех команд, единственным действием которых является вывод чего-либо на экран, например для команды echo, применяемой в целях индикации прохождения по make-файлу:

```
@echo   About to make distribution files
```

Когда программе make передается опция '-n' или '--just-print', происходит только отображение, без выполнения. Смотрите раздел 9.7 [Обзор опций]. В этом и только в этом случае печатаются даже команды, начинающиеся с символа '@ '. Эта опция полезна для выяснения того, какие команды make рассматривает как необходимые, без реального их выполнения.

Опции программы make '-s' или '--silent' предотвращают все отображения, как будто все команды начинаются с '@ '. Появление в make-файле правила без зависимостей для специальной цели .SILENT дает такой же эффект (смотрите раздел 4.7 [Специальные встроенные имена целей]). .SILENT представляет собой явно устаревшую возможность, так как использование символа '@ ' является более гибким.

5.2 Выполнение команд

Когда настает время выполнять команды для обновления цели, они выполняются путем создания новой командной подоболочки для каждой строки. (На практике, make может осуществлять сокращения, которые не будут влиять на результаты.)

ЗАМЕЧАНИЕ: это означает, что команды командной оболочки, такие, как cd, которая устанавливает для каждого процесса местоположение его данных, не будут воздействовать на следующие командные строки. Если вы хотите, чтобы команда cd воздействовала на следующую команду, разместите обе на одной строке с точкой с запятой между ними. В этом случае make будет рассматривать их как одну команду и передаст их вместе командной оболочке, которая последовательно выполнит их. Например:

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

Если вам хотелось бы разбить одну команду командной оболочки на несколько тексовых строк, вы должны использовать символ '\' в конце всех ее подстрок, кроме последней. Такая последовательность текстовых строк объединяется в одну посредством удаления последовательностей 'обратная косая черта - перевод строки' перед передачей ее командной оболочке. Таким образом, написанное ниже эквивалентно предыдущему примеру:

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

Программа, используемая в качестве командной оболочки, определяется при помощи переменной SHELL. По умолчанию, используется программа '/bin/sh'.

В отличие от большинства переменных, переменная SHELL никогда не устанавливается из командной среды. Так дело обстоит из-за того, что переменная окружения SHELL используется для определения вашей личной настройки программы командной оболочки для диалогового использования. Было бы очень плохо, если бы такого рода личная настройка влияла на функционирование make-файлов. Смотрите раздел 6.9 [Переменные из командной среды].

5.3 Параллельное выполнение

GNU-версия программы make умеет одновременно выполнять несколько команд. Обычно make будет выполнять одновременно только одну команду, ожидая ее завершения, прежде, чем запускать следующую. Однако, опция '-j' или '--jobs' дает программе make указание выполнять параллельно несколько команд.

Если за опцией '-j' следует целое число, то оно является количесивом команд, выполняемых одновременно - это число называется количеством рабочих гнезд. Если после опции '-j' нету ничего, напоминающего целое число, это означает, что на количество рабочих гнезд ограничений нет. По умолчанию количество рабочих гнезд равно одному, что означает последовательное выполнение (одна команда в каждый момент времени).

Одним из неприятных следствий параллельного выполнения нескольких команд является то, что выходные данные от всех команд приходят тогда, когда команды посылают их, поэтому сообщения от разных команд могут быть перемешаны.

Еще одна проблема заключается в невозможности приема двумя процессами входных данных из одного и того же устройства - таким образом, для того, чтобы убедиться, что одновременно только одна команда пытается принимать входные данные с терминала, make закроет стандартные входные потоки всех выполняемых команд, кроме одной. Это означает, что попытка чтения со сандартного входа обычно обернется фатальной ошибкой (сигнал 'Broken pipe') для большинства порожденных процессов, если их несколько.

Невозможно предсказать, какая команда будет иметь действующий входной поток (который будет идти с терминала или оттуда, откуда вы перенаправили стандартный вход программы make). Первая запущенная команда получит его первым, а первая команда, стартовавшая после того, как та финишировала, получит его следующей, и так далее.

Мы изменим этот аспект работы программы make, если найдем лучшую альтернативу. Между тем, вам ни в коем случае не следует полагаться на использование какой-либо командой стандартного входа, если вы используете возможность параллельного выполнения; если же вы не пользуетесь этой возможностью, стандартный вход нормально работает во всех командах.

Если выполнение команды не удается (она уничтожается с помощью сигнала или завершается с ненулевым результатом), и ошибки для этой команды не игнорируются (смотрите раздел 5.4 [Ошибки в командах]), оставшиеся командные строки для обновления той же самой цели не будут выполняться. Если выполнение команды не удается, и при этом не установлена опция '-k' или '--keep-going' (смотрите раздел 9.7 [Обзор опций]), make прерывает выполнение. Если make разрушается по любым причинам (включая сигнал) в тот момент, когда выполняются порожденные процессы, он ожидает их окончания, прежде чем на самом деле выйти.

Когда система сильно загружена, вы, вероятно захотите выполнять меньше заданий, чем тогда, когда она загружена слабо. Вы можете использовать опцию '-l', чтобы задать программе make ограничение на количество одновременно выполняемых заданий на основе средней загрузки. За опцией '-l' или '--max-load' следует число с плавающей точкой. Например, опция

```
-l 2.5
```

не позволит программе make запустить более одного задания, если средняя загрузка превышает 2.5. Опция '-l', за которой не следует число, отменяет ограничение на загрузку, если оно было установлено предыдущей опцией '-l'.

Более точно, когда make собирается запустить задание, и у него уже выполняется одно задание, он проверяет текущую среднюю загрузку - если она не меньше чем предел, определенный с помощью '-l', make ожидает до тех пор, пока средняя загрузка не опустится ниже предела, или пока не завершится другое задание.

По умолчанию предел загрузки не установлен.

5.4 Ошибки в командах

После завершения каждой команды командной оболочки make смотрит ее возвращаемый результат. Если программа успешно завершилась, выполняется следующая командная строка в новой командной оболочке; при завершении последней командной строки завершается правило.

Если встретилась ошибка (возвращаемый результат - ненулевой), make прекращает выполнение текущего правила и, возможно, всех правил.

Иногда неудачное выполнение определенной команды не является признаком проблемы. Например, вы можете использовать команду mkdir, чтобы убедиться, что каталог существует. Если каталог существует, mkdir сообщит об ошибке, но, тем не менее, вы, вероятно, захотите, чтобы make продолжил работу.

Чтобы игнорировать ошибки в командной строке, напишите символ '-' в начале ее текста (после начального символа табуляции). Этот символ отбрасывается, прежде чем команда передается для исполнения командной оболочке.

Например:

```
clean:

        -rm -f *.o
```

Это приводит к тому, что после rm выполнение команд будет продолжаться, даже если попытка удаления файлов окажется неудачной.

Когда вы запускаете make с опцией '-i' или '--ignore-errors', ошибки игнорируются во всех командах всех правил. Правило make-файла со специальной целью .IGNORE имеет такой же эффект, если у него нет зависимостей. Эти способы игнорирования ошибок являются устаревшими, поскольку использование символа '-' - более гибко.

Когда ошибка игнорируется из-за символа '-' или из-за флага '-i', make обрабатывает ошибочное завершение так же, как и успешное, за исключением того, что он печатает сообщение, которое указывает вам код результата, с которым завершилась команда, и говорит о том, что ошибка была игнорирована.

Когда встречается ошибка, про которую программе make не сказано, что ее надо игнорировать, подразумевается, что текущая цель не может быть корректно обновлена, также как и любая другая цель, которая прямо или косвенно зависит от нее. Для этих целей больше никаких команд исполняться не будет, так как их предусловия не были выполнены.

Обычно в такой ситуации make прекращает работу, возвращая ненулевой результат. Однако, если указана опция '-k' или '--keep-going', make, прежде, чем прекратить работу и возвратить ненулевой результат, продолжит обработку других зависимостей оставшихся целей, при необходимости обновляя их. Например, после ошибки при компиляции одного из объектных файлов, 'make -k' продолжит компиляцию других объектных файлов, хотя он знает, что их компоновка будет невозможна. Смотрите раздел 9.7 [Обзор опций].

Обычное поведение предполагает, что вашей задачей является получение обновленных целей; как только make выясняет, что это невозможно, он может непосредственно в этот же момент сообщить о неудаче. Опция '-k' указывает, что на самом деле задачей является тестирование максимально возможного количества изменений в программе, а также, возможно, выявление нескольких независимых проблем, с тем, чтобы вы могли могли скорректировать их всех перед следующей попыткой компиляции. Вот почему команда редактора Emacs 'compile' по умолчанию передает опцию '-k'.

Обычно в случае неудачного выполнения команды, если она вообще изменяла целевой файл, файл становится поврежденным и не может быть использован - или, по крайней мере, он не полностью обновлен. Кроме того, время последнего изменения файла говорит о том, что он в данный момент является обновленным, поэтому при следующем своем запуске make не попытается обновить этот файл. Ситуация точно такая же, как и при уничтожении команды с помощью сигнала - смотрите раздел 5.5 [Прерывания]. Таким образом, в общем случае то, что нужно сделать - это удалить целевой файл, если команда неудачно завершилась после начала изменения файла. make будет делать это, если в качестве цели указано .DELETE_ON_ERROR. Это почти всегда является тем, что вы хотите, чтобы делал make, но это не является исторически сложившейся практикой; поэтому для совместимости вы должны явно потребовать этого.

5.5 Прерывание или уничтожение программы make

Если make получает фатальный сигнал при выполнении команды, он может удалить целевой файл, который предполагалось обновить с ее использованием. Это делается в том случае, если время последней модификации целевого файла изменилось с тех пор, как make проверял его вначале.

Целевой файл удаляется для уверенности в том, что он будет корректно перегенерирован при следующем запуске программы make. Почему именно так ? Предположим, вы нажимаете Ctrl-c при выполнении компиляции в тот момент, когда началась запись в объектный файл 'foo.c'. Нажатие Ctrl-c уничтожает компилятор, и в результате получается неполный файл, время последней модификации которого более позднее, чем время последней модификации файла 'foo.c'. Но make также получает сигнал Ctrl-c и удаляет этот неполный файл. Если бы make не сделал этого, то при следующем его вызове считалось бы, что 'foo.o' не требует обновления, что привело бы к странному сообщению об ошибке от компоновщика, который попытался бы скомпоновать объектный файл, половина содержимого которого отсутствует.

Вы можете предотвратить удаление целевого файла в такой ситуации, указав его в качестве зависимости специальной цели .PRECIOUS. Перед обновлением цели make проверяет, находится ли она в числе зависимостей .PRECIOUS и на основании этого решает, должна ли удаляться цель при возникновении сигнала.

Некоторые причины, по которым вы могли бы сделать это : цель обновляется каким-либо элементарным способом, цель существует только для фиксации времени модификации или цель должна существовать все время, чтобы предотвратить неприятности другого рода.

5.6 Рекурсивное использование программы make

Рекурсивное использование программы make означает ее использование в качестве команды в make-файле. Это метод полезен тогда, когда вы хотите иметь отдельные make-файлы для различных подсистем, составляющих сложную систему. Например, предположим, что у вас есть подкаталог 'subdir', который имеет свой собственный make-файл, и вы хотели бы, чтобы make-файл объемлющего каталога запускал программу make для подкаталога. Вы можете сделать это, написав следующее:

```
subsystem:
        cd subdir; $(MAKE)
```

или, что эквивалентно, следующее (смотрите раздел 9.7 [Обзор опций]):

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

Вы можете писать рекурсивные команды make просто путем копирования этого примера, но есть много моментов, которые необходимо знать относительно того, как и почему они работают, а также относительно того, как взаимодействуют между собой порожденный процесс make (make нижнего уровня) и make вернего уровня.

Как работает переменная make

Рекурсивные команды make всегда должны использовать переменную MAKE, а не непосредственное имя команды 'make', как показано ниже:

```
subsystem:
        cd subdir; $(MAKE)
```

Значением этой переменной является имя файла, с помощью которого была вызвана программа make. Если этим именем файла было '/bin/make', то выпоняемая команда - 'cd subdir; /bin/make'. Если вы используете специальную версию программы make для обработки make-файла вернего уровня, то та же специальная версия будет исполняться при рекурсивных вызовах.

Как специальная возможность, использование переменной MAKE в командах правила изменяет действие опций '-t' ('--touch'), '-n'('--just-print') и '-q' ('--question'). Использование переменной MAKE имеет такой же эффект, как и использование символа '+' в начале командной строки (смотрите раздел 9.3 [Вместо исполнения команд]).

Рассмотрим команду 'make -t' в приведенном выше примере. (Опция '-t' помечает цели как обновленные без реального выполнения каких-либо команд - смотрите раздел 9.3 [Вместо исполнения команд]). Согласно обычному определению '-t', в

данном примере команда 'make -t' создала бы файл с именем 'subsystem' больше ничего не сделала бы. На самом деле, вам бы хотелось, чтобы запустилось 'cd subdir; make -t', но это потребовало бы выполнения команды, а опция '-t' указывает на то, что следует не выполнять команды.

Специальная обработка некоторых опций направлена на то, чтобы они выполняли то, чего вы от них хотите: всегда в тех случаях, когда командная строка правила содержит переменную MAKE, опции '-t', '-n' и '-q' не применяются к данной строке. Командные строки, содержащие MAKE, исполняются обычным образом, несмотря на наличие опции, приводящей к тому, что большинство команд не исполняются. Обычный механизм с использованием переменной MAKEFLAGS передает опции порожденному процессу make (смотрите раздел 5.6.3 [Опции для связи с порожденным процессом make]), поэтому ваш запрос на обновление файлов без исполнения соответствующих команд или на печать команд распространяется на подсистему.

Связь порожденным процессом make через переменные

Значения переменных процесса make верхнего уровня могут быть по явному требованию переданы порожденному процессу make через командную среду. Эти переменные по умолчанию определены в порожденном процессе make, но они не перекрывают то, что определено в make-файле, используемом порожденным процессом make, если вы не используете опцию '-e' (смотрите раздел 9.7 [Обзор опций]).

Чтобы передать, или, другими словами, экспортировать, переменную, make добавляет переменную и ее значение в командную среду для выполнения каждой команды. Порожденный процесс make, в свою очередь, использует командную среду для инициализации своей таблицы значений переменных. Смотрите раздел 6.9 [Переменные из командной среды].

Кроме явного требования, make экспортирует переменную в том случае, если она либо изначально определена в командной среде, или установлена в командной строке, и при этом ее имя состоит только из букв, цифр и символов подчеркивания. Некоторые командные оболочки не могут обрабатывать имена переменных окружения, включающих в себя символы, отличные от букв, цифр и символов подчеркивания.

Специальные переменные 'SHELL' и 'MAKEFLAGS' экспортируются всегда (если вы не отменяете их экспортирование). 'MAKEFILES' экспортируется, если вы устанавливаете во что-нибудь ее значение.

make автоматически передает значения переменных, которые были определены в командной строке путем добавления их к переменной MAKEFLAGS. Смотрите следующий раздел.

Обычно переменные не передаются, если они были по умолчанию созданы процессом make (смотрите раздел 10.3 [Переменные, используемые неявными правилами]). Порожденный процесс make сам определит их.

Если вы хотите экспортировать определенные переменные порожденному процессу make, используйте директиву make, как показано ниже:

```
export ПЕРЕМЕннАЯ ...
```

Если вы хотите отменить экспортирование переменной, используйте директиву unexport, как показано ниже:

```
unexport ПЕРЕМЕннАЯ ...
```

Для удобства вы можете определить переменную и одновременно экпортировать ее следующим способом:

```
export ПЕРЕМЕннАЯ = ЗнАЧЕнИЕ
```

что имеет такой же эффект, как и следующая запись:

```
ПЕРЕМЕннАЯ = ЗнАЧЕнИЕ
export ПЕРЕМЕннАЯ
```

а

```
export ПЕРЕМЕннАЯ := ЗнАЧЕнИЕ
```

имеет такой же эффект, как и следующая запись:

```
ПЕРЕМЕннАЯ := ЗнАЧЕнИЕ
export ПЕРЕМЕннАЯ
```

Аналогично,

```
export ПЕРЕМЕннАЯ += ЗнАЧЕнИЕ
```

является другой формой такой записи:

```
ПЕРЕМЕннАЯ += ЗнАЧЕнИЕ
export ПЕРЕМЕннАЯ
```

Смотрите раздел 6.6 [Добавление дополнительного фрагмента к переменным].

Вы можете заметить, что директивы export и unexport работают в программе make таким же образом, как они работают в командной оболочке, sh.

Если вы хотите, чтобы по умолчанию экспортировались все переменные, вы можете использовать директиву export без указания переменных:

```
export
```

Она указывает программе make на то, что переменные, которые не упоминаются явным образом в директивах export или unexport, должны быть экспортированы. Любая переменная, указанная в директиве unexport, не будет экспортирована. Если вы используете директиву export без указания переменных для того, чтобы по умолчанию переменные экспортировались, переменные, чьи имена содержат

символы, отличные от алфавитно-цифровых символов и символов подчеркивания, не будут экспортированы, если они не упоминаются в директиве export.

Поведение, определяемое директивой export без указания переменных было поведением по умолчанию в более старых GNU-версиях программы make. Если ваш make-файл построен в расчете на такое поведения и вы хотите, чтобы он был совместим с более старыми версиями программы make, вы можете написать правило со специальной целью .EXPORT_ALL_VARIABLES вместо использования директивы export. Это будет игнорировано старыми версиями программы make, в то время как директива export вызовет синтаксическую ошибку.

Аналогично, вы можете использовать unexport без указания переменных для того, чтобы указать программе make то, что по умолчанию переменные не должны экспортироваться. Поскольку такое поведение предусмотрено по умолчанию, вам потребуется делать это только в том случае, если ранее (возможно, во включаемом make-файле) была использована директива export без указания переменных. Вы не можете использовать директивы export и unexport без указания переменых для того, чтобы для одних команд переменные экспортировались, а для других - нет. Последняя директива без указания переменых export или unexport определяет поведение всего сеанса выполнения make.

В качестве дополнительной возможности, переменная MAKELEVEL изменяется при передаче ее с одного уровня на другой. Значением этой переменной является строка, представляющая глубину вложенности в виде десятичного числа. Для процесса make верхнего уровня ее значение - '0', для порожденного процесса make - '1', для процесса make, порожденного из порожденного процесса make - '2', и так далее. Увеличение значения происходит тогда, когда make устанавливает среду для команды.

Основное предназначение переменной MAKELEVEL - проверка ее значения в условной директиве (смотрите главу 7 [Условные части make-файла]); таким способом вы можете написать make-файл, который ведется себя по-разному, в зависимости от того, запущен ли он рекурсивно или же - напрямую вами.

Вы можете использовать переменную MAKEFILES для того, чтобы заставить все команды запуска порожденных процессов make использовать дополнительные make-файлы. Значением переменной MAKEFILES является разделенный пробелами список имен файлов. Эта переменная, будучи определенной на самом внешнем уровне, передается через командную среду; затем она служит в качестве списка дополнительных make-файлов, читаемых порожденным процессом make перед чтением обычных или определенных при помощи параметров make-файлов. Смотрите раздел 3.4 [Переменная MAKEFILES].

Опции для связи с порожденным процессом make

Такие флаги, как '-s' и '-k' автоматически передаются порожденному процессу make через переменную MAKEFLAGS. Эта переменная, автоматически устанавливаемая программой make, содержит буквы тех опций, которые получает данный экземпляр make. Таким образом, если вы запускаете 'make -ks', то MAKEFLAGS получает значение 'ks'.

Следовательно, каждый порожденный процесс make получает в своей командной среде значение MAKEFLAGS. В ответ на это он берет опции из этого значения и

обрабатывает их так, как будто они были переданы в качестве аргументов. Смотрите раздел 9.7 [Обзор опций].

Таким же образом переменные, определенные в командной строке, передаются порожденному процессу make через MAKEFLAGS. Те слова из значения MAKEFLAGS, которые содержат символ '=', make обрабатывает как определения переменных, как будто они появились в командной строке. Смотрите раздел 9.5 [Перекрывающиеся переменные].

Опции '-C', '-f', '-o' и '-W' не указываются в MAKEFLAGS - эти опции не передаются порожденному процессу make.

Опция '-j' представляет собой особый случай (смотрите раздел 5.3 [Параллельное выполнение]). Если вы устанавливаете ее в численное значение, то в MAKEFLAGS всегда подставляется '-j 1' вместо определенного вами значения. Это из-за того, что при передаче опции '-j' порожденным процессам make вы бы получили гораздо больше параллельно исполняющихся заданий, чем запрашивали. Если вы указываете '-j' без числового аргумента, что означает параллельное исполнение максимально возможного количества заданий, то такая опция передается без изменений, так как несколько бесконечностей в сумме дают одну бесконечность.

Если вы нехотите передавать порожденному процессу make другие опции, вам следует изменить значение MAKEFLAGS, как показано ниже:

```
MAKEFLAGS=
subsystem:
        cd subdir; $(MAKE)
```

или так:

```
subsystem:
        cd subdir; $(MAKE) MAKEFLAGS=
```

На самом деле определения переменных командной строки появляются в переменной MAKEOVERRIDES, а MAKEFLAGS содержит ссылку на эту переменную. Если вы хотите обычным образом передать порожденным процессам make опции, но не хотите передавать им определения переменных командной строки, вы можете переустановить в пустое значение MAKEOVERRIDES, как показано ниже:

```
MAKEOVERRIDES=
```

Это не является типичным полезным действием. Однако, некоторые системы имеют сильное фиксированное ограничение на размер командной среды, и помещение такого большого количества информации в значение переменой MAKEFLAGS может превысить его. Если вы видите сообщение об ошибке 'Arg list too long', именно это может быть причиной. (Для строгой совместимости с POSIX.2, изменение MAKEOVERRIDES не влияет на MAKEFLAGS, если в make-файле появляется специальная цель '.POSIX'. Вы, вероятно, на это не обращаете внимание.)

В целях исторической совместимости существует также похожая переменная MFLAGS. Она имеет такое же значение, как и MAKEFLAGS, за исключением того,

что она не содержит определения переменных командной строки, и она всегда, когда непустая, начинается с символа '-' (MAKEFLAGS начинается с символа '-' только тогда, когда она начинается с опции, не имеющей однобуквенной версии, например '--warn-undefined-variables'). MFLAGS традиционно использовалась явным образом в рекурсивной команде make, как показано ниже:

```
subsystem:
        cd subdir; $(MAKE) $(MFLAGS)
```

но сейчас MAKEFLAGS делает такое использование излишним. Если вы хотите, чтобы ваши make-файлы были совместимыми со старыми make-программами, используйте этот метод - он будет также прекрасно работать с более новыми версиями программмы make.

Переменная MAKEFLAGS также может быть полезной, если вы хотите иметь определенные опции, такие как '-k' (смотрите раздел 9.7 [Обзор опций]), установленными каждый раз, когда вы запускаете make. Вы просто определяете значение переменной MAKEFLAGS в вашей командной среде. Вы также можете установить в MAKEFLAGS в make-файле для того, чтобы определить дополнительные опции, которые также должны иметь силу для соответствующего make-файла. (Обратите внимание, что вы не можете таким способом использовать MFLAGS. Эта переменная установлена только для совместимости - make не интерпретирует значение, в которое вы ее каким-либо способом устанавливаете.)

Когда программа make интерпретирует значение переменной MAKEFLAGS (либо из командной среды, либо из make-файла), она в первую очередь подставляет в его начало символ '-', если значение переменной не начинается уже с него. Затем make разрубает значение на слова, разделенные пробелами, и обрабатывает эти слова так, как будто они являются опциями, передаваемыми через командную строку (за исключением того, что опции '-C', '-f', '-h', '-o', '-W' и их версии с длинными именами игнорируются, а также не фиксируется ошибок для некорректных опций).

Если вы устанавливаете MAKEFLAGS в вашей командной среде, вам следует убедиться в том, что вы не включили какие-либо опции, которые серьезно повлияют на действия программы make и изменят предназначение make-файлов и самой программы make. Например, если бы в этой переменной была указана одна из опций '-t', '-n', или '-q', это могло бы вызвать разрушительные последствия, и, конечно, имело бы, по меньшей мере, удивительные, и, возможно, надоедающие эффекты.

Опция '--print-directory'

Если вы используете несколько уровней рекурсивных вызовов программы make, опция '-w' или '--print-directory' может сделать выход программы намного более легким для понимания посредством показа каждого каталога в момент начала и окончания его обработки. Например, если 'make -w' выполняется в каталоге '/u/gnu/make', то make напечатает строку следующей формы:

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

прежде, чем что-либо сделать, и строку следующей формы:

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

когда обработка завершена.

Обычно вам не требуется определять эту опцию, поскольку тогда, когда вы пишете 'make', это делается за вас: '-w' автоматически включается, когда вы используете опцию '-C' и в порожденных процессах make. Программа make не будет автоматически включать опцию '-w', если вы при этом используете '-s', которая подавляет вывод, или '--no-print-directory', для явного ее выключения.

5.7 Определение именованных командных последовательностей

Когда одна и та же последовательность команд используется при порождении различных целей, вы можете определить ее как именованную последовательность при помощи директивы define и обращаться к именованной последовательности из правил для этих целей. Именованная последовательность на самом деле является переменной, поэтому ее имя не должно конфликтовать с другими именами переменных.

Вот пример определения именованной последовательности команд:

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c$ $@
endef
```

Здесь run-yacc является именем определяемой переменной, endef обозначает конец определения, строки между именем и концом определения представляют собой команды. Директива define не заменяет ссылки на переменные и вызовы функции в именованной последовательности - символы '$', скобки, имена переменных и т.п., все они становятся частью переменной, которую вы определяете. Смотрите раздел 6.8 [Определение многостроковых переменных] для полной информации о директиве define.

Первая команда в этом примере запускает Yacc для первой зависимости любого правила, использующего эту именованную последовательность. Выходной файл программы Yacc всегда называется 'y.tab.c'. Вторая команда переименовывает выходной файл в имя целевого файла правила.

Чтобы использовать именованную последовательность, подставьте переменную в команды правила. Вы можете подставить ее так же, как и любую другую переменную (смотрите раздел 6.1 [Основы обращения к переменным]). Поскольку переменные, определенные с помощью директивы define, являются рекурсивно подставляемыми переменными, все обращения к переменным, которые вы написали внутри конструкции define, в этот момент заменяются на их значения. Например, в правиле

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

'foo.y' будет подставлено вместо переменной '$^' в том месте, где она встречается в значении переменной run-yacc, а 'foo.c' - вместо '$@ '.

Это реалистичный пример, однако именно он не требуется на практике, поскольку make имеет неявное правило, действующее для файлов с указанными именами, для выполнения этих команд (смотрите главу 10 [Использование неявных правил]).

При выполении команд каждая строка именованной последовательности обрабатывается точно так же, если бы она сама появилась в правиле, с предшествующим символом табуляции. В частности, make вызывает командные подоболочки для каждой строки. Вы можете использовать специальные префиксные символы, которые воздействуют на командные строки ('@ ', '-' и '+') в каждой строке именованной последовательности. Смотрите главу 5 [Написание команд в правилах]. Например, при использовании такой именованной последовательности:

```
        define frobnicate
        <htmlurl name="@echo" url="mailto:@echo">
"frobnicating target $@
        frob-step-1 $< -o $@      -step-1
        frob-step-2 $@      -step-1 -o $@
        endef
```

программа make не будет отображать первую строку, команду echo. Но следующие две командные строки будут отображены.

С другой стороны, префиксные символы в командной строке, относящиеся к именованной последовательности, применяются к каждой строке последовательности. Таким образом, правило:

```
        frob.out: frob.in
        @       $(frobnicate)
```

не отображает ни одну команду. (Смотрите раздел 5.1 [Отображение команды] для полного объяснения символа '@ '.)

5.8 Использование пустых команд

Иногда полезно определять команды, которые ничего не делают. Это делается путем использования команды, не включающей в себя ничего, кроме пробела. Например, правило

```
        target: ;
```

определяет пустую командную строку для target. Вы могли бы также использовать текстовую строку, начинающуюся с символа табуляции, для определения пустой

командной строки, но это приводило бы к путанице, поскольку такая текстовая строка выглядела бы пустой.

Вы можете удивиться, почему бы вы могли захотеть определить командную строку, которая ничего не делает. Единственная причина заключается в том, что это полезно для защиты цели от использования неявных команд (из неявных правил или специальной цели .DEFAULT - смотрите главу 10 [Неявные правила] и раздел 10.6 [Определение правил последней возможности, используемых по умолчанию]).

Вы могли склониться к определению пустых командных строк для для целей, которые не являются настоящими файлами, а существуют только для того, чтобы их зависимости могли быть перегенерированы. Однако, это не лучший способ действия в такой ситуации, поскольку зависимости могут не быть должным образом перегенерированы, если целевой файл на самом деле существует. Смотрите раздел 4.4 [Цели-имена действий], где описывается более подходящий для этого способ.

6. Как использовать переменные

Переменная представляет собой имя, определенное в make-файле для представления текстовой строки, называемой значением переменной. Такое значение подставляется, при явном указании на это, в цели, зависимости, команды и другие части make-файла. (В некоторых других версиях программы make переменные называются макросами.)

Переменные и функции во всех частях make-файла вычисляются при их чтении, за исключением команд командной оболочки в правилах, правой части определения переменной с использованием символа '=' и тел определений переменных с использованием директивы define.

Переменные могут представлять списки имен файлов, опции, передаваемые компилятору, запускаемые программы, каталоги для поиска исходных файлов, каталоги для записи выхода, или все остальное, что вы можете представить.

Именем переменной может быть любая последовательность символов, не содержащая символов ':', '#', '=' , а также начальных или конечных пробелов. Однако, следует избегать имен переменных, содержащих символы, отличные от букв, цифр и символов подчеркивания, поскольку таким символам в будущем может быть назначено специальное значение, а в некоторых командных оболочках их нельзя будет передать через командную среду порожденному процессу make (смотрите раздел 5.6.2 [Связь порожденным процессом make через переменные]).

Имена переменных чувствительны к регистру. Каждое из имен 'foo', 'FOO' и 'Foo' ссылается на отдельную переменную.

Традиционным является использование в именах переменных больших букв, но мы рекомендуем использовать маленькие буквы для имен переменных, служащих в make-файла для внутренних нужд, и резервировать верхний регистр для параметров, управляющих неявными правилами, и для параметров, которые предназначены для переопределения пользователем при помощи опции командной строки (смотрите раздел 9.5 [Перекрывающиеся переменные]).

Несколько переменных имеют имена, представляющие собой одиночный символ пунктуации или несколько таких символов. Это автоматические переменные, и они имеют отдельно оговоренное использование. Смотрите раздел 10.5.3 [Автоматические переменные].

6.1 Основы обращения к переменным

Для подстановки значения переменной напишите знак доллара с последующим именем переменной в круглых или фигурных скобках: как '$(foo)', так и '${foo}' являются правильными ссылками на переменную foo. Это специальное значение символа '$' является причиной того, что вы должны писать '$$' для обеспечения эффекта появления одного знака доллара в имени файла или команде.

Ссылки на переменные могут быть использованы в любом контексте: в целях, в зависимостях, в командах, в большинстве директив и в значениях новых переменных. Вот типичный пример, в котором переменная содержит имена всех объектных файлов программы:

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

$(objects) : defs.h
```

Ссылки на переменные обрабатываются при помощи строгой текстуальной подстановки. Таким образом, правило

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

могло бы быть использовано для компиляции C-программы 'prog.c'. Так как при присвоении переменной значения пробелы, предшествующие ему, игнорируются, значением переменной foo является именно 'c'. (На самом деле вам не рекомендуется писать make-файлы таким образом !)

Если за знаком доллара следует символ, отличный от знака доллара или открывающейся круглой или квадратной скобки, то этот символ обрабатывается как имя переменной. Таким образом, вы могли бы обратиться к переменной x при помощи '$x'. Однако, такая практика крайне нежелательна, за исключением случая автоматических переменных (смотрите раздел 10.5.3 [Автоматические переменные]).

6.2 Две разновидности переменных

Есть два способа, с помощью которых переменная в GNU-версии программы make может получить значение - мы будем называть их двумя разновидностями переменных. Две разновидности различаются тем, как они определяются и что с ними просиходит при их вычислении.

Первая разновидность переменной - это рекурсивно вычисляемая переменная. Переменные такого рода определяются в пределах одной строки make-файла с использованием символа '=' (смотрите раздел 6.5 [Установка переменных]), или при помощи директивы define (смотрите раздел 6.8 [Определение многостроковых переменных]). Определяемое вами значение устанавливается неявным образом - если переменная содержит ссылки на другие переменные, они заменяются на значения всегда, когда происходит подстановка переменной (во время вычисления какой-либо другой строки). Когда такое происходит, это называется рекурсивным вычислением.

Например, фрагмент make-файла

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

all:;echo $(foo)
```

отобразит на экране 'Huh?': ссылка на переменную '$(foo)' заменяется на ссылку на переменную '$(bar)', которая заменяется на ссылку на переменную '$(ugh)', которая, наконец, заменяется на 'Huh?'.

Только эта разновидность переменных поддерживается другими версиями программы make. Она имеет свои достоинства и недостатки. Преимущество (по мнению большинства) заключается в том, что следующий фрагмент:

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

сделает то, что предполагается: когда в команде происходит 'CFLAGS' вычисление, результатом вычисления будет '-Ifoo -Ibar -O'. Основным недостатком является то, что вы не можете ничего добавить в конец переменной, как показано ниже:

```
CFLAGS = $(CFLAGS) -O
```

поскольку это вызовет бесконечный цикл при вычислении переменной. (На самом деле, программа make обнаруживает бесконечный цикл и сообщает об ошибке.)

Еще один недостаток состоит в том, что любая функция (смотрите главу 8 [Функции преобразования текста]), упомянутая в определении, будет выполняться каждый раз при вычислении переменной. Это замедляет выполнение программы make; хуже того, это приводит к тому, что функции wildcard и shell дают непредсказуемые результаты, так как вы не можете легко проконтролировать, когда, и даже сколько раз, они вызываются.

Для того, чтобы избавиться от всех проблем и неудобств рекурсивно вычисляемых переменных, есть другая разновидность: упрощенно вычисляемые переменные.

Упрощенно вычисляемые переменные определяются в пределах одной строки make-файла с использованием ':=' (смотрите раздел 6.5 [Установка переменных]). Значение упрощенно вычисляемой переменной просматривается один раз за все время работы с ней, при этом в ее определении происходит замена всех ссылок на

другие переменные и функции на их значения. В действительности, значением упрощенно вычисляемой переменной является результат вычисления написанного вами текста. Оно не содержит никаких ссылок на другие переменные - она содержит их значения на тот момент, когда она определялась. Следовательно, данный фрагмент:

```
x := foo
y := $(x) bar
x := later
```

эквивалентнен следующему:

```
y := foo bar
x := later
```

При ссылке на упрощенно вычисляемую переменную, происходит просто подстановка ее значения.

Вот несколько более сложный пример, иллюстрирующий использование ':=' вместе с функцией shell. (смотрите раздел 8.6 [Функция shell]). Этот пример также показывает использование переменной MAKELEVEL, которая изменяется при ее передаче с одного уровня на другой. (Смотрите раздел 5.6.2 [Связь порожденным процессом make через переменные] для информации о переменной MAKELEVEL.)

```
ifeq (0,${MAKELEVEL})
cur-dir   := $(shell pwd)
whoami    := $(shell whoami)
host-type := $(shell arch)
MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}
endif
```

Преимущество такого использования ':=' состоит в том, что типичная команда 'спуска в каталог' выглядит в данном случае так:

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

Упрощенно вычисляемые переменные, вообще говоря, делают программирование сложных make-файлов более предсказуемым, поскольку они работают, как переменные в большинстве языков программирования. Они позволяют вам переопределять переменную, используя ее собственное значение (или ее значение, обработанное некоторым образом одной из преобразующей функций) и намного более эффективно использовать преобразующие функции (смотрите главу 8 [Функции преобразования текста]).

Вы также можете использовать их для внесения в значения переменных управляемого ведущего пробела. Ведущие пробельные символы удаляются из вашего указанного вами значения перед подстановкой значений вместо ссылок на переменные и вызовов функций; это означает, что вы можете включить ведущие

пробелы в значение переменной посредством защиты их с помощью ссылок на переменные, как показано ниже:

```
nullstring :=
space := $(nullstring) # конец строки
```

Здесь значением переменной space является ровно один пробел. Комментарий '# конец строки' вставлен сюда только для ясности. Так как ведомые пробельные символы не удаляются из значений переменых, просто пробел в конце строки имел бы такой же эффект (но это было бы довольно сложно читать). Если вы добавляете пробел в конце значениея переменной, неплохо бы добавить в конце строки подобный комментарий, чтобы сделать ваш замысел ясным. Напротив, если вы не хотите никаких пробельных символов в конце значения вашей переменной, вы должны запомнить, что не следует добавлять случайный комментарий в конце строки после нескольких пробелов, как показано ниже:

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

Здесь значением переменной dir является '/foo/bar ' (с четырьмя ведомыми пробелами), что, вероятно, не является замыслом. (Представьте что-либо типа '$ (dir)/file' с таким определением !)

6.3 Дополнительные возможности для ссылки на переменные

Этот раздел описывает некоторые дополнительные возможности, которые вы можете использовать для обращения с переменными более гибкими способами.

Ссылки с заменой

На место ссылки на замену подставляется значение переменной с определяемыми вами изменения. Она имеет форму '$(var:a=b)' (или '${var:a=b}') и означает, что в значении переменной var, вхождение 'a' в конце каждого слова в этом значении заменяется на 'b', а получившаяся строка будет подставлена на место ссылки.

Когда мы говорим "в конце каждого слова", мы имеем ввиду, что либо за вхождением 'a' должен следовать пробел, либо оно должно находиться в конце значения для того, чтобы быть замененным; другие вхождения 'a' в значение остаются неизмененными. Например, фрагмент make-файла

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

устанавливает переменную 'bar' в 'a.c b.c c.c'. Смотрите раздел 6.5 [Установка переменных].

На самом деле, ссылка с заменой является сокращением использования преобразующей функции patsubst (смотрите раздел 8.2 [Функции подстановки и анализа строк]). Для совместимости с лругими реализациями программы make, данная версия поддерживает ссылку с заменой, также как и patsubst.

Еще один вид ссылки с заменой позволяет вам использовать всю мощь функции patsubst. Он имеет такую же форму '$(var:a=b)', описанную выше, за исключением того, что теперь 'a' должна содержать один, и только один, символ '%'. Этот случай эквивалентен вызову '$(patsubst A,B,$(VAR))'. Смотрите раздел 8.2 [Функции подстановки и анализа строк] для описания функции patsubst.

Например, фрагмент make-файла

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

устанавливает переменную 'bar' в 'a.c b.c c.c'.

Вычисляемые имена переменных

Вычисляемые имена переменных - это сложное понятие, необходимое только для утонченного программирования make-файла. В большинстве случаев вам нет необходимости изучать их, надо только знать, что создание переменной со знаком доллара в ее имени может привести к странным результатам. Однако, если вы человек того типа, который хочет все понять или вы на самом деле интересуетесь тем, что представляют собой такие переменные, читайте дальше.

На переменные можно ссылаться в имени переменной. Это называется вычисляемым именем переменной или вложенной ссылкой на переменную. Например, фрагмент make-файла

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

определяет 'z' в качестве значения переменной a: ссылка '$(x)' внутри '$($(x))' заменяется на 'y', поэтому '$($(x))' заменяется на ссылку '$(y)', которая, в свою очередь, заменяется на 'z'. Здесь имя ссылающейся переменной не указано явным образом - оно вычисляется путем замены '$(x)'. В данном случае ссылка '$(x)' вложена в более внешнюю ссылку на переменную.

Предыдущий пример показывает два уровня вложенности, но возможно любое количество уровней. Например, здесь три уровня вложенности:

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

В этом примере самая внутренняя ссылка '$(x)' заменяется expands to 'y', поэтому '$($(x))' заменяется на ссылку '$(y)', которая, в свою очередь, заменяется на 'z'; теперь мы имеет ссылку '$(z)', которая превращается в 'u'.

Ссылки на рекурсивно вычисляемые переменные внутри имени переменной перевычисляются обычным образом. Например, данный фрагмент make-файла:

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

определяет 'Hello' в качестве значения переменной a: ссылка '$($(x))' превращается в ссылку '$($(y))', которая превращается в ссылку '$(z)', которая превращается в 'Hello'.

Вложенные ссылки на переменные могут также содержать модифицированные ссылки и вызовы функций (смотрите главу 8 [Функции преобразования текста]), так же, как и любые другие ссылки. Например, использование функции subst (смотрите раздел 8.2 [Функции подстановки и анализа строк]), как в данном примере:

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

определяет 'Hello' в качестве значения переменной a. Сомнительно, что кто-нибудь когда-либо захочет написать вложенную ссылку, запутанную так же, как и эта, но она работает: ссылка '$($($(z)))' заменяется на ссылку '$($(y))', которая превращается в '$($(subst1,2,$(x)))'. Здесь из переменной x берется значение 'variable1', которое заменяется путем подстановки на 'variable2', таким образом строка целиком принимает вид '$(variable2)', что является простой ссылкой на переменную , значением которой является 'Hello'.

Вычисляемому имени переменной не обязательно состоять целиком из одной ссылки на переменную. Оно может содержать несколько ссылок на переменные, а также какой-нибудь неизменяемый текст. Например, в данном фрагменте make-файла:

```
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))
```

переменной dirs будет дано, такое же значение, как переменной 'a_dirs', '1_dirs', 'a_files' или '1_files', в зависимости от установленных значений переменных 'use_a' и 'use_dirs'.

Вычисляемые имена переменных также могут быть использованы в ссылках с заменой. Например, в данном фрагменте make-файла

```
a_objects := a.o b.o c.o
1_objects := 1.o 2.o 3.o

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

в качестве значения переменной sources определяется либо 'a.c b.c c.c', либо '1.c 2.c 3.c', в зависимости от значения переменной a1.

Единственное ограничение на такого рода использование вложенных ссылок на переменные состоит в том, что они не могут определять часть имени вызываемой функции. Это из-за того, что проверка на корректность имени функции производится до вычисления вложенных ссылок. Например, в данном фрагменте make-файла:

```
ifdef do_sort
func := sort
else
func := strip
endif

bar := a d b g q c

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

принимается попытка присвоить переменной 'foo' либо значение 'sort a d b g q c', либо значение 'strip a d b g q c', а не передавать 'a d b g q c' в качестве аргумента либо функции sort, либо функции strip. Это ограничение в будущем может быть снято, если такое изменение покажется хорошей идеей.

Вы также можете использовать вычисляемые имена переменных в левой части присваивания значения переменной или в директиве define, как показано ниже:

```
dir = foo
$(dir)_sources := $(wildcard $(dir)/*.c)
define $(dir)_print
lpr $($(dir)_sources)
endef
```

В этом примере определяются переменные 'dir', 'foo_sources' и 'foo_print'.

Обратите внимание, что вложенные ссылки на переменные полностью отличаются от рекурсивно вычисляемых переменных (смотрите раздел 6.2 [Две разновидности переменных]), хотя и те, и другие сложными способами используются вместе при программировании make-файла.

6.4 Как переменные получают свои значения

Переменные могут получать значения несколькими различными способами:

- Вы можете определить перекрывающее значение при запуске программы make. Смотрите раздел 9.5 "Перекрывающиеся переменные".
- Вы можете определить значение в make-файле, либо при помощи присваиванием (смотрите раздел 6.5 "Установка переменных"), либо при помощи определения многостроковой переменной (смотрите раздел 6.8 "Определение многостроковых переменных").
- Переменные из командной среды становятся переменными программы make. смотрите раздел 6.9 "Переменные из командной среды".
- Некоторым автоматическим переменным в каждом правиле присваивается новое значение. Смотрите раздел 10.5.3 "Автоматические переменные".
- Некоторые переменные имеют постоянные первоначальные значения. Смотрите раздел 10.3 "Переменные, используемые неявными правилами".

6.5 Установка переменных

Чтобы установить переменную из make-файла, напишите строку, начинающуюся с имени переменной, за которым следует '=' или ':='. Все, что в данной строке следует за '=' или ':=', становится значением. Например, в данной строке:

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

определяется переменная с именем objects. Пробелы вокруг имени переменной и непосредственно после символа '=' игнорируются.

Переменные, определенные при помощи символа '=', являются рекурсивно вычисляемыми переменными. Переменные, определенные с помощью ':=', являются упрощенно вычисляемыми переменными - эти определения могут содержать ссылки на переменные, которые будут вычислены, прежде чем будет сделано определение. Смотрите раздел 6.2 [Две разновидности переменных].

Имя переменной может содержать ссылки на функции и переменные, которые вычисляются для определения того, какое имя на самом деле надо использовать, в тот момент, когда считывается строка make-файла.

На длину значения переменной нет ограничений, за исключением количества пространства для подкачки на компьютере. Когда определение переменной длинное, неплохо бы разбить ее на несколько строк, вставляя в определении символ '\' с последующим переводом строки там, где это удобно. Это не повлияет на работу программы make, но сделает make-файл более легким для чтения.

Для большинства имен переменных считается, что соответствующая переменная имеет в качестве значения пустую строку, если вы нигде ее не устанавливали. Несколько переменных имеют встроенные первоначальные значения, которые не являются пустыми, но вы можете установить их обычными способами (смотрите раздел 10.3 [Переменные, используемые неявными правилами]). Некоторые специальные переменные в каждом правиле автоматически устанавливаются в новое значение - они называются автоматическими переменными (смотрите раздел 10.5.3 [Автоматические переменные]).

6.6 Добавление дополнительного фрагмента к переменным

Часто полезно добавлять дополнительный фрагмент к значению уже определенной переменной. Это делается при помощи строки, содержащей '+=', как показано ниже:

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

В этом примере берется значение переменной objects, и к нему добавляется фрагмент 'another.o' (с предшествующим пробелом). Таким образом, данный фрагмент make-файла

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

устанавливает 'main.o foo.o bar.o utils.o another.o' в качестве значения переменной objects.

Использование '+=' аналогично следующему:

```
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o
```

но отличается теми деталями, которые становятся важными при использовании более сложных значений.

Когда расматриваемая переменная ранее не была определена, '+=' действует так же, как обычный символ '=': определяется рекурсивно вычисляемую переменную. Однако, если есть предыдущее определение, тогда то, что именно делает '+=' зависит от того, какая разновидность переменной определена первоначально. Смотрите раздел 6.2 [Две разновидности переменных] для объяснения двух разновидностей переменных.

Когда вы что-либо добавляете к значению переменной при помощи '+=', программа make, по существу, действует так, как будто вы включили дополнительный текст в первоначальное определение переменной. Если вы изначально определили переменную при помощи ':=', сделав ее упрощенно вычисляемой переменной, '+=' производит добавление к этому упрощенно-вычисленному определению, и вычисляет новый фрагмент перед добавлением его к старому значению, так же, как это делается при использовании ':=' (смотрите раздел 6.5 [Установка переменных] для полного объяснения ':='). Фактически, следуюзий фрагмент make-файла:

```
variable := value
variable += more
```

в точности эквивалентен такому фрагменту:

```
variable := value
variable := $(variable) more
```

С другой стороны, когда вы используете '+=' с переменной, которую вы при помощи одиночного символа '=' изначально определили как рекурсивно-вычисляемую, программа make некоторые вещи делает немного иначе. Вспомните, что когда вы определяете рекурсивно-вычисляемую переменную, программа make не вычисляет сразу значение установленных вами ссылок на переменные и функции. Вместо этого, она она запоминает фиксированные фрагменты и хранит эти ссылки на переменные и функции с тем, чтобы вычислить их позже, когда вы обратитесь к новой пременной (смотрите раздел 6.2 [Две разновидности переменных]). Когда вы используете '+=' с рекурсивно-вычисляемой переменной, имеется невычисленный фрагмент, к которому make добавляет определенный вами новый фрагмент.

```
variable = value
variable += more
```

Приведенные выше строки make-файла, грубо говоря, эквивалентны следующим строкам:

```
temp = value
variable = $(temp) more
```

за исключением того, что, конечно, никогда не определяется переменная с именем temp. Важность использования '+=' проявляется в той ситуации, когда старое значение переменной содержит ссылки на переменные. Рассморим такой типичный пример:

```
CFLAGS = $(includes) -O
...
CFLAGS += -pg # enable profiling
```

Первая строка определяет переменную CFLAGS со ссылкой на другую переменную, includes. (CFLAGS используется правилами для C-компиляции, смотрите раздел 10.2 [Перечень неявных правил].) Использование для определения '=' делает CFLAGS рекурсивно-вычисляемой переменной, а это означает, что '$(includes) -O' не вычисляется в тот момент, когда программа make обрабатывает значение определение переменной CFLAGS. Таким образом, чтобы получить эффект от использования переменной includes, не обязательно наличие у нее определенного значения на момент определения CFLAGS. Достаточно того, чтобы переменная includes была определена перед любой ссылкой на CFLAGS. Если бы мы попробовали добавить значение к CFLAGS без использования '+=', мы могли бы сделать это примерно так:

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

Это близко к истине, но не совсем то, чего мы хотим. При использовании ':=' CFLAGS переопределяется как упрощенно-вычисляемая переменная - это означает, что программа make вычисляет фрагмент '$(CFLAGS) -pg' перед установкой значения переменной. Если переменная includes еще неопределена, мы получим ' -O -pg', и более позднее определение переменной includes не окажет никакого воздействия. Напротив, при использовании '+=' мы устанавливаем переменную CFLAFGS в невычесленное значение '$(includes) -O -pg'. Таким образом, мы сохраняем ссылку на переменную includes, поэтому если эта переменная оказывается определенной где-нибудь дальше, ссылка вида '$ (CFLAGS)' все-таки будет использовать ее значение.

6.7 Директива override

Если переменная была установлена при помощи аргумента командной строки (смотрите раздел 9.5 [Перекрывающиеся переменные]), то обычные присваивания в make-файле игнорируются. Если вы хотите вы хотите установить переменную в make-файле, даже если она установлена при помощи аргумента командной строки, вы можете использовать директиву override, которая представляет собой строку make-файла, выглядящую следующим образом:

```
override VARIABLE = VALUE
```

или так:

```
override VARIABLE := VALUE
```

Для добавления дополнительного фрагмента к переменной, определенной в командной строке, используйте такую строку make-файла:

```
override VARIABLE += MORE TEXT
```

Смотрите раздел 6.6 [Добавление дополнительного фрагмента к переменным].

Директива override была введена не для усиления войны между make-файлами и аргументами командной строки. Она была введена для того, чтобы вы могли изменять и дополнять значения, которые пользователь определяет при помощи аргументов командной строки.

Например, предположим, что вы всегда хотите, чтобы при запуске C-компилятора использовалась опция '-g', но вы хотели бы позволить позволить пользователю определять другие опции, как обычно, в аргументе командной строки.

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

Вы можете также использовать директивы override с директивами define. Это делается так же, как вы и могли ожидать:

```
override define foo
bar
endef
```

Смотрите следующий раздел для информации о директиве define.

6.8 Определение многостроковых переменных

Еще один способ устанвки значения переменной - использовать директиву define. Эта директива имеет необычный синтаксис, который позволяет включать в значение символы перевода строки, что удобно для определения именованных последовательностей команд (смотрите раздел 5.7 [Определение именованных командных последовательностей]).

За директивой define на той же строке следует имя переменной и ничего больше. Значение, в которое устанавливается переменная, появляется на последующих строках. Конец значения обозначается строкой, содержащей только одно слово endef. За исключением этого отличия в синтаксисе, директива define работает точно так же, как и '=': она создает рекурсивно вычисляемую переменную (смотрите раздел 6.2 [Две разновидности переменных]). Имя переменной может содержать ссылки на функции и переменные, которые вычисляются в момент чтения директивы для определения того, какое реальное имя переменной следует использовать.

```
define two-lines
echo foo
echo $(bar)
endef
```

При обычном присваивании значение не может содержать символов перевода строки, в то же время символы перевода строки, которые разделяют строки в значении, определяемом директивой define, становятся частью значения переменной (за исключением последнего перевода строки, который предшествует директиве endef и не рассматривается как часть значения).

Предыдущий пример функционально эквивалентен приведенному ниже:

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

так как две команды, разделенные точкой с запятой, ведут себя во многом так же, как две отдельные команды командной оболочки. Однако, обратите внимание, что использование двух отдельных строк означает, что программа make будет вызывать командную оболочку дважды, запуская независимые подоболочки для каждой строки. Смотрите раздел 5.2 [Выполнение команд].

Если вы хотите, чтобы определения переменных, сделанные при помощи define, имели преимущество перед определениями переменных из командной строки, вы можете вместе с директивой define использовать директиву override, как показано ниже:

```
override define two-lines
foo
$(bar)
endef
```

Смотрите раздел 6.7 [Директива override].

6.9 Переменные из командной среды

Переменные могут приходить в программу make из командной среды, в которой make запускается. Каждая переменная командной среды, которая доступна программе make при старте, преобразуется в переменную программы make с таким же именем и значением. Но явное присваивание в make-файле или при помощи арргумента командной строки перекрывает значение из командной среды. (Если определена опция '-e', значение из командной среды перекрывает присваивания в make-файле. Смотрите раздел 9.7 [Обзор опций]. Но такая практика не рекомендуется.)

Таким образом, установив переменную CFLAGS в вашей командной среде, вы можете сделать так, чтобы во всех сеансах компиляции C-программ в большинстве make-файлов использовались выбранные вами опции компилятора. Это безопасно для переменных со стандартными или обговоренными предназначениями, поскольку вы знаете, что ни один make-файл не будет использовать их ни для чего другого. (Но это не является абсолютно надежным - некоторые make-файлы явным образом устанавливают переменную CFLAGS и, следовательно, на них не действует значение из командной среды).

Когда программа make вызывается рекурсивно, переменные, определенные в более внешних порожденных процессах make, могут быть переданы более внутренним порожденных процессах make через командную среду (смотрите раздел 5.6 [Рекурсивное использование make]). По умолчанию, при рекурсивных вызовах передаются только переменные, которые пришли из командной среды или определены в командной строке. Для того, чтобы передать другие переменные, вы можете использовать директиву export. Смотрите раздел 5.6.2 [Связь порожденным процессом make через переменные], для выяснения всех деталей.

Другое использование переменной из командной строки не рекомендуется. make-файлы, функционирование которых зависит от установок неконтролируемых ими переменных командной среды, не являются дальновидно написанными, поскольку это может привести к тому, что разные пользователи получат разные результаты от одного и того же make-файла. Это противоречит самому предназначению большинства make-файлов.

Такие проблемы были бы особенно вероятны в связи с переменной SHELL, которая обычно присутствует в командной среде для определения выбранной пользователем диалоговой командной оболочки. Было бы очень нежелательно, чтобы этот выбор воздействовал на программу make. Поэтому программа make игнорирует значение переменной SHELL из командной среды.

7. Условные части make-файла

При использовании условной конструкции, часть make-файла обрабатывется или игнорируется, в зависимости от значений переменных. Условные конструкции могут сравнивать значение одной переменной со значением другой переменной или значение переменной с постоянной строкой. Условные конструкции управляют тем, что программа make на самом деле "видит" в make-файле, поэтому они не могут быть использованы для управления командами командной оболочки во время их исполнения.

7.1 Пример условной конструкции

Приведенный ниже пример условной конструкции указывает программе make использовать один набор библиотек, если значением переменной CC является 'gcc', и другой набор библиотек - в противном случае. Его работа основывается на управлении тем, какая из двух командных строк будет использована в правиле в качестве команды. В результате 'CC=gcc' в качестве аргумента программы make изменяет не только используемый компилятор, но также и компонуемые библиотеки.

```
libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
p        $(CC) -o foo $(objects) $(libs_for_gcc)
else
p        $(CC) -o foo $(objects) $(normal_libs)
endif
```

Эта условная конструкция использует три директивы: одну директиву ifeq, одну директиву else и одну директиву endif.

Директива ifeq начинает условную конструкцию и определяет условие. Она содержит два аргумента, разделенных запятой и окруженных круглыми скобками. Для обоих частей производится подстановка значения переменной, после чего они сравниваются. Строки make-файла, следующие за директивой ifeq обрабатываются, если два аргумента идентичны, в противном случае они игнорируются.

При использовании директтивы else, следующие за ней строки должны быть обработаны, если предыдущее условие не выполнилось. В вышеприведенном примере это означает, что вторая альтернатива команды компоновки изпользуется всегда, когда не используется первая альтернатива. Наличие директивы else в условной конструкции не является обязательным.

Директива endif заканчивает условную конструкцию. Каждая условная конструкция должна заканчиваться директивой endif. За ней следует безусловный фрагмент make-файла.

Как показывает этот пример, условная конструкция работает на текстуальном уровне: строки условной конструкции обрабатываются или игнорируются, в соответствии с условиями, как часть make-файла. Именно поэтому более крупные синтаксические элементы make-файла, такие как правила, могут пересекаться с началом или концом условной конструкции.

Когда значением переменной CC является 'gcc', из фрагмента make-файла, приведенного в предыдущем примере, получается такой фрагмент:

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

Когда значением переменной CC является что-либо, отличное от 'gcc', получается такой фрагмент:

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

Эквивалентный результат может быть достигнут еще одним способом, с помощью условной обработки присваивания значения переменной и последующего безусловного ее использования.

```
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)
```

7.2 Синтаксис условных конструкций

Синтаксис простой условной конструкции без использования else следующий:

```
УСЛОВНАЯ-ДИРЕКТИВА
ФРАГМЕНТ-ДЛЯ-ВЫПОЛНЕННОГО-УСЛОВИЯ
endif
```

ФРАГМЕНТ-ДЛЯ-ВЫПОЛНЕННОГО-УСЛОВИЯ может представлять собой любые строки текста, которые будут считаться частью make-файла, если условие истинно. Если условие ложно, никакой другой фрагмент взамен не используется.

Синтаксис сложной условной конструкции следующий:

```
УСЛОВНАЯ-ДИРЕКТИВА
ФРАГМЕНТ-ДЛЯ-ВЫПОЛНЕННОГО-УСЛОВИЯ
else
ФРАГМЕНТ-ДЛЯ-НЕВЫПОЛНЕННОГО-УСЛОВИЯ
endif
```

Если условие истинно, используется ФРАГМЕНТ-ДЛЯ-ВЫПОЛНЕННОГО-УСЛОВИЯ, в противном случае используется ФРАГМЕНТ-ДЛЯ-

НЕВЫПОЛНЕННОГО-УСЛОВИЯ. ФРАГМЕНТ-ДЛЯ-НЕВЫПОЛНЕННОГО-УСЛОВИЯ может занимать любое количество строк текста.

Синтаксис УСЛОВНОЙ-ДИРЕКТИВЫ в простой и в сложной условной конструкции один и тот же. Есть четыре различных директивы, которые проверяют различные условия. Вот их список:

'ifeq (ARG1, ARG2)'

'ifeq 'ARG1' 'ARG2''

'ifeq "ARG1" "ARG2"'

'ifeq "ARG1" 'ARG2''

'ifeq 'ARG1' "ARG2"'

Подставляет значения для всех ссылок на переменные в переменных arg1 и arg2 и сравнивает их. Если они идентичны, обрабатывается ФРАГМЕНТ-ДЛЯ-ВЫПОЛНЕННОГО-УСЛОВИЯ, в противном случае - обрабатывается ФРАГМЕНТ-ДЛЯ-НЕВЫПОЛНЕННОГО-УСЛОВИЯ, если он есть

Часто вы хотите проверить, имеет ли переменная непустое значение. Когда переменная получается в результате сложных вычислений переменных и функций, те подставляемые значения, которые вы рассматриваете как простые, могут, на самом деле, содержать пробельные символы и, таким образом, не считаться пустыми. Однако, вы можете использовать функцию strip (смотрите раздел 8.2 [Функции для работы с текстом]), чтобы избежать интерпретации пробелов как непустых значений. Например, в результате вычисления данной условной директивы:

```
ifeq ($(strip $(foo)),)
ФРАГМЕНТ-ДЛЯ-ПУСТОГО-ЗНАЧЕНИЯ
endif
```

будет обрабатываться ФРАГМЕНТ-ДЛЯ-ПУСТОГО-ЗНАЧЕНИЯ, даже если результат вычисления $(foo) содержит пробельные символы.

ifneq (ARG1, ARG2)

ifneq 'ARG1' 'ARG2'

ifneq "ARG1" "ARG2"

ifneq "ARG1" 'ARG2'

ifneq 'ARG1' "ARG2"

Подставляет значения для всех ссылок на переменные в переменных arg1 и arg2 и сравнивает их. Если они различаются, обрабатывается ФРАГМЕНТ-ДЛЯ-ВЫПОЛНЕННОГО-УСЛОВИЯ, в противном случае - обрабатывается ФРАГМЕНТ-ДЛЯ-НЕВЫПОЛНЕННОГО-УСЛОВИЯ, если он есть

ifdef ИМЯ-ПЕРЕМЕННОЙ

Если переменная с указанным именем имеет непустое значение, обрабатывается ФРАГМЕНТ-ДЛЯ-ВЫПОЛНЕННОГО-УСЛОВИЯ, в противном случае - обрабатывается ФРАГМЕНТ-ДЛЯ-НЕВЫПОЛНЕННОГО-УСЛОВИЯ, если он есть. Переменные, которые нигде не были определены, имеют пустое значение.

Обратите внимание, что директива ifdef проверяет, имеет ли переменная значение. Она не вычисляет переменную, чтобы увидеть, является ли ее значение непустым. Следовательно, проверка с использование директивы ifdef определит выполнение условия для всех переменных, чьи определения имеют вид, отличный от foo =. Чтобы проверить на пустое значение, используйте директиву ifeq ($(foo),). Например, следующий фрагмент make-файла:

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

устанавливает 'yes' в качестве значения переменной frobozz, в то время как такой фрагмент:

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

устанавливает 'no' в качестве значения переменной frobozz.

ifndef ИМЯ-ПЕРЕМЕННОЙ

Если переменная с указанным именем имеет пустое значение, обрабатывается ФРАГМЕНТ-ДЛЯ-ВЫПОЛНЕННОГО-УСЛОВИЯ, в противном случае - обрабатывается ФРАГМЕНТ-ДЛЯ-НЕВЫПОЛНЕННОГО-УСЛОВИЯ, если он есть.

В начале строки с условной директивой разрешается добавлять пробелы, но символ табуляции не разрешен. (Если строка начинается с символа табуляции, она будет рассматриваться как команда для правила.) Кроме этого, дополнительные пробелы или символы табуляции без последствий могут вставляться в любом месте, только не внутри имени директивы и не внутри аргумента. В конце строки может появиться символ начала комментария '#'.

Двумя другими директивами, играющими роль в условной конструкции являются директивы else и endif. Каждая из этих директив записывается в одно слово, без аргументов. В начале строки допускаются и игнорируются добаленные пробелы, а вконце строки - добавленые пробелы и символы табуляции. В конце строки может появиться комментарий, начинающийся с символа '#'.

Условные конструкции воздействуют на то, какие строки make-файла использует программа make. Если условие истинно, make считывает строки ФРАГМЕНТА-ДЛЯ-ВЫПОЛНЕННОГО-УСЛОВИЯ как часть make-файла, если же условие ложно, make полностью игнорирует эти строки. Из этого следует, что синтаксические единицы make-файла, такие как правила, могут быть безопасно разбиты на части началом или окончанием условной конструкции.

make обрабатывает условные конструкции в момент чтения make-файла. Слкдовательно, вы не можете использовать автоматические переменные в условиях условных конструкций, поскольку они не определены до момента выполнения команд (смотрите раздел 10.5.3 [Автоматические переменные]).

Чтобы избежать ужасного беспорядка, не разрешается начинать условную конструкцию в одном make-файле и заканчивать ее в другом. Однако, внутри условной конструкции вы можете написать директиву include, гарантирующую, что вы не пытаетесь закончить условную конструкцию во включаемом файле.

7.3 Условные конструкции, которые проверяют опции

Вы можете написать условную конструкцию, которая проверяет опцию командной строки программы make, такую как '-t', используя переменную MAKEFLAGS вместе с функцией findstring (смотрите раздел 8.2 [Функции подстановки и анализа строк]). Это полезно в тех случаях, когда программы touch недостаточно для того, чтобы файл выглядел обновленным.

Функция findstring определяет, появляется ли одна строка внутри другой в качестве подстроки. Если вы хотите проверить опцию '-t', используйте 't' в качестве первой строки и значение переменной MAKEFLAGS в качестве второй.

Здесь приведен пример того, как ввести соглашение об использовании 'ranlib -t' при окончании отметки архивного файла как обновленного:

```
archive.a: ...
ifneq (,$(findstring t,$(MAKEFLAGS)))
        +touch archive.a
        +ranlib -t archive.a
else
        ranlib archive.a
endif
```

Префикс '+' помечает соответствующие командные строки как "рекурсивные" для того, чтобы они были исполнены, несмотря на использование опции -t. Смотрите раздел 5.6 [Рекурсивное использование make].

8. Функции преобразования текста

Функции позволяют вам производить в make-файле обработку текста для определения обрабатываемых файлов или используемых команд. Вы используете функцию при помощи вызова функции, где вы указываете имя функции и определенный текст (аргументы), который предназначен для обрабоки с помощью функции. Результат работы функции подсталяется в make-файл на место вызова, точно также, как могло быть подставлено значение переменной на место ссылки на нее.

8.1 Синтаксис вызова функции

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

```
$(ФУНКЦИЯ АРГУМЕНТЫ)
```

или так:

```
${ФУНКЦИЯ АРГУМЕНТЫ}
```

Здесь ФУНКЦИЯ представляет собой имя функции, которое берется из небольшого набора имен, встроенных в программу make. Для определения новых функций возможностей нет.

АРГУМЕНТЫ представляют собой аргументы функции. Они отделяются от имени функции одним или более пробелами или символами табуляции, а в том случае, если имеется более, чем один аргумент, то они разделяются запятыми. Эти пробельные символы и запятые не являются частью значения аргумента. Ограничители, используемые вами для ограничения вызова функции, как круглые скобки, так и фигурные, могут появляться среди аргументов только с соответствующими парными символами, другие виды ограничителей могут появляться в одиночку. Если аргументы сами содержат ссылки вызовы других функций или ссылки на переменные, правильнее всего использовать один и тот же вид ограничителей для всех ссылок - то есть, пишите '$(subst a,b,$(x))', а не '$(subst a,b,${x})'. Причина в том что запись с одним видом ограничителей является более ясной, а также, при такой записи, для нахождения конца ссылки ищется только один вид ограничителей.

Текст, соответствующий каждому аргументу, обрабатывается путем подстановки значений переменных и результатов вызовов функций для получения значения аргумента, которое является текстом, с которым работает функция. Подстановка производится в том порядке, в котором аргументы появляются.

Запятые и непарные скобки, круглые или квадратные, не могут явным образом появляться в тексте, соответствующем аргументу, ведущие пробелы не могут

явным образом появляться в тексте, соответствующем первому аргументу. Эти символы могут быть помещены в значение аргумента путем подстановки переменной. Сначала надо определить переменные comma и space, значениями которых являются отдельные символы запятой и пробела, а затем подставить эти переменные там, где требуются такие символы, как показано ниже:

```
comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
# bar is now 'a,b,c'.
```

В этом примере функция subst заменяет каждый пробел вместо запятой во всем значении переменной foo, после чего результат работы функции подставляется на место ее вызова.

8.2 Функции подстановки и анализа строк

Ниже приведены некоторые функции, которые обрабатывают строки:

'$(subst ФРАГМЕНТ,ЗАМЕНА,ТЕКСТ)'

Выполняет текстуальную замену в тексте ТЕКСТ: каждое вхождение ФРАГМЕНТА заменяется на ЗАМЕНУ. Результат подставляется на место вызова функции. Например, на место следующего вызова функции:

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

подставляется строка 'fEEt on the strEEt'.

'$(patsubst ШАБЛОН,ЗАМЕНА,ТЕКСТ)'

Находит в ТЕКСТЕ разделенные пробельными символами слова, соответствующие ШАБЛОНУ, и заменяет их на ЗАМЕНУ. При этом шаблон может содержать символ '%', который действует как шаблон, соответствующий любому количеству любых символов внутри слова. Если в ЗАМЕНЕ также содержатся символы '%', то они заменяются текстом, соответствующим символу '%' в шаблоне.

Специальное значение символа '%' в вызове функции patsubst может быть отключено предшествующим символом '\'.Специальное назначение символа '\', который в противном случае отменял бы специальное назначение последующего символа '%', может быть отменено еще одним символом '\'. Символы '\', отменяющие специальное назначение символов '%' или других символов '\', удаляются из шаблона перед тем, как он будет сравниваться с именами файлов или в него будет подставляться основа. Символы '\', которые заведемо не влияют на трактовку символа '%', остаются нетронутыми. Например, в шаблоне 'the\%weird\ \%pattern\\' фрагмент 'the%weird\' предшествует действующему символу '%', а фрагмент 'pattern\\' следует за ним. С двумя заключительными символами '\' ничего не происходит, поскольку они не могут воздействовать ни на какие символы '%'.

Пробельные символы между словами преобразуются в одиночные пробелы, ведущие и ведомые пробельные символы отбрасываются.

Например, следующий вызов функции

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

порождает значение 'x.c.o bar.o'.

Ссылки с заменой (смотрите раздел 6.3.1 [Ссылки с заменой]) являются более простым способом получить результат, аналогичный использованию функции patsubst. Например, такая ссылка с заменой:

```
$(VAR:PATTERN=REPLACEMENT)
```

эквивалентна вызову функции patsubst:

```
$(patsubst PATTERN,REPLACEMENT,$(VAR))
```

Еще одно сокращение упрощает одно из наиболее частых использований функции patsubst: замену суффикса в конце именем файлов. Такая ссылка с заменой:

```
$(VAR:SUFFIX=REPLACEMENT)
```

эквивалентна вызову функции patsubst:

```
$(patsubst %SUFFIX,%REPLACEMENT,$(VAR))
```

Например, у вас мог бы быть список объектных файлов:

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

Чтобы получить список соответствующих исходных файлов, вы могли

бы просто написать:

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

вместо испрользования общей формы:

```
$(patsubst %.o,%.c,$(objects))
```

'$(strip СТРОКА)'

Удаляет ведущие и ведомые пробелы из СТРОКИ заменяет каждую внутреннюю последовательность из одного или более пробельного символа на один пробел. Таким образом, результатом вызова '$(strip a b c)' является 'a b c'.

Функция strip может быть очень полезной при использовании ее вместе с условными конструкциями. При сравнении чего-либо с пустой строкой '' с помощью

директив ifeq или ifneq вы обычно хотите, чтобы строка, состоящая только из пробельных символов, была равна пустой строке (смотрите главу 7 [Условные конструкции]).

Таким образом, в следующем фрагменте make-файла, возможно, не удастся получить желаемый результат:

```
.PHONY: all
ifneq   "$(needs_made)" ""
all: $(needs_made)
else
all:;<htmlurl name="@echo" url="mailto:@echo">
'Nothing to make!'
endif
```

Замена в директиве ссылки на переменную $(needs_made) на вызов функции $ (strip $(needs_made)) ifneq сделало бы этот фрагмент более корректным.

'$(findstring ФРАГМЕНТ,СТРОКА)'

Ищет в СТРОКЕ вхождение ФРАГМЕНТА. Если вхождение есть, то результатом функции является ФРАГМЕНТ, в противном случае результатом является пустая строка. Вы можете использовать эту функцию в условной конструкции для того, чтобы проверить наличие специальной подстроки в данной строке. Таким образом, следующих вызовы функций:

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

порождают, соответственно, значения 'a' и '' (пустую строку). Смотрите раздел 7.3 [Проверка опций] для информации опрактическом применении функции findstring.

'$(filter ШАБЛОН...,ТЕКСТ)'

Удаляет из ТЕКСТА все разделенные пробельными символами слова, которые не соответствуют ни одному из шаблонных слов, возвращая только слова, соответствующие, по крайней мере, одному из шаблонов. Шаблоны пишутся с использованием символа '%', также как и шаблоны, используемые в описанной выше функции patsubst.

Функция filter может быть использована для выделения из переменной различных типов строк (таких, как имена файлов). Например, приведенный ниже фрагмент make-файла:

```
sources := foo.c bar.c baz.s ugh.h
foo: $(sources)
        cc $(filter %.c %.s,$(sources)) -o foo
```

говорит о том, что файл 'foo' зависит от файлов 'foo.c', 'bar.c', 'baz.s' и 'ugh.h', но только 'foo.c', 'bar.c' и 'baz.s' должны быть определены для компилятора в командной строке.

'$(filter-out ШАБЛОН...,ТЕКСТ)'

Удаляет из ТЕКСТА все разделенные пробельными символами слова, которые соответствуют какому-нибудь из шаблонных слов, возвращая только слова, не соответствующие ни одному из шаблонов. Это является точной противоположностью функции filter.

Например, при таких определениях переменных:

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

приведенный ниже фрагмент make-файла приводит к генерации списка, содержащего все объектные файлы, не указанные в переменной mains

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

'$(sort СПИСОК)'

Список в лексическом порядке слова из СПИСКА, удаляя дублирующиеся слова. Результатом является список слов, разделенных одиночными пробелами. Таким образом, при таком вызове функции:

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

получается значение 'bar foo lose'.

Кстати, поскольку функция sort удаляет дублирующиеся слова, вы можете использовать ее для этой цели, даже если вам не нужны возможности, связанные с сортировкой.

Вот реалистичный пример использования функций subst и patsubst. Предположим, что make-файл использует переменную VPATH для определения списка каталогов, в которых программа make должна искать файлы зависимости (смотрите раздел 4.3.1 [VPATH: Путь поиска для всех зависимостей]). Это пример показывает, как указать C-компилятору искать заголовочные файлы в том же списке каталогов.

Значение переменной VPATH представляет собой список каталогов, разделенных двоеточиями, например 'src:../headers'. Сначала надо использовать функцию subst для замены двоеточий на пробелы:

```
$(subst :, ,$(VPATH))
```

В результате получается значение 'src ../headers'. Затем следует использовать функцию patsubst для того чтобы подставить перед каждым именем каталога оицию '-I'. Получившееся значение может быть добавлено к значению переменной CFLAGS, которая автоматически передается C-компилятору, как показано ниже:

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

В итоге к имевшемуся ранее значению переменной CFLAGS добавляется фрагмент '-Isrc -I../headers'. Из-за использования директивы override, присваивание нового значения будет происходить даже в том случае, если предыдущее значение переменной CFLAGS было определено при помощи аргумента командной строки.

8.3 Функции для обработки имен файлов

Несколько дополнительных встроенных функций специально ориентированы на работу с именами файлов и списками имен файлов.

Каждая из нижнпреведенных функций выполняет специальное преобразование над именем файла. Аргкмент функции рассматривается как последовательность имен файлов, разделенных пробельными символами (ведущие и ведомые пробельные символы игнорируются). Все имена файлов из последовательности преобразуются одинаковым образом, а результаты пребразования каждого файла сцепляются в одно значение с использованием между ними одиночного пробела.

'$(dir ИМЕНА...)'

Выделяет часть, определяющую каталог, из каждого имени файла, указанного в списке ИМЕН. Часть имени файла, определяющая каталог, представляет собой часть имени от его начала до последнего символа '/' (включительно). Если в имени файла не содержится символа '/', частью, определяющей каталог, является './'. Например, при таком вызове функции:

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

в качестве результата получается 'src/ ./'.

'$(notdir ИМЕНА...)'

Выделяет из каждого имени файла, указанного в списке ИМЕН, то, что не входит в часть, определяющую каталог,. Если имя файла не содержит ни одного символа '/', то оно остается неизмененным. В противном случае, из имени файла удаляется все то, что в нем расположено до последнего символа '/'.

Имя файла, заканчивающееся символом '/' преобразуется в пустую строку. Это является удачным, поскольку означает, что результат не всегда содержит такое же количество разделенных пробельными символами имен файлов, как и аргумент, но мы не видим другой подходящей альтернативы. Например, при таком вызове функции:

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

в качестве результата получается 'foo.c hacks'.

'$(suffix ИМЕНА...)'

Выделяет суффикс каждого имени файла из списка ИМЕН. Если имя файла содержит точку, то суффиксом является часть имени от его последней точки до конца. В противном случае, суффиксом является пустая строка. Часто это означает, что результат функции будет пустым при непустом аргументе, а при аргументе, содержащем несколько имен файлов, результат может содержать их в меньшем количестве:

Например, при таком вызове функции:

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

в качестве результата получается '.c'.

'$(basename ИМЕНА...)'

Выделяет из каждого имени файла из списка ИМЕН базовое имя - все то, что не относится к суффиксу. Если имя файла содержит точку, то базовым именем является часть имени от его начала последней точки (исключительно). В противном случае, базовым именем является все имя файла. Например, при таком вызове функции:

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

в качестве результата получается 'src/foo hacks'.

'$(addsuffix СУФФИКС,ИМЕНА...)'

Аргумент ИМЕНА рассматривается как последовательность имен, разделенных пробельными символами, а СУФФИКС используется как одно целое. Значение аргумента СУФФИКС добавляется в конец каждого отдельного имени и получившиеся удлиненные имена сцепляются, с одиночными пробелами между собой. Например, при таком вызове функции:

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

в качестве результата получается 'foo.c bar.c'.

'$(addprefix ПРЕФИКС,ИМЕНА...)'

Аргумент ИМЕНА рассматривается как последовательность имен, разделенных пробельными символами, а ПРЕФИКС используется как одно целое. Значение аргумента ПРЕФИКС двставляется в начало каждого отдельного имени и получившиеся удлиненные имена сцепляются, с одиночными пробелами между собой. Например, при таком вызове функции:

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

в качестве результата получается 'src/foo src/bar'.

'$(join СПИСОК1,СПИСОК2)'

Сцепляет слова из двух аргументов: два первых слова (по одному из каждого аргумента), в результате сцепления, образуют первое слово результата, два вторых слова образуют второе слово результата, и так далее. Таким образом, n-е слово результата образуется из n-х слов каждого аргумента. Если в одном из аргументов слов больше, чем в другом, избыточные слова копируются в результат неизмененными.

Например, при вызове '$(join a b,.c .o)' в качестве результата получается 'a.c b.o'.

Пробельные символы между словами в списке не сохраняются - они заменяются одиночными пробелами.

Эта функция может слить результаты функций dir and notdir, порождая первоначальный список файлов, переданный этим двум функциям.

'$(word N,ТЕКСТ)'

Возвращает N-е слово ТЕКСТА. Допустимые значения переменной N начинаются с 1. Если N больше, чем количество слов в ТЕКСТЕ, результатом является пустое значение. Например, при таком вызове:

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

результатом будет 'bar'.

'$(words ТЕКСТ)'

Возвращает количество слов в ТЕКСТЕ. Таким образом, последнее слово текста может быть получено при помощи вызова '$(word $(words TEXT),TEXT)'.

'$(firstword ИМЕНА...)'

Аргумент ИМЕНА рассматривается как последовательность имен, разделенных пробельными символами. Значением является первое имя в последовательности. Оставшаяся часть имени игнорируется.

Например, при таком вызове функции:

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

в качестве результата получается 'foo'. Хотя вызов $(firstword TEXT) аналогичен вызову $(word 1,TEXT), функция firstword остается в употреблении из-за ее простоты.

'$(wildcard ШАБЛОН)'

Аргумент ШАБЛОН является шаблоном имени файла, обычно содержащим шаблонные символы (как шаблонах имени файла, используемых в командной

оболочке). Результатом функции wildcard является разделенный пробелами список имен существующих файлов, удовлетворяющих шаблону. Смотрите раздел 4.2 [Использование шаблонных символов в именах файлов].

8.4 Функция foreach

Функция foreach сильно отличается о других функций. При ее использовании определенная часть текста используется повторно, при этом каждый раз над ней выполняются различные подстановки. Это похоже на команду for в командной оболочке sh и на команду csh в командной C-оболочке csh.

Синтакисис функции foreach слудующий:

```
$(foreach ПЕРЕМЕННАЯ,СПИСОК,ТЕКСТ)
```

Первые два аргумента, ПЕРЕМЕННАЯ и СПИСОК, вычисляются до того, как что-либо еще будет сделано; обратите внимание, что последний аргумент, ТЕКСТ, не вычисляется в это время. Затем для каждого слова из вычисленного значения аргумента СПИСОК, переменная с именем, полученным из вычисленного значения аргумента ПЕРЕМЕННАЯ, получает в качестве значения это слово, и аргумент ТЕКСТ вычисляется. Предполагается, что ТЕКСТ содержит ссылки на эту переменную, поэтому результат ее вычисления будет каждый раз различным.

В итоге аргумент ТЕКСТ вычисляется столько раз, сколько разделенных пробельными симврролами слов есть в СПИСКЕ. Результаты множественных вычислений аргумента ТЕКСТ сцепляются, с пробелами между ними, порождая результат функции foreach.

В приведенном ниже простом примере в качестве значения переменной 'files' устанавливается список всех файлов в каталогах, перечисленных списке, представленном переменной 'dirs':

```
dirs := a b c d
files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

В данном случае значением аргумента является '$(wildcard $(dir)/*)'. При первой итерации в качестве значения dir берется 'a', что приводит к такому же результату, как и вызов '$(wildcard a/*)', при второй итерации получается результат, аналогичный вызову '$(wildcard b/*)', а при третьей - вызову '$(wildcard c/*)'.

Этот пример дает такой же результат (за исключением установки переменной dirs), как и следующий пример:

```
files := $(wildcard a/* b/* c/* d/*)
```

Когда аргумент ТЕКСТ сложен, вы можете улучшить читабельность, дав ему имя при помощи дополнительной переменной:

```
find_files = $(wildcard $(dir)/*)
dirs := a b c d
```

```
files := $(foreach dir,$(dirs),$(find_files))
```

В этом примере мы для этого используем переменную find_files. Мы используем просто символ '=' для того, чтобы определить рекурсивно-вычисляемую переменную, и поэтому ее значение, на самом деле, содержит вызов функции, который повторно вычисляется под управлением функции foreach - для упрощенно-вычисляемой переменной это бы не сработало, так как функция wildcard была бы вызвана только один раз, во время определения переменной find_files.

Функция foreach не оказывает необратимого эффекта на переменную, соответствующую аргументу ПЕРЕМЕННАЯ - ее значение и разновидность после вызова функции foreach остаются такими же, как и были ранее. Другие значения, которые берутся из СПИСКА, находятся в действии только временно, в период выполнения функции foreach. Переменная, соответствующая аргументу ПЕРЕМЕННАЯ, в период выполнения функции foreach является упрощенно вычисляемой переменной. Если переменная, соответствующая аргументу ПЕРЕМЕННАЯ, была неопределенной перед вызовом функции foreach, то она остается неопределенной и после вызова. Смотрите раздел 6.2 [Две разновидности переменных].

Вы должны быть внимательны прииспользовании сложных переменных выражений, значения которых используются в качестве имен переменных, поскольку многие странные значения являются допустимыми именами переменных, но, вероятно, они представляют собой не то, чего вы хотели. Например, такое присваивание:

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

могло быть полезным, если бы в значении переменной find_files была ссылка на переменную с именем 'Esta escrito en espanol!', но наиболее вероятно, что это является ошибкой.

8.5 Функция origin

Функция origin отличается от большинства других функций тем, что она не обрабатывает значение переменной - она дает вам определенную информацию о переменной. В частности, она дает вам информацию о происхождении переменной.

Функция origin имеет такой синтаксис:

```
$(origin ПЕРЕМЕННАЯ)
```

Обратите внимание на то, что аргумент ПЕРЕМЕННАЯ - это имя переменной, на которую делается запрос, а не ссылка на эту переменную. Следовательно, обычно при написании аргумента вы не будете использовать символ '$' или круглые скобки. (Однако, вы можете использовать в имени ссылку на переменную, если хотите, чтобы имя не было фиксированным.)

Результатом этой функции является строка, дающая вам информацию о том, как была определена переменная, определяемая аргументом ПЕРЕМЕННАЯ:

'undefined'

если эта переменная нигде не была определена.

'default'

Если эта переменная имеет определение, действующее по умолчанию, что обычно имеет место для переменной CC и подобных ей. Смотрите раздел 10.3 [Переменные, используемые неявными правилами]. Обратите внимание, что если вы переопределили переменную, имеющую значение по умолчанию, функция origin даст вам информацию о происхождении, соответствующую более позднему определению.

'environment'

Если эта переменная была определена как переменная командной среды, и при этом не указана опция '-e' (смотрите раздел 9.7 [Обзор опций]).

'environment override'

Если эта переменная была определена как переменная командной среды, и при этом указана опция '-e' (смотрите раздел 9.7 [Обзор опций]).

'file'

Если эта переменная была определена в make-файле

'command line'

Если эта переменная была определена в командной строке

'override'

Если эта переменная была определена в make-файле при помощи директивы override (смотрите раздел 6.7 [Директива override]).

'automatic'

Если эта переменная является автоматической переменной, определяемой для выполнения команд в каждом правиле (смотрите раздел 10.5.3 [Автоматические переменные]).

Эта информация полезна в первую очередь (не считая вашего любопытства) для определения, хотите ли вы доверять значению переменной. Например, предположим, что у вас есть make-файл 'foo', в котором происходит включение другого make-файла - 'bar'. Вы хотите, чтобы при использовании команды 'make -f bar' переменная bletch была определена в файле 'bar', даже если в командной среде содержится определение переменной bletch. Однако, если в файле 'foo' перед включением файла 'bar' определена если переменная bletch, вы не хотите перекрывать это определение. Это могло бы быть реализовано с использованием в

файле 'foo' директивы override, что давало бы этому определению преимущество перед более поздним определением в фале 'bar', но, к сожалению, директива override перекрыла бы также любые определения, данные в командной строке. Таким образом, файл 'bar' мог бы содержать такой фрагмент:

```
ifdef bletch
  ifeq "$(origin bletch)" "environment"
    bletch = barf, gag, etc.
  endif
endif
```

Если бы переменная bletch была определена в командной среде, это вызвало бы ее переопределение.

Если вы хотите перекрыть предыдущее определение переменной bletch, если она определена в командной среде, даже при использовании опции '-e', вы могли бы вместо этого написать:

```
ifneq "$(findstring environment,$(origin bletch))" ""
bletch = barf, gag, etc.
endif
```

В данном случае переопределение имеет местов том случае, если при вызове '$ (origin bletch)' возвращается значение 'environment' или 'environment override'. Смотрите раздел 8.2 [Функции подстановки и анализа строк].

8.6 Функция shell

Функция shell отличается от любой другой функции, кроме функции wildcard (смотрите раздекл 4.2.3 [Функция wildcard]), тем, что она общается с внешним, по отношению к программе make, миром.

Функция shell выполняет те же действия, которые выполняют обратные апострофы (`') в большинстве командных оболочек: она выполняет подстановку результатов команд. Это означает, что она принимает аргумент, являющийся командой командной оболочки, а ее результатом является выход этой команды. Единственной работой, выполняемой программой make над результатом перед подстановкой его в окружающий текст, является преобразование символов перевода строки в пробелы.

Команды, запускаемые при вызовах функции shell, запускаются в момент выполнения вызовов функции. В большинстве случаев, это происходит в момент считывания make-файла. Исключение состоит в том, что вызовы функции в командах правила выполняются в момент запуска команд, и это правило, точно так же, как и к другим, применимо и к функции shell.

Ниже приведены некоторые примеры использования функции shell. В первом случае:

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

в качестве значения переменной contents устанавливается содержимое файла 'foo', с пробелом (а не символом перевода строки) в качестве разделителей строк файла. А во втором случае:

```
files := $(shell echo *.c)
```

в качестве значения переменной contents устанавливается результат поиска файлов текущего каталога, соответствующих шаблону '*.c'. Если программа make не использует очень необычную команду оболочки, такой вызов дает такой же результат, что и вызов '$(wildcard *.c)'.

9. Как запускать make

make-файл, определяющий, как перекомпилировать программу, может быть использован более, чем одним способом. Самое простое использование заключается в перекомпиляции каждого файла, зависимости которого были обновлены. Обычно make-файлы пишутся так, чтобы при запуске программы make без аргументов она делала именно это.

Но вы могли бы захотеть обновить только некоторые из файлов, вы могли бы захотеть использовать другой компилятор или другие опции компилятора, вы могли бы захотеть просто выяснить, у какие файлы нуждаются в обновлении, не изменяя их.

Передавая программе make аргументы при ее запуске, вы можете сделать все это, а также много другого.

Возвращаемым результатом программы make всегда является одно из трех значений:

0	Возвращаемым результатом является нуль, если программа make выполнилась успешно
2	Возвращаемым результатом является двойка, если программа make обнаруживает какую-либо ошибку. Она выдаст на экран сообщение, описывающее, какая именно ошибка произошла.
1	Возвращаемым результатом является единица, если вы используете опцию '-q' и программа make определяет, что какая-то цель не является, на момент запуска программы make, обновленной. Смотрите раздел 9.3 [Вместо исполнения команд].

9.1 Аргументы для определения make-файла

Для определения имени make-файла служит опция '-f' или '--file' (работает также опция '--makefile'). Например, фрагмент командной строки '-f altmake' указывает на использование в качестве make-файла файла с именем 'altmake'.

Если вы используете опцию '-f' несколько раз, указывая за каждым ее вхождением аргумент, все определенные таким образом файлы совместно используются в качестве make-файлов.

Если вы не используете опцию '-f' или '--file', то в качестве имен по умолчанию пробуются имена 'GNUmakefile', 'makefile' и 'Makefile', именно в таком порядке, и в роли make-файла используется первый из этих трех, который существует или может быть порожден (смотрите главу 3 [Написание make-файлов]).

9.2 Аргументы для определения главных целей

Главные цели представляют собой цели, которые в конечном счете стремится обновить программа make. Другие цели обновляются только в том случае, если они появляются в качестве зависимостей гланых целей или в качестве зависимостей зависимостей главных целей, и т.д..

По умолчанию, главной целью является первая цель в make-файле (не считая целей, начинающихся с точки). Следовательно, make-файлы обычно пишутся так, чтобы первой целью была цель для компиляции всей программы или всех программ, которые они описывают. Если первое правило make-файла имеет несколько целей, то только первая цель правила, а не весь список, становится главной целью по умолчанию.

Вы можете определить другую главную цель или главные цели при помощи аргументов программы make. Используйте имя главной цели как аргумент. Если вы определяете несколько главных целей, то программа make обрабатывает каждую из них по очереди, в том порядке, в котором вы их упоминаете.

Любая цель make-файла может быть определена в качестве главной цели (если она не начинается с символа '-' и не содержит символ '=', так как в этих случаях это будет воспринято, соответственно, как опция или определение переменной). В качестве главной цели может быть определена даже цель, не содержащаяся в make-файле, если программа make может найти неявное правило, которое указывает, как ее породить.

Одно из использований определения главной цели возможно в том случае, если вы хотите откомпилировать только часть программы или только одну из нескольких программ. Определите в качестве главной цели каждый файл, который вы хотите заново породить. Например, рассмотрим каталог, содержащий несколько программ, с make-файлом, который начинается примерно так:

```
.PHONY: all
all: size nm ld ar as
```

Если вы работаете над программой size, то вы могли бы захотеть написать в качестве команды 'make size', и при этом были бы перекомпилированы только файлы этой программы.

Еще одно использование определения главной цели состоит в порождении файлов, который при работе с главной целью по умолчанию не породились. Например, это может быть файл с отладочной информацией или версия программы, компилирующаяся специально для тестирования, которые могут иметь правило в make-файле, но не являться зависимостью главной цели про умолчанию.

Определение главной цели также может использоваться при запуске команд, связанных с целями-именами действий (смотрите раздел 4.4 [Цели-имена действий]) или пустыми целями (смотрите раздел 4.6 [Пустые целевые файлы для фиксации событий]). Многие make-файлы содержат цель-имя действия и именем

'clean', которая удаляет все, за исключением исходных файлов. Естественно, это делается только в том случае, если вы явно требуете этого при помощи 'make clean'. Ниже приведен список типичных имен целей-имен действий и пустых целей. Смотрите раздел 14.3 [Стандартные цели], для детального списка всех стандартных имен целей, которые используются программными пакетами GNU.

‘all'	Порождает все цели верхнего уровня, которые упомянуты в make-файле
‘clean'	Порождает все цели верхнего уровня, которые упомянуты в make-файле
‘mostlyclean'	Аналогично цели 'clean', но может воздержаться от удаления некоторых файлов, перекомпилировать которые обычно не хочется. Например, в GCC цель 'mostlyclean' не удаляет файл 'libgcc.a', поскольку ее перекомпиляция требуется редко и занимает много времени.
'distclean' **'realclean'** **'clobber'**	Любая из этих целей могла бы быть определена для удаления некоторых файлов, которые не удаляются при помощи цели 'clean'. Например, таким способом могли бы удаляться конфигурационные файлы или связи, которые обычно создаются в качестве подготовки к компиляции, даже если сам make-файл не может создать эти файлы.
'install'	Копирует исполняемый файл в каталог, в котором пользователи обычно ищут команды, копирует все вспомогательные файлы, которые используются исполняемым файлом, в тот каталог, в котором он их ищет.
'print'	Печатает список измененных исходных файлов.
'tar'	Создает из исходных файлов tar-файл.
'shar'	Создает из исходных файлов архив командной оболочки (shar-файл).
'dist'	Создает из исходных файлов распространяемый файл. Это может быть tar-файл или shar-файл, или же сжатая версия одного из них, или даже что-нибудь еще, не упомянутое здесь.
'TAGS'	Обновляет таблицу тэгов программы, получаемой при помощи данного make-файла.
'check' **'test'**	Выполняет самотестирование программы, получаемой при помощи данного make-файла.

9.3 Вместо исполнения команд

make-файл указывает программе make, как определить, является ли цель обновленной и как обновлять каждую цель. Но обновление целей - это не всегда

то, что вам хочется. Некоторые опции определяют другие действия программы make.

'-n' **'--just-print'** **'--dry-run'** **'--recon'**	"Нет операций". Действия программы make состоят в выводе на экран тех команд, которые использовались бы для обновления целей, но без реального их выполнения.
'-t' **'--touch'**	"Изменение времени обновления". Действия программы make состоят в том, что цели помечаются как обновленные, без реального их изменения. Другими словами, программа make иммитирует компиляцию целей, но на самом деле не изменяет их содержимого.
'-q' **'--question'**	"Запрос". Действия программы make состоят в выявлении того, являются ли цели уже обновленными, без выдачи каких-либо сообщений, и при этом, независимо от результатов, не выполняется ни одной команды.
'-W FILE' **'--what-if=FILE'** **'--assume-new=FILE'** **'--new-file=FILE'**	"А что, если". За каждой опцией '-W' следует имя файла. В качестве времени изменения данных файлов программой make запоминается текущее время, хотя реальные времена модификации остаются прежними. Вы можете использовать опцию '-W' вместе с опцией '-n' для того, чтобы увидеть что произошло бы, если вам потребовалось изменить определенные файлы.

При указанной опции '-n', программа make печатает команды, которые обычно она бы выполняла, но без их выполнения.

При указанной опции '-t', программа make игнорирует команды в правилах и использует (точнее, создает эффект использования) команду touch для каждой цели, которую нужно заново порождать. При этом также печатается команда touch, если не используется опция '-s' или цель .SILENT. Для ускорения работы, программа make, в действительности, не вызывает программу touch. Она напрямую выполняет ее работу.

При указанной опции '-q', программа make ничего не печатает и не выполняет никаких команд, но возвращаемый ею результат является нулем, если, и только если, рассматриваемые ею цели уже являются обновленными. Если результатом является единица, то это значит, что необходимо выполнить какое-либо обновление. Если программа make обнаруживает ошибку, то результатом является двойка, поэтому вы можете отличить ошибку от необновленной цели.

Является ошибкой использование более, чем одной из этих трех опций в одном вызове программы make.

Опции '-n', '-t', и '-q' не воздействуют на командные строки, которые начинаются с символа '+' или содержат в качестве подстроки '$(MAKE)' или '${MAKE}'. Обратите внимание, что только строка, начинающаяся с символа '+' или содержащая

подстроку '$(MAKE)' или '${MAKE}' выполняется, невзирая на эти опции. Другие строки в том же правиле не выполняются, если они также не начинаются с символа '+' и не содержат в качестве подстроки '$(MAKE)' или '${MAKE}'. (Смотрите раздел 5.6.1 [Как работает переменная make].)

Опция '-W' обеспечивает две возможности:

- Если вы также используете опцию '-n' или '-q', вы можете увидеть, что делала бы программа make, если бы вам потребовалось изменить некоторые файлы.
- Без опций '-n' и '-q', когда программа make на самом деле выполняет команды, опция '-W' может заставить программу make действовать так, как если бы некоторые файлы были изменены, без реального изменения этих файлов.

Обратите внимание, что опции '-p' and '-v' позволяют вам получить другую информацию о программе make и о используемых make-файлах (смотрите раздел 9.7 [Обзор опций]).

9.4 Предотвращение перекомпиляции некоторых файлов

Иногда у вас может быть иметься измененный исходный файл, но при этом вы можете не захотеть перекомпилировать все файлы, которые зависят от него. Например, предположим, что вы добавляете макрос или объявление в заголовочный файл, от которого зависят многие другие файлы. Следуя установленному порядку, программа make предполагает, что любое изменение в заголовочном файле требует перекомпиляции всех зависимых файлов, но вы знаете, что они не обязательно должны быть перекомпилированы и вы бы предпочли не тратить время, ожидая выполнения их компиляции.

Если перед изменением заголовочного файла вы предвидите эту проблему, то вы можете использовать опцию '-t'. Эта опция указывает программе make не выполнять команды из правил, а вместо этого пометить цель как обновленную путем изменения времени ее последней модификации. Вам следует придерживаться такого порядка действий:

1. Используйте команду make для перекомпиляции исходных файлов, которые на самом деле требуют перекомпиляции.
2. Сделайте изменения в исходных файлах.
3. Используйте команду 'make -t' для отметки всех объектных файлов как обновленных. Когда вы в следующий раз запустите программу make, изменения в заголовочных файлах не вызовут никакой перекомпиляции.

Если вы уже изменили заголовочный файл в тот момент, когда некоторые файлы требуют перекомпиляции, делать это слишком поздно. Вместо этого, вы можете использовать опцию '-o <имя файла>', которая отмечает указанный файл как "старый" (смотрите раздел 9.7 [Обзор опций]). Это означает, что сам этот файл не будет заново порожен и ничего больше не будет заново порождено по причине изменения этого файла. Следуйте такому порядку действий:

1. Перекомпилируйте исходные файлы, которые нуждаются в компиляции по причинам, не зависящим от определенного заголовочного файла, при помощи команды 'make -o <имя заголовочного файла>'. Если речь идет о

нескольких заголовочных файлах, используйте отдельную опцию '-o' для каждого заголовочного файла.

2. Измените время обновления всех объектных файлов при помощи команды 'make -t'.

9.5 Перекрывающиеся переменные

Аргумент, который содержит символ '=', определяет значение переменной: аргумент 'v=x' устанавливает x в качестве значения переменной v. Если вы определяете значение таким образом, то в make-файле все обычные присваивания значения этой же переменной игнорируются - мы говорим, что они перекрываются аргументом командной строки.

Наиболее типичным использованием этой возможности является передача дополнительных опций компиляторам. Например, в грамотно написанном make-файле в каждую команду, которая запускает C-компилятор, включается переменная CFLAGS, поэтому файл 'foo.c' должен компилироваться примерно так:

```
cc -c $(CFLAGS) foo.c
```

Таким образом, в какое значение вы ни устанавливаете переменную CFLAGS, это воздействует на каждый сеанс компиляции. make-файл, возможно, определяет обычное значение для переменной CFLAGS, например так:

```
CFLAGS=-g
```

Каждый раз, когда вы запускаете программу make, вы, если хотите, можете перекрыть это значение. Например, если вы напишете в командной строке 'make CFLAGS='-g -O'', каждый раз C-компиляция будет выполняться при помощи командной строки 'cc -c -g -O'. (Это иллюстрирует то, как вы можете использовать в командной оболочке апострофы для включения в значение переменной, при ее перекрытии, пробелов и других специальных символов.)

Переменная CFLAGS - это только одна их многих стандартных переменных, которые существуют только для того, чтобы вы могли изменять их таким образом. Смотрите раздел 10.3 [Переменные, используемые неявными правилами], где приведен полный их список.

Вы можете также написать make-файл, работа которого зависела бы от дополнительных переменных, введенных в употребление вами, что дало бы пользователю возможность управлять другими аспектами работы make-файла путем изменения переменных.

Когда вы перекрываете переменную при помощи аргумента командной строки, вы можете определить либо рекурсивно вычисляемую переменную, либо упрощенно вычисляемую переменную. В примерах, показанных выше, создается рекурсивно вычисляемая переменная; для создания упрощенно вычисляемой переменной, пишите ':=' вместо '='. Но если вы не хотите включать в определяемое вами значение ссылку на переменную или вызов функции, то становится безразличным, какого вида переменную вы создаете.

Если один способ, при помощи которого make-файл может изменить переменную, которую вы перекрыли. Он заключается в использовании директивы override,

которая представляет собой строку, которая выглядит примерно так: 'override VARIABLE = VALUE' (смотрите раздел 6.7 [Директива override]).

9.6 Проверка компиляции программ

Обычно, когда происходит ошибка при выполнении команды командной оболочки, программа make немедленно прекращает работу, возвращая ненулевой результат. Никаких команд ни для какой цели после этого не выполняется. Ошибка подразумевает, что главная цель не может быть корректно заново порождена, и программа make, как только узнает об ошибке, сразу сообщает об этом.

Когда вы компилируете программу, которую вы только что изменили, это - не то, что вы хотите. Вместо этого, вы бы предпочли, чтобы программа make попробовала откомпилировать каждый файл, который позволяет это сделать, чтобы показать вам как можно больше ошибок компиляции.

В таких случаях вам следует использовать опцию '-k' или '--keep-going'. Она указывает программе make продолжать работу с другими зависимостями обновляемых целей, при необходимости заново порождая эти зависимости, перед прекращением работы возвратом ненулевого результата. Например, в случае ошибки при компиляции одного объектного файла, программа make, запущенная при помощи командной строки 'make -k', продолжит компиляцию других объектных файлов, хотя уже и известно, что их компоновка будет невозможна. Помимо продолжения работы после после неудачной команды командной оболочки, программа make, запущенная при помощи командной строки 'make -k', будет продолжать работу как можно дольше после того, как она выяснит, что не знает, как породить целевой файл или файл зависимости. Такая ситуация всегда будет вызывать сообщение об ошибке, но без опции '-k' это является фатальной ошибкой (смотрите раздел 9.7 [Обзор опций]).

Обычное поведение програмы make предполагает, что вы стремитесь получить обновленные главные цели - как только программа make выясняет, что это невозможно, она также может сразу сообщить о неудаче в своей работе. Опция '-k' указывает на то, что на самом деле программа make используется для проверки как можно большего количества изменений, сделанных в программе, возможно, для поиска нескольких независимых проблем для того, чтобы вы могли исправить их перед следующей попыткой компиляции. Именно поэтому M-x compile, команда редактора Emacs, по умолчанию передает программе make опцию '-k'.

9.7 Обзор опций

Вот список всех опций, которые понимает программа make:

'-b' **'-m'**	Эти опции включены для совместимости с другими версиями программы make и они игнорируются.

'-C <каталог>' **'--directory=<каталог>'**	Перед чтением make-файлов переходит в каталог, определяемый аргументом <каталог>. Если определены несколько опций '-C', то каждая из них интерпретируется относительно предыдущей: фрагмент командной строки '-C / -C etc' эквивалентен такому фрагменту: '-C /etc'. Это обычно используется при рекурсивных вызовах программы make. (смотрите раздел 5.6 [Рекурсивное использование программы make]).
'-d' **'--debug'**	Помимо обычной работы, выводит отладочную информацию. Отладочная информация говорит о том, какие файлы рассматриваются на предмет порождения их заново, времена изменения каких файлов сравниваются и с какими результатами, какие файлы на самом деле нуждаются в порождении их заново, какие неявные правила принимаются в расчет и какие из них применяются - все, что имеет отношение к тому, как программа make решает, что делать.
'-e' **'--environment-overrides'**	Дает переменным, взятым из командной среды, приоритет перед переменными из make-файлов. Смотрите раздел 6.9 [Переменные из командной среды].
'-f <имя файла>' **'--file=<имя файла>'** **'--makefile=<имя файла>'**	Читает в качестве make-файла файл с именем, определяемым аргументом <имя файла>. Смотрите главу 3 [Написание make-файлов].
'-h' **'--help'**	Напоминает вам об опциях, которые понимает программа make, после чего заканчивает работу.
'-i' **'--ignore-errors'**	Игнорирует все ошибки в командах, выполняемых для порождения файлов заново. Смотрите раздел 5.4 [Ошибки в командах].

-I <каталог>' '--include-dir=<каталог>'	Назначает каталог, определяемый аргументом <каталог> для поиска включаемых make-файлов. Смотрите раздел 3.3 [Включение других make-файлов]. Если если используется несколько опций '-I' для определения нескольких каталогов, поиск по каталогам происходит в том порядке, в котором они перечислены.
'-j [<задания>]' '--jobs=[<задания>]'	Определяет количество заданий (команд), которые будут выполняться одновременно. При отсутствии аргумента, программа make запускает одновременно столько заданий, сколько возможно. Если определена более, чем одна опция '-j', то действует та, которая указана последней в командной строке. Смотрите раздел 5.3 [Параллельное выполнение], для дополнительной информации о том, как выполняются команды.
'-k' '--keep-going'	Продолжает работу после ошибки как можно дольше. В то время как порождение цели закончилось неудачно, и цели, зависящие от нее, не могут быть заново порождены, другие зависимости этих целей все равно могут быть обработаны. Смотрите раздел 9.6 [Проверка компиляции программ].
'-l [<загрузка>]' '--load-average[=<загрузка>]' '--max-load[=<загрузка>]'	Определяет, что ни одно новое задание (команда) не должно стартовать, если есть другие запущенные задания и средняя загрузка не меньше, чем значение, определяемое аргументом <загрузка> (число с плавающей точкой). Отсутствие аргумента отменяет действие предыдущего предела загрузки.
'-n' '--just-print' '--dry-run' '--recon'	Печатает команды, который должны выполняться, но не выполняет их. Смотрите раздел 9.3 [Вместо исполнения команд].

'-o <имя файла>' **'--old-file=<имя файла>'** **'--assume-old=<имя файла>'**	Не порождает заново файл с именем, определяемым аргументом <имя файла>, даже если он более старый, чем его зависимости, и не порождает заново ничего, что заново порождалось бы в случае изменений в этом файле. По существу, файл обрабатывается как очень старый и его правила игнорируются. Смотрите раздел 9.4 [Предотвращение перекомпиляции некоторых файлов].
'-p' **'--print-data-base'**	Выводится на экран базу данных (правила и значения переменных), которая получается в результате чтения make-файлов, а затем выполняется как обычно или каким-либо другим образом, если это определено. При использовании этой опции также выводится на экран информация о версии, которая дается с помощью опции '-v' (смотрите ниже). Чтобы вывести на экран базу данных, не пытаясь породить заново никаких файлов, используйте команду 'make -p -f /dev/null'.
'-q' **'--question'**	"Режим запроса". Не выполняет никаких команд и ничего не печатает - просто возвращает результат, который является нулем, если указанные цели уже обновлены, единицей, если требуется какое-либо новое порождение или двойкой, если обнаруживается ошибка. Смотрите раздел 9.3 [Вместо исполнения команд].
'-r' **'--no-builtin-rules'**	Отменяет использование встроенных неявных правил (смотрите главу 10 [Использование неявных правил]). Вы можете при этом определить ваши собственные неявные правила путем написания шаблонных правил (смотрите раздел 10.5 [Определение и переопределение шаблонных правил]). Опция '-r' также очищает список суффиксов, по умолчанию используемый для суффиксных правил (смотрите раздел 10.7 [Устаревшие суффиксные правила]). Но вы можете при этом, с помощью правила для цели .SUFFIXES, определить ваши собственные суффиксы и затем определить ваши собственные суффиксные правила.

'-s' **'--silent'** **'--quiet'**	Работа без сообщений - не выводит на экран команды в момент их исполнения.
'-S' **'--no-keep-going** **'--stop'**	Отменяет действие опции '-k'. Эта опция не нужна никогда, за исключением рекурсивного использования программы make, когда опция '-k' может быть унаследована от процесса make верхнего уровня через переменную MAKEFLAGS (смотрите раздел 5.6 [Рекурсивное использование программы make]), а также установки вами в вашей командной среде опции '-k' в переменную MAKEFLAGS.
'-t' **'--touch'**	Изменение времени обновления файлов (пометка их как обновленных без реального их изменения) вместо запуска соответствующих им команд. Это используется в целях создания иммитации того, что команды выполнены, для того, чтобы обмануть будущие вызовы программы make. Смотрите раздел 9.3 [Вместо исполнения команд].
'-v' **'--version'**	Выводит на экран версию программы make, а также ее copyright, список авторов и замечание об отсутствии гарантии - после этого работа завершается.
'-w' **'--print-directory'**	Выводит на экран сообщение, содержащее имя рабочего каталога как перед, так и после выполнения make-файла. Это может быть полезным для отслеживания ошибок из сложным образом вложенных рекурсивных команд make. Смотритие раздел 5.6 [Рекурсивное использование программы make]. (На практике вам редко придется определять эту опцию, так как программа make делает это за вас - смотрите раздел 5.6.4 [Опция '--print-directory']).
'--no-print-directory'	Отменяет вывод на экран имени рабочего каталога, осуществляемый с помощью опции '-w'. Этот способ полезен тогда, когда опция '-w' устанавливается автоматически, но вы не хотите видеть лишних сообщений. Смотрите раздел 5.6.4 [Опция '--print-directory'].

'-W <имя файла>' '--what-if=<имя файла>' '--new-file=<имя файла>' '--assume-new=<имя файла>'	Делает вид, что цель, определяемая аргументом <имя файла>, только что была изменена. При использовании вместе с опцией '-n', эта опция показывает вам, что бы произошло, если бы вам потребовалось изменить этот файл. Без опции '-n', эта опция почти аналогична запуску команды touch для данного файла перед запуском программы make, за исключением того, что время изменения изменяется только в представлении программы make. Смотрите раздел 9.3 [Вместо исполнения команд].
'--warn-undefined-variables'	Порождает предупреждающее сообщение всегда, когда программа make встречает ссылку на неопределенную переменную. Это может быть полезным, когда вы пытаетесь отладить make-файлы, сложными способами использующие переменные.

10. Использование неявных правил

Очень часто используются определенные стандартные способы порождения заново целевых файлов. Например, одним из типичных способов порождения объектного файла является порождение его из исходного C-файла с использованием C-компилятора, cc.

Неявные правила указывают программе make, как использовать типичные приемы, для того, чтобы чтобы вам не требовалось детально определять их тогда, когда вы хотите использовать их. Например, есть неявное правило для C-компиляции. Имена файлов определяют, какие неявные правила вступают в действие. Например, при C-компиляции обычно берется файл с именем, оканчивающемся на '.c' и порождается файл с именем, оканчивающемся на '.o'. Таким образом, программа make применяет неявное правило для C-компиляции, когда она обнаруживает эту комбинацию окончаний имен файлов.

Цепочка неявных правил может применяться последовательно - например, программа make заново породит файл с именем, оканчивающемся на '.o' из файла с именем, оканчивающемся на '.y' через файл с именем, оканчивающемся на '.c'. Смотрите раздел 10.4 [Цепочки неявных правил].

Встроенные неявные правила используют в своих командах несколько переменных, так что путем изменения значений переменных вы можете изменять способ, в соответствии с которым работает неявное правило. Например, переменная CFLAGS управляет опциями, передаваемыми C-компилятору неявными правилами для C-компиляции. Смотрите раздел 10.3 [Переменные, используемые неявными правилами].

Вы можете определить ваши собственные неявные правила с помощью написания шаблонных правил. Смотрите раздел 10.5 [Определение и переопределение шаблонных правил].

Суффиксные правила представляют собой более ограниченный способ определения неявных правил. Шаблонные правила являются более общими и ясными, но суффиксные правила оставлены для совместимости. Смотрите раздел 10.7 [Устаревшие суффиксные правила].

10.1 Использование неявных правил

Чтобы дать возможность программе make найти общий метод для обновления целевого файла, все, что вам нужно сделать - это воздержаться от самостоятельного определения команд. Либо напишите правило, не содержащее командных строк, либо вообще не пишите правило. Тогда программа make, основываясь том, какой тип исходного файла существует или может быть порожден, определит, какое неявное правило использовать.

Например, предположим, что make-файл выглядит примерно так:

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

Поскольку вы упоминаете файл 'foo.o', но не даете для него правила, программа make автоматически будет искать неявное правило, которое определяет, как его обновлять. Это происходит независимо от того, существует или нет в данный момент файл 'foo.o'.

Если неявное правило найдено, из него могут быть получены как команды, так и одна или несколько зависимостей (исходных файлов). Вам бы стоило написать для цели 'foo.o' правило без командных строк, если бы вам нужно было определить дополнительные зависимости, например, заголовочные файлы, которые не могут быть получены из неявного правила.

Каждое неявное правило имеет шаблон цели и шаблоны зависимостей. Может быть много неявных правил с одним и тем же шаблоном цели. Например, многочисленные правила порождают '.o'-файлы: одно из '.c'-файла при помощи C-компилятора, другое из '.p'-файла при помощи компилятора с языка Паскаль и т.д.. Правилом, применяемым в конкретной ситуации, является то правило, чьи зависимости существуют или могут быть порождены. Таким образом, если у вас есть файл 'foo.c', программа make запустит C-компилятор, в противном случае, если у вас есть файл 'foo.p', программа make запустит компилятор с языка Паскаль и т.д..

Конечно, когда вы пишете make-файл, вы знаете, какое неявное правило вы хотите, чтобы использовала программа make, и вы будете уверены, что она выберет именно это неявное правило, если будете знать, какие файлы зависимостей предполагаются существующими. Смотрите раздел 10.2 [Перечень неявных правил], для ознакомления с перечнем всех предопределенных неявных правил.

Выше мы сказали, что неявное правило применяется, если требуемые зависимости "существуют или могут быть порождены". Файл "может быть порожден", если он явно упоминается в make-файле в качестве цели или зависимости, или же если

рекурсивно может быть найдено неявное правило, определяющее, как его породить. Когда неявная зависимость является результатом другого неявного правила, мы говорим, что происходит образование цепочки. Смотрите раздел 10.4 [Цепочки неявных правил].

Вообще говоря, программа make ищет неявное правило для каждой цели и для каждого правила с двойным двоеточием, которые не имеют команд. Файл, который упоминается только в качестве зависимости, рассматривается как цель, чье правило ничего не определяет, поэтому для него происходит поиск неявного правила. Смотрите раздел 10.8 [Алгоритм поиска неявного правила], в котором приводятся подробности организации поиска.

Обратите внимание, что явные зависимости не оказывают влияния на поиск неявного правила. Например, рассмотрим такое явное правило:

```
foo.o: foo.p
```

Зависимость от файла 'foo.p' не обязательно означает, что программа make будет заново порождать 'foo.o' в соответствии с неявным правилом для порождения объектного файла, '.o'-файла, из исходного файла, написанного на языке Паскаль, '.p'-файла. Например, если существует также файл 'foo.c', вместо этого будет использовано неявное правило для порождения объектного файла из исходного C-файла, поскольку в списке предопределенных неявных правил (смотрите раздел 10.2 [Перечень неявных правил]) оно появляется перед правилом для языка Паскаль.

Если вы не хотите, чтобы неявное правило было использовано для цели, которая не имеет команд, вы можете установить для этой цели пустую команду, написав точку с запятой (смотрите раздел 5.8 [Определение пустых команд]).

10.2 Перечень неявных правил

Вот перечень предопределенных неявных правил, которые всегда доступны, если make-файл явным образом не перекрывает и не отменяет их. Смотрите раздел 10.5.6 [Отмена неявных правих], где приведена информация об отмене или перекрытии неявных правил. Опция '-r' или '--no-builtin-rules' отменяет все предопределенные правила.

Не все из этих правил всегда будут определены, даже при отсутствии опции '-r'. Многие из предопределенных неявных правил реализуются в программе make как суффиксные правила, поэтому какие из них будут определены, зависит от списка суффиксов (списка зависимостей специальной цели .SUFFIXES). По умолчанию список суффиксов
такой : .out, .a, .ln, .o, .c, .cc, .C, .p, .f, .F, .r, .y, .l, .s, .S, .mod, .sym, .def, .h, .info, .dvi, .te
x, .texinfo, .texi, .txinfo, .w, .ch, .web, .sh, .elc, .el. Все неявные правила из описанных ниже, чьи зависимости имеют один из этих суффиксов, являются на самом деле суффиксными правилами. Если вы изменяете список суффиксов, то действовать будут только те предопределенные суффиксные правила, имена которых состтоят из одного или двух суффиксов, которые указаны в определенном вами списке - действие правила, чьи суффиксы отсутствуют в списке, отменяется. Смотрите раздел 10.7 [Устаревшие суффиксные правила], где где во всех деталях описаны суффиксные правила.

Компиляция C-программ

'n.o' автоматически порождается из 'n.c' при помощи команды в форме '$(CC) -c $(CPPFLAGS) $(CFLAGS)'.

Компиляция программ на языке C++

'n.o' автоматически порождается из 'n.cc' или 'n.C' при помощи команды в форме '$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)'. Мы рекомендуем вам использовать для исходных файлов, написанных на языке C++, вместо суффикса '.C' суффикс '.cc'.

Компиляция программ на языке Паскаль

'n.o' автоматически порождается из 'n.c' при помощи команды '$(PC) -c $(PFLAGS)'.

Компиляция программ на языках Фортран и Ратфор

'n.o' автоматически порождается из 'n.r', 'n.F' or 'n.f' путем запуска компилятора с языка Фортран. Точный вид у используемой команды следующий:

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

Обработка программ на языках Фортран и Ратфор при помощи препроцессора

'n.f' автоматически порождается из 'n.r' or 'n.F'. Это правило просто запускает препроцессор для преобразования программы на языке Ратфор, представляющей собой подлежащую обработке препроцессором программу на языке Фортран, в программу на языке Фортран в чистом виде. Точный вид у используемой команды следующий:

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

Компиляция программ на языке Модула-2

'n.sym' порождается из 'n.def' при помощи команд в форме '$(M2C) $(M2FLAGS) $(DEFFLAGS)'. 'n.o' порождается 'n.mod', форма команды такая: '$(M2C) $(M2FLAGS) $(MODFLAGS)'.

Ассемблирование и обработка препроцессором программ на языке Ассемблера

'n.o' автоматически порождается из 'n.s' путем запуска ассемблера, программы as. Точный вид команды такой: '$(AS) $(ASFLAGS)'.

'n.s' автоматически порождается из 'n.S' путем запуска препроцессора языка C, программы cpp. Точный вид команды такой: '$(CPP) $(CPPFLAGS)'.

Компоновка одиночного объектного файла

'n' автоматически порождается из 'n.o' путем запуска компоновщика (обычно он называется ld) через C-компилятор. Точный вид у используемой команды такой: '$(CC) $(LDFLAGS) N.o $(LOADLIBES)'.

Это правило правильно работает для простой программы только с одним исходным файлом. Оно также будет правильно работать, если есть несколько объектных файлов (предположительно, получаемых из каких-то дополнительных исходных файлов), один из которых имеет имя, соответствующее имени исполняемого файла. Таким образом, следующее правило:

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

если существуют все из файлов 'x.c', 'y.c' и 'z.c' приведет к исполнению такой последовательности 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 случаях, например, при отсутствии объектного файла, имя которого получается из имени исполняемого файла, вы должны написать явную команду для компоновки.

Файл любого вида, для которого предусмотрено автоматическое порождение объектного файла с именем, оканчивающимся на '.o', будет скомпонован с использованием компилятора ('$(CC)', '$(FC)' или '$(PC)', C-компилятор '$(CC)' используется для ассемблерования '.s'-файлов) без опции -c. Это может быть сделано с использованием объектного '.o'-файла в качестве промежуточного, но быстрее сделать компиляцию и компоновку за один шаг, поэтому именно так и делается.

Получение C-программ при помощи программы Yacc

'n.c' автоматически порождается из 'n.y' путем запуска программы Yacc при помощи команды '$(YACC) $(YFLAGS)'.

Получение C-программ при помощи программы Lex

'n.c' автоматически порождается из 'n.l' путем запуска программы Lex. При этом используется такая команда: '$(LEX) $(LFLAGS)'.

Получение программ на языке Ратфор при помощи программы Lex

'n.r' автоматически порождается из 'n.l' путем запуска программы Lex. При этом используется такая команда: '$(LEX) $(LFLAGS)'.

Соглашение об использовании одного и того же суффикса '.l' для всех Lex-файлов, порождают ли они код на языке C или на языке Ратфор делает для программы make невозможным автоматическое определение, какой из двух языков вы используете в каждом конкретном случае. Если программа make вызывается для порождения заново объектного файла из '.l'-файла, она должна выбрать, какой компилятор использовать. Она выберет C-компилятор, поскольку он более популярен. Если вы используете язык Ратфор, сделайте так, чтобы программа make знала об этом, упомянув в make-файле 'n.r'. Или же, если вы используете только Ратфор, без C-файлов, удалите '.c' из списка суффиксов неявных правил, как показано ниже:

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

Порождение lint-библиотек из программ на C, Lex и Yacc

'n.ln' порождается из 'n.c' путем запуска программы lint. Точный вид команды такой : '$(LINT) $(LINTFLAGS) $(CPPFLAGS) -i'. Такая же команда используется для работы с C-кодом, полученным из 'n.y' или 'n.l'.

TeX-файлы и Web-файлы

'n.dvi' порождается из 'n.tex' при помощи команды '$(TEX)'. 'n.tex' порождается из 'n.web' при помощи '$(WEAVE)' или из 'n.w' (и из 'n.ch', если он существует или может быть порожден) при помощи '$(CWEAVE)'. 'n.p' порождается из 'n.web' при помощи '$(TANGLE)', а 'n.c' порождается из 'n.w' (и из 'n.ch', если он существует или может быть порожден) при помощи '$(CTANGLE)'.

Texinfo-файлы и Info-файлы

'n.dvi' порождается из 'n.texinfo', 'n.texi' или 'n.txinfo' при помощи команды '$(TEXI2DVI) $(TEXI2DVI_FLAGS)'. 'n.info' порождается из 'n.texinfo', 'n.texi' или 'n.txinfo' при помощи команды '$(MAKEINFO) $(MAKEINFO_FLAGS)'.

RCS-файлы

Любой файл 'n' извлекается, если это необходимо, из RCS-файла с именем либо 'n,v' or 'RCS/n,v'. Точный вид используемой команды такой: '$(CO) $(COFLAGS)'. Если файл 'n' уже существует, то он не будет извлекаться из RCS-файла, даже если RCS-файл является более новым. Правила для RCS-файлов являются терминальными (смотрите раздел 10.5.5 [Шаблонные правила с произвольным соответствием]), поэтому RCS-файлы не могут быть сгенерированы из какого-нибудь еще источника - они должны реально существовать.

SCCS

Любой файл 'n' извлекается, если это необходимо, из SCCS-файла с именем либо 's.n' or 'SCCS/s.n'. Точный вид используемой команды такой: '$(GET) $(GFLAGS)'. Правила для SCCS-файлов являются терминальными (смотрите раздел 10.5.5 [Шаблонные правила с произвольным соответствием]), поэтому SCCS-файлы не могут быть сгенерированы из какого-нибудь еще источника - они должны реально существовать.

Для улучшения использования SCCS, файл 'n' копируется из 'n.sh', и затем делается исполняемым (для всех). Это предназначено для командных файлов, с которыми работает программа SCCS. Так как программа RSC сохраняет права на исполнение файла, вам не требуется использовать эту особенность при работе с RSC.

Мы рекомендуем вам избегать избегать использования програмы SCCS. Программа RCS прочно удерживает ведущее положение и, кроме того, является свободной. Выбирая свободное программное обеспечение вместо аналогичного (или худшего) программного обеспечения, являющегося чьей-либо собственностью, вы поддерживаете развитие свободного программного обеспечения.

Обычно вы хотите изменять только упомянутые в вышеприведенном списке переменные, которые документированы в следующем разделе.

Однако, команды во встроенных неявных правилах используют, на самом деле, такие переменные, как COMPILE.c, LINK.p и PREPROCESS.S, чьи значения содержат приведенные выше команды.

Программа make следует соглашению, согласно которому правило для компиляции исходного '.x'-файла использует переменную COMPILE.x. Аналогично, правило для порождения исполняемого файла из '.x'-файла использует переменную LINK.x, а правило для обработки '.x'-файла препроцессором использует переменную PREPROCESS.x.

Каждое правило, которое порождает объектный файл, использует переменную OUTPUT_OPTION. Программа в make определяет эту переменную либо как содержащую значение '-o $@ ', либо как пустую, в зависимости от требуемой в период компиляции опции. Опция '-o' нужна вам для того, чтобы гарантировать, что выход идет в правильный файл, когда исходный файл находится в другом каталоге, как это бывает при использовании переменной VPATH (смотрите раздел 4.3 [Поиск по каталогам]). Однако, компиляторы на некоторых системах не воспринимают опцию '-o' для объектных файлов. Если вы используете такую систему и используете переменную VPATH, в некоторых сеансах компиляции выход будет помещен в не туда, куда надо. Возможный обход этой проблемы заключается в присваивании переменной OUTPUT_OPTION значения '; mv $*.o $@ '.

10.3 Переменные, используемые неявными правилами

Команды во встроенных неявных правилах обильно используют некоторое переменные с предопределенными значениями. Вы можете изменять эти переменные в make-файле, при помощи аргументов программы make или же в командной среде для то, чтобы изменить работу неявных правил, не переопределяя самих правил.

Например, команда, используемая для компиляции исходного C-файла в действительности выглядит так: '$(CC) -c $(CFLAGS) $(CPPFLAGS)'. По умолчанию значение перененной CC является 'cc', значениями остальных используемых здесь переменных является пустое значение, в результате чего получается команда 'cc -c'. С помощью переопределения переменной CC в значение 'ncc' вы могли бы сделать так, чтобы для всех C-компиляций, выполняемых в результате работы неявных правил, мспользовался компилятор 'ncc'. С помощью переопределения переменной CFLAGS в значение '-g', вы могли бы передать в каждом сеансе компиляции опцию '-g'. Все неявные правила, выполняющие C-компиляцию, используют ссылку на переменную '$(CC)' для определения имени программы-компилятора, и все они среди аргументов , передаваемых компилятору, содержат ссылку на переменную '$(CFLAGS)'.

Переменные, используемые в неявных правилах, разделяются на два класса: переменные, являющиеся именами программ (такие, как CC) и переменные, содержащие аргументы для программы (такие, как CFLAGS). ("Имя программы" может также содержать некоторые аргументы командной строки, но оно должно начинаться с имени реальной исполняемой программы.) Если значение переменной содержит более, чем один аргумент, разделяйте их при помощи пробелов.

Вот список переменных, используемых в качестве имен программ во встроенных правилах:

AR	Поддерживающая архивы программа, по умолчанию 'ar'.
AS	Программа для выполнения ассемблирования, по умолчанию 'as'.
CC	Программа для компиляции C-программ, по умолчанию 'cc'.
CXX	Программа для компиляции программ на языке C++, по умолчанию 'g++'.
CO	Программа для извлечения файлов из RCS, по умолчанию 'co'.
CPP	Программа для запуска C-препроцессора, с выдачей результатов на стандартный выход, по умолчанию '$(CC) -E'.
FC	Программа для компиляции или обработки препроцессором программ на языке Фортран или Ратфор, по умолчанию 'f77'.
GET	Программа для извлечения файлов из SCCS, по умолчанию 'get'.
LEX	Программа, используемая для преобразования Lex-грамматик в программы на языке C или Ратфор, по умолчанию 'lex'.
PC	Программа для компиляции программ на языке Паскаль, по умолчанию 'pc'.
YACC	Программа, используемая для преобразования Yacc-грамматик в C-программы, по умолчанию 'yacc'.
YACCR	Программа, используемая для преобразования Yacc-грамматик в программы на языке Ратфор, по умолчанию 'yacc -r'.
MAKEINFO	Программа для проеобразования исходного Texinfo-файла в Info-файл, по умолчанию 'makeinfo'.

TEX	Программа для создания dvi-файлов из исходных Tex-файлов, по умолчанию 'tex'.
TEXI2DVI	Программа для создания dvi-файлов из исходных Texinfo-файлов, по умолчанию 'texi2dvi'.
WEAVE	Программа для перевода из Web-формата в TeX-формат, по умолчанию 'weave'.
CWEAVE	Программа для перевода из C-Web-формата в TeX-формат, по умолчанию 'cweave'.
TANGLE	Программа для перевода из Web-формата в программу на языке Паскаль, по умолчанию 'tangle'.
CTANGLE	Программа для перевода из Web-формата в C-программу, по умолчанию 'tangle'.
RM	Команда для удаления файла, по умолчанию 'rm -f'.

Вот список переменных, значения которых являются дополнительными аргументами для приведенных выше программ. По умолчанию значениями всех из них является пустая строка, если не указано другого.

ARFLAGS	Опции, передаваемые поддердивающей архивы программе, по умолчанию 'rv'.
ASFLAGS	Опции, передаваемые поддердивающей архивы программе, по умолчанию 'rv'.
CFLAGS	Дополнительные опции, передаваемые C-компилятору.
CXXFLAGS	Дополнительные опции, передаваемые компилятору с языка C++.
COFLAGS	Дополнительные опции, передаваемые программе co, входящей в систему RCS.
CPPFLAGS	Дополнительные опции, передаваемые C-препроцессору и программам, которые его используют (компиляторам с языков C и Фортран).
FFLAGS	Дополнительные опции, передаваемые компилятору с языка Фортран.
GFLAGS	Дополнительные опции, передаваемые программе get, входящей в систему SCCS.
LDFLAGS	Дополнительные параметры, передаваемые компиляторам, когда преполагается вызов из них компоновщика, 'ld'.
LFLAGS	Дополнительные опции, передаваемые программе Lex.
PFLAGS	Дополнительные опции, передаваемые компилятору с языка Паскаль.

RFLAGS	Дополнительные опции, передаваемые компилятору с языка Фортран для использования с программами на языке Ратфор.
YFLAGS	Дополнительные опции, передаваемые программе Yacc.

10.4 Цепочки неявных правил

Иногда файл может быть порожден с помощью последовательности неявных правил. Например, файл 'n.o' мог бы быть порожден из 'n.y' запуском сначала программы Yacc, а затем cc. Такая последовательность называется цепочкой.

Если файл 'n.c' существует или упоминается в make-файле, то не требуется никакого специального поиска: программа make узнает, что объектный файл может быть порожден при помощи C-компиляции из 'n.c', затем, при определении того, как породить файл 'n.c', используется правило для запуска программы Yacc. В конечном счете, как 'n.c', так и 'n.o' становятся обновленными.

Однако, даже если файл 'n.c' не существует и не упоминается в make-файле, программа make способна разглядеть в нем отсутствующую связь между файлами 'N.o' и 'N.y'! В этом случае 'n.c' называется "промежуточным файлом". Как только программа make решила использовать промежуточный файл, он вводится в базу данных, как если бы он был упомянут в make-файле, вместе с неявным правилом, которое указывает, как его создавать.

Порождение заново промежуточных файлов происходит с использованием соответствующих им правил точно также как и всех других файлов. Различие заключается в том, что при завершении программы make промежуточный файл удаляется. Следовательно, промежуточный файл, который не существовал перед началом работы программы make, также не существует и после окончания ее работы. Об удалении вам сообщается при помощи вывода на экран команды 'rm -f', которая показывает, что делает программа make. (Вы можете указать шаблон цели неявного правила (например, '%.o') в качестве зависимости специальной цели '.PRECIOUS', чтобы предохранить от удаления промежуточные файлы, порождаемые неявными правилами с шаблонами цели, соответствующими указанному шаблону; смотрите раздел 5.5 [Прерывания].)

Цепочка может включать в себя более, чем два неявных правила. Например, в возможно порождение файла 'foo' из файла 'RCS/foo.y,v' посредством запуска программ RCS, Yacc и cc. В таком случае файлы 'foo.y' и 'foo.c', как один, так и другой, являются промежуточными файлами, которые в конечном счете удаляются.

Никакое отдельное неявное правило не может появляться в цепочке неоднократно. Это означает, что программа make даже не станет рассматривать такую нелепость, как порождение файла 'foo' из файла 'foo.o.o' с помощью выполнения компоновщика дважды. Это ограничение несет с собой дополнительную пользу, заключающуюся в предотвращении возникновения какого-либо бесконечного цикла во время поиска цепочки неявных правил.

Существуют некоторые специальные неявные правила, предназначенные для оптимизации работы в некоторых ситуациях, которые в противном случае обрабатывались бы с помощью цепочек правил. Например, порождение файла 'foo' из файла 'foo.c' могло бы обрабатываться с помощью компиляции и компоновки,

выполняющихся в рамках отдельных связанных в цепочку правил, с использованием 'foo.c' в качестве промежуточного файла. Но на самом деле суть происходящего заключается в том, что специальное правило для этого случая выполняет компиляцию и компоновку при помощи одной команды cc. Использование оптимизированного правила дает преимущество перед использованием пошаговой цепочки поскольку нем ускоряется выполнение правил.

10.5 Определение и переопределение шаблонных правил

Вы определяете неявное правило с помощью записи шаблонного правила. Шаблонное правило похоже на обычное правило, за исключением того, что его цель содержит символ '%' (ровно один). Цель рассматривается как шаблон для сопоставления с ним имен файлов - символу '%' может соответствовать любая непустая подстрока, в то время как любому другому символу соответствует только он сам. Зависимости также используют символ '%', чтобы показать как их имена соотносятся с именем цели.

Таким образом, шаблонное правило '%.o : %.c' указывает программе make, как на породить любой файл с именем вида 'stem.o' из другого файла, с именем вида 'stem.c'.

Обратите внимание, что в шаблонных правилах подстановка значения, соответствующего символу '%' происходит после подстановки значений всех переменных и функций, что имеет место при чтении make-файла. Смотрите главу 8 [Функции преобразования текста].

Введение в шаблонные правила

Шаблонное правило содержит в цели символ '%' (ровно один), в остальном он выглядит точно также, как и обычное правило. Цель представляет собой шаблон, для сопоставления с именами файлов, символ '%' сопоставляется с любой непустой подстрокой, в то время как любые другие символы сопоставляются только сами себе.

Например, если рассматривать в качестве шаблона '%.c', то с ним сопоставляется любое имя файла, которое заканчивается в '.c'. Если рассматривать в качестве шаблона 's.%.c', то с ним сопоставляется любое имя файла, которое начинается на 's.', заканчивается в '.c' и занимает, по меньшей иере, пять символов в длину. (В подстроке должен быть, по крайней мере, один символом, для того, чтобы ее можно было сопоставить символу '%'.) Подстрока, которая сопоставляется символу '%', называется основой.

Символ '%' в зависимости шаблонного правила употребляется для обозначения той же самой основы, которая была сопоставлена символу '%' в цели. Для применения шаблонного правила, его шаблону цели должно быть сопоставлено обрабатываемое имя файла, в результате чего его шаблоны зависимостей должны превратиться в имена файлов, которые существуют или могут быть порождены. Эти файлы становятся зависимостями цели.

Таким образом, правило в такого вида:

```
%.o : %.c ; КОМАНДА...
```

определяет, как породить файл 'n.o' с использованием другого файла, 'n.c', в качестве его зависимости, в предположении, что 'n.c' существует или может быть порожден.

В правиле могут также быть зависимости, в которых не содержитсяч символ '%' - такие зависимости присоединяются к любому файлу, порождаемому этим шаблонным правилом.

Там может также быть зависимостями, которые не используют '%' - такая зависимость присоединяет к каждому файлу, делаемому этим правилом образца. Такие неизменяемые зависимости время от времени являются полезными.

Шаблонному правилу необязательно иметь какие-либо зависимости, которые содержат символ '%', более того, фактически, необязательно иметь какие-либо зависимости вообще. Такое правило, по сути, является общим шаблоном. Оно обеспечивает способ для порождения любого файла, который сопоставляется шаблону цели. Смотрите раздел 10.6 [Определение правил последней возможности, используемых по умолчанию].

Шаблонные правила могут иметь более, чем одну цель. В отличие от обычных правил, это не означает несколько разных правил с одними и теми же зависимостями и командами. Если шаблонное правило имеет несколько целей, то программа make определяет на основании этого, что команды правила отвечают за порождение всех целей. Для порождения всех целей команды выполняются только один раз. При поиске шаблонного правила для сопоставления с целью, шаблоны цели правила, отличные от того, с которым сопоставляется цель, для которой требуется правило, являются лишними - программа make заботится только о том, чтобы определить команды и зависимости рассматриваемого в данный момент файла. Однако, когда выполняются команды, соответствующие этому файлу, другие цели помечаются как обновившиеся сами.

Порядок в котором шаблонные правила появляются в make-файле существеннен, так как рассматриваются они в этом порядке. Из двух в равной степени применимых правил, используется только первое, найденное в тексте. Правила, которые пишете вы имеют приоритет перед встроенными. Тем не менее, имейте в виду, что правило, зависимости которого действительно существуют или упоминаются в make-файле, всегда имеют приоритет перед правилом с зависимостями, которые должны быть порождены с помощью построения цепочки других неявных правил.

Примеры шаблонных правил

Вот некоторые примеры шаблонных правил, на самом деле являющихся предопределенными программой make. Сначала рассмотрим правило, которое компилирует '.c'-файлы в '.o'-файлы:

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

Приведенный фрагмент make-файла определяет правило, которое может породить любой файл 'x.o' из файла 'x.c'. Команда использует автоматические переменные '$@ ' и '$<', которые на место которых каждый раз при применении правила подставляются, соответственно, имена целевого и исходного файлов.

Вот второе встроенное правило:

```
% :: RCS/%,v
        $(CO) $(СОФЛАГИ) $<
```

Здесь определяется правило, которое может породить произвольный файл 'x' из соответствующего файла 'x,v' в подкаталоге 'RCS'. Поскольку целью является '%', это правило применится к любому файлу, при условии, что соответствующий файл зависимости существует. Двойное двоеточие делает правило терминальным, и это означает, что его зависимость не может быть промежуточным файлом (смотрите раздел 10.5.5 [Шаблонные правила с произвольным соответствием]).

Следующее шаблонное правило имеет две цели:

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

Это указывает программе make на то, что команда 'bison -d X.y' будет порождать как файл 'x.tab.c' так и файл 'x.tab.h'. 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 файла 'parse.y' изменяется, команда 'bison -d parse.y' будет выполнена только один раз, в результате чего будут обновлены зависимости, как файла 'parse.tab.o', так и файла 'scan.o'. (Предполагается, файл 'parse.tab.o' будет перекомпилироваться из файла 'parse.tab', а файл'scan.o' - из 'scan.c', в то время как файл 'foo' компонуется из файлов 'parse.tab.o', 'scan.o', и других его зависимостей, и после этого он будет всегда счастливо выполняться.)

Автоматические переменные

Предположим, что вы записываете шаблонное правило для скомпиляции '.c'-файла в '.o'-файл: как вы напишите команду 'cc', чтобы она работала с правильным именем исходного файла? Вы не можете записать имя в команде, поскольку это имя разное при каждом использовании неявного правила.

То, что вам следует делать - это использовать специальную возможность программы make, автоматические переменные. Эти переменные имеют значения, заново вычисленные для каждого выполняемого правила на основе цели и зависимостей правила. В данном примере, вы бы использовали переменную '$@ ' для имени объектного файла и переменную '$<' для имени исходного файла.

Вот список автоматических переменных:

`$@'	Имя файла цели правила. Если целью является элемент архива, то '$@' - имя файла архива. В шаблонном правиле с несколькими целями (смотрите раздел 10.5.1 [Введение в шаблонные правила]), '$@' - имя той цели, которая вызвала выполнение команд правила.
`$%'	Имя элемента цели, в том случае, когда цель - элемент архива. Смотрите главу 11 [Элементы архива в качестве целей]. Например, если целью является - 'foo.a(bar.o)', то значение переменной '$%' - 'bar.o', а переменной '$@ ' - 'foo.a'. Переменная '$%' имеет пустое значение, когда цель не -элемент архива.

`$<'	Имя первой зависимости. Если цель получила свои команды из неявного правила, то это будет первая зависимость, добавленная неявным правилом (смотрите главу 10 [Использование неявных правил].).
`$?'	Имена всех зависимостей, которые являются более новыми, чем цель, с пробелами между ними. Для зависимостей, которые являются элементами архива, используются только имя элемента (смотрите главу 11 [Архивы].).
`$^'	Имена всех зависимостей, с пробелами между ими. Для зависимостей, которые являются элементами архива, используются только имя элемента (смотрите главу 11 [Архивы].). Цель имеет только одну зависимость для каждого файла, от которого он зависит от, независимо от того, сколько раз каждый файл указан в качестве зависимости. Таким образом, если вы для цели неоднократно укажете одну и ту же зависимость, значение переменной '$^' будет содержать только одну копию ее имени.
`$+'	Эта переменная аналогична переменной '$^', только зависимости, указанные неоднократно дублируются в том в порядке, в котором они указаны в make-файле. Это в первую очередь полезно для для использования в командах компоновки, где является существенным повторение имен библиотек в определенном порядке.
`$*'	Основа с которой сопоставляется неявное правило (смотрите раздел 10.5.4 [Как определяется соответствие шаблону]). Если целью является файл 'dir/a.foo.b', а шаблон цели - 'a.%.b', то основой будет 'dir/foo'. Основа полезна для создания имен файлов, связанных с правилом. В статическом шаблонном правиле основа представляет собой часть имени файла, которая сопоставляется символу '%' в шаблоне цели. В явном правиле основы нет, поэтому в этом случае переменная '$*' не может иметь определенного значения. Вместо этого, если имя цели оканчивается распознаваемым суффиксом (смотрите раздел 10.7 [Устаревшие суффиксные правила]), то в качестве значения переменной '$*' устанавливается имя цели без суффикса. Например, если имя цели - 'foo.c', то в качестве значения переменной '$*' устанавливается 'foo, так как '.c' - суффикс. GNU-версия программы make делать эту причудливую вещь только для совместимости с другими реализациями программы make. Вам, вообще говоря, следует избегать использования переменной '$*', за исключением неявных правил или статических шаблоных правил. Если имя цели в явном правиле не оканчивается распознаваемым суффиксом, то для этого правила в качестве значения переменной '$*' устанавливается пустая строка.

Переменная '$?' полезна даже в явных правилах, когда вы хотите работать только с зависимостями, которые были изменены. Например, предположим, что архив с именем 'lib' предназначен для хранения копий нескольких объектных файлов.

Приведенное ниже правило копирует в архив только измененные объектные файлы:

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

Из переменных, перечисленных выше, четыре имеют значения, являющиеся одиночными именами файлов, а две имеют значения, которые представляют собой списки имен файлов. Эти шесть переменных имеют возможности получить а качестве значения только имя каталога из полного имени файла или только само имя файла в пределах каталога. Вариант имени переменной определяется путем добавления, соответственно, 'D' или 'F'. Эти варианты в GNU-версии программы make являются полуустаревшими, поскольку для получения аналогичного эффекта могут использоваться функции dir и notdir (смотрите раздел 8.3 [Функции для обработки имен файлов]). Тем не менее, обратите внимание, что при использовании всех вариантов с буквой 'D', конечный символ '/', который всегда появляется в качестве результата функции dir , опускается.

Ниже приводится список вариантов:

'$(@D)'	Часть имени файла цели, определяющая каталог, с удаленным конечным символом '/'. Если значением переменной '$@' является 'dir/foo.o', то значением переменной '$(@D)' является 'dir'. Этим значением является '.', если значение переменной '$@' не содержит символа '/'.
$(@F)'	Часть имени файла цели, определяющая файл внутри каталога. Если значением переменной '$@' является 'dir/foo.o', то значением переменной '$(@F)' является 'foo.c'. Переменная '$(@F)' эквивалентна вызову функции '$(@)'.
'$(*D)'** **'$(*F)'	Часть основы, определяющая, соответственно, каталог и файл внутри каталога - 'dir' и 'foo' в этом примере.
'$(%D)'** **'$(%F)'	Часть имени целевого элемента архива, определяющая, соответственно, каталог и файл внутри каталога. Это имеет смысл только для целей, являющихся элементами архива, и имеющих форму 'АРХИВ(ЭЛЕМЕНТ)', и полезно только когда ЭЛЕМЕНТ может содержать имя каталога. (Смотрите раздел 11.1 [Элементы архива в качестве целей].)
'$(<D)'** **'$(<F)'	Часть имени первой зависимости, определяющая, соответственно, каталог и файл внутри каталога.
'$(^D)'** **'$(^F)'	Списки частей имен всех зависимостей, определяющих, соответственно, каталоги и файлы внутри каталогов.
'$(?D)'** **'$(?F)'	Списки частей имен всех зависимостей, которые являются более новыми, чем цель, определяющих, соответственно, каталоги и файлы внутри каталогов.

Обратите внимание, что мы, когда говорим об этих автоматических переменных, используем специальные стилистические соглашения. Мы пишем "значение переменной '$<'", а не "переменная '<'", как мы написали бы для обычных

переменных, таких, как objects и CFLAGS. Мы думаем, в данном случае это соглашение выглядит более естественным. Не стоит полагать, что это имеет глубокое значение - ссылка '$<' ссылается на переменную с именем '<', точно так же, как ссылка '$(CFLAGS)' ссылается на переменную с именем 'CFLAGS'. Вы могли бы точно так же использовать запись '$(<)' вместо `$<'.

Как определяется соответствие шаблону

Шаблон цели состоит из символа '%', заключенного между префиксом и суффиксом, один из которых или оба сразу могут быть пустыми. Имя файла сопоставимо с шаблоном только в том случае, если оно начинается с указанного префикса и заканчивается указанным суффиксом, без перекрытия. Фрагмент текста, расположенный между префиксом и суффиксом, называется основой. Таким образом, если образцу '%.o' сопоставляется имя файла 'test.o', то основой является 'test'. Зависимости шаблонного правила преобразуются в реальные имена файлов путем замены символа '%' на основу. Таким образом, если в этом же примере одна из зависимостей записана в виде '%.c', она преобразуется в 'test.c'.

Когда шаблон цели не содержит символа '/' (а обычно он его не содержит), из имени файла убирется часть, определяющая имена каталогов, прежде, чем это имя будет сравниваться с префиксом и суффиксом цели. После сравнения имени файла с шаблоном цели, имена каталогов, вместе с символом '/', которым они заканчиваются, добавляются к именам файлов зависимостей, сгенерированных из шаблонов зависимости шаблонного правила и к имени целевого файла. Директории игнорируются только для того, чтобы обнаружить неявное правило, которое надо использовать, а не при применении этого правила. Таким образом, шаблону 'e%t' сопоставляется имя файла 'src/eat', с основой 'src/a'. Когда зависимости преобразуются в имена файлов, часть основы, определябщая катологи, добавляется в их начало, в то время как оставшаяся часть основы подставляется на место символа '%'. Основа 'src/a' в применении к шаблону зависимости 'c%r' дает имя файла 'src/car'.

Шаблонные правила с произвольным соответствием

Когда целью шаблонного правила является просто символ '%', то с ним сопоставляется произволное имя файла. Мы вызываем такие правила правилами с произвольным соответствием. Они очень полезны, но их обработка может занять у программы make много времени, поскольку она должна обработать каждое такое правило для каждого имени файла, указанного или как цель, или как зависимость.

Допустим, в make-файле упоминается файл 'foo.c'. Для обработки этой цели, программа make должна рассмотреть получение его при помощи компоновки объектного файла 'foo.c.o', или же, при помощи C-компиляции и компоновки, выполняемых за один шаг, из файла 'foo.c.c', или же, при помощи компиляции и компоновки программы на языке Паскаль, из файла 'foo.c.p', а также много других возможностей.

Мы знаем, что эти возможности являются нелепыми, поскольку 'foo.c' представляет собой не исполняемый, исходный C-файл. Если бы программа make рассматривала эти возможности, она в конце концов отвергла бы их, поскольку такие файлы, как 'foo.c.o' и 'foo.c.p' не существовали бы. Но эти возможности

настолько многочисленны, что программа make работала бы очень медленно, если бы ей требовалось их рассматривать.

Для того, чтобы ускорения работы, мы установили различные ограничения на то, как программа make рассматривает правила с произвольным соответствием. Есть два разных ограничения, которые могут быть применены, и всякий раз, когда вы определяете правило с произвольным соответствием, вы должны выбрать для этого правила одно из них.

Один вариант -, пометить правило с произвольным соответствием как терминальное, используя при его определении двойное двоеточие. Если правило является терминальным, оно не применяется в том случае, когда его зависимости реально не существуют. Зависимости, которые могли быть порождены при помощи других неявных правил не являются достаточно хорошими. Другими словами, после терминального правила не допустимо никакого дальнейшего продолжения цепочки правил.

Например, встроенные неявные правила для извлечения исходных файллв из RCS-файлов и SCCS-файлов являются терминальными - в результате, если файл 'foo.c,v' не существует, программа make даже не рассмотрит попытку породить его, как промежуточный файл, из файла 'foo.c,v.o' или из файла 'RCS/SCCS/s.foo.c,v'. RCS-файлы и SCCS-файлы обычно являются окончательными исходными файлами, которые не должны быть заново порожденными из любых других файлов, следовательно, программа make может сберечь время, не занимаясь поисками пути для того, чтобы заново породить их.

Если вы не помечаете правило с произвольным соответствием как терминальное, то оно является нетерминальным. Нетерминальное правило с произвольным соответствием не может быть применено к имени файла, которое указывает на специфический тип данных. Имя файла указывает на специфический тип данных если оно соответствует некоторой цели неявного правила, не являющегося правилом с произвольным соответствием.

Например, имя файла 'foo.c' сопоставляется с целью шаблонного правила '%.c : %.y' (правило для выполнения программы Yacc). Независимо от того, применимо ли на самом деле это правило (что имеет место только в том случае, когда есть файл 'foo.y'), того факта, что имеется соответствие с его целью, достаточно для того, чтобы предотвратить рассмотрение любых нетерминальных правил с произвольным соответствием для файла 'foo.c'. Таким образом, программа make даже не рассмотрит попытку породить файл 'foo.c', как исполняемый файл, из файлов 'foo.c.o', 'foo.c.c', 'foo.c.p', и т.п..

Мотивация этого ограничения заключается в том, что нетерминальные правила с произвольным соответствием используются для порождения файлов, содержащих особые типы данных (например, выполняемые файлы), а имя файла с распознаваемым суффиксом указывает на некоторый другой специфический тип данных (например, исходный C-файл).

Исключительно для распознавание определенных имен файлов, с тем, чтобы для них не рассматривались нетерминальные правила с произволным соответствием, введены специальные встроенные шаблонные правила-заглушки. Эти правила-заглушки не имеют никаких зависимостей и никаких команд, и они игнорируются во всех остальных случаях.

Например, встроенное неявное правило

%.p:

существует для того, чтобы гарантировать, что исходные файлы на языке Паскаль, как, например, 'foo.p', сопоставляются с определенным шаблоном цели и, тем самым, предотвратить трату времени на поиск файла 'foo.p.o' или файла 'foo.p.c'.

Шаблонные правила-заглушки, такие как правило для '%.p' порождаются для всех суффиксов, указанных как имеющие силу для использования в суффиксных правилах (смотрите раздел 10.7 [Устаревшие суффиксные правила]).

Отмена действия неявных правил

Вы можете перекрыть встроенное неявное правило (или то, которое вы сами определили) с помощью определения нового шаблонного правила с такими же целью и зависимостями, но с другими командами. Если определено новое правило, то встроенное заменяется. Положение нового правила в последовательности неявных правил определяется тем, где вы пишете новое правило.

Вы можете отменить действие встроенного неявного правила, определив шаблонное правило с той же самой целью, но без команд. Например, следующий фрагмент make-файла отменит правило, которое запускает ассемблер:

%.o : %.s

10.6 Определение правил последней возможности, используемых по умолчанию

Вы можете определить неявное правило последней возможности с помощью написания терминального шаблонного правила с произвольным соответствием без зависимостей (смотрите раздел 10.5.5 [Шаблонные правила с произвольным соответствием]). Оно во всем аналогично любому другому шаблонному правилу, единственное, что специфично для него - это то, что с ним будет сопоставляться любая цель. Таким образом, команды такого правила используются для всех целей и зависимостей, которые не имеют своих собственных команд и к которым неприменимо ни одно другое неявное правило.

Например, при тестировании make-файла, вы могли бы заботиться не о том, содержат ли исходные файлы реальные данные, а только о том, что они существуют. В таком случае вы могли бы сделать следующее:

```
%::
        touch $@
```

что привело бы к тому, что все требуемые (как зависимости) файлы были созданы автоматически.

Кроме того, вы можете определить команды, которые будут использованы для целей, у которых совсем нет правил, даже тех, которые не определяют команды. Это делается при помощи правила для цели .DEFAULT. Команды этого правила используются для всех зависимостей, которые не появляются в роли целей ни в одном явном правиле, и для которых неприменимо ни одно неявное правило. Естественно, правила для цели .DEFAULT не будет, если вы не его не напишете.

Если вы используете .DEFAULT без команд и зависимостей, как показано ниже:

```
.DEFAULT:
```

то команды, ранее запомненные как соответствующие цели .DEFAULT , очищаются. После этого программа make действует так, как будто вы совсем никогда не определяли правило для цели .DEFAULT.

Если вы не хотите, чтобы цель получала команды из шаблонного правила с произвольным соответствием и из правила для цели .DEFAULT, но при этом вы не хотите, чтобы для этой цели запускалась какая-либо команда, вы можете определить для нее пустую команду (смотрите раздел 5.8 [Определение пустых команд]).

Вы можете использовать правило последней возможности для перекрытия части другого make-файла. Смотрите раздел 3.6 [Перекрытие части другого make-файла].

10.7 Устаревшие суффиксные правила

Суффиксные правила представляют собой устареший способ определения неявных правил для программы make. Суффиксные правила являются устаревшими, поскольку шаблонные правила - более общие и более ясные. Они поддерживаются в GNU-версии программы make для совместимости со старыми make-файлами. Они разделяются два типа: правила двойного суффикса и правило одиночного суффикса.

Правило двойного суффикса определяется парой суффиксов: суффикс цели и суффикс исходного файла. Этому правилу соответствует любой файл, имя которого заканчивается на суффикс цели. Соответствующая неявная зависимость строится путем замены в имени файла суффикса цели на суффикс исходного файла. Двухсуффиксное правило, суффикс цели и суффикс исходного файла которого представляют собой, соответственно, '.o' и '.c', эквивалентно шаблонному правилу '%.o : %.c'.

Правило одиночного суффикса определяется одиночным суффиксом, который представляет собой суффикс исходного файла. Этому правилу соответствует любое имя файла, а соответствующее имя неявной зависимости создается путем добавления суффикс исходного файла. Правило одиночного суффикса, исходным суффиксом которого является '.c', эквивалентно шаблонному правилу '% : %.c'.

Определения суффиксных правил распознаются при помощи сравнения цели каждого правила с определенным списком известных суффиксов. Если программа make видит правило, целью которого является известный ей суффикс, это правило воспринимается как правило одиночного суффикса. Если программа make видит правило, целью которого являются два известных ей конкатенированных суффикса, это правило берется в качестве правила двойного суффикса.

Например, как '.c', так и '.o' по умолчанию находятся в списке известных суффиксов. Следовательно, если Вв определяете правило, целью которого является '.c.o', то программа make принимает его как правило двойного суффикса с суффиксом исходного файла '.c' и суффиксом цели '.o'. Ниже приведен устаревший способ для опредения правило для компиляции исходного C-файла:

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

Суффиксные правила не могут иметь никаких собственных зависимостей. Если у них есть хоть одна зависимость, они обрабатываются как обычные файлы со странными именами, а не как суффиксные правила. Таким образом, приведенное ниже правило правило:

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

говорит о том, как породить файл '.c.o' из файла зависимости 'foo.h', что совсем не похоже на шаблонное правило:

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

которое, говорит о том, как порождать '.o'-файлы из '.c'-файлов, при этом порождения любых '.o'-файлов с использованием этого шаблонного правила также зависят от 'foo.h'.

Суффиксные правила без команд также не имеют смысла. Они не отменяют действия предыдущих правил, что делают шаблонные правила без команд (смотрите раздел 10.5.6 [Отмена действия неявных правил]). Они просто вводят в базу данных в качестве цели суффикс или пару конкатенированных суффиксов.

Известные программе make суффиксы представляют собой просто имена зависимостей специальной цели .SUFFIXES. Вы можете добавить ваши собственные суффиксы, написав правило для цели .SUFFIXES, которое добавляют дополнительные зависимости, как показаном ниже фрагменте make-файла:

```
.SUFFIXES: .hack .win
```

где в конец списка суффиксов добавляются '.hack' и '.win'.

Если вы хотите убрать известные по умолчанию суффиксы, а не просто добатить к ним новые, напишите для цели .SUFFIXES правило без зависимостей. Это специально обрабатывается как удаление всех существующих зависимостей цели .SUFFIXES. После этого вы можете написать еще одно правило для добавления тех суффиксов, которые вы хотите добавить. Например,

```
.SUFFIXES:            # Удаляет суффиксы по умолчанию
.SUFFIXES: .c .o .h # Определяет новый список суффиксов
```

Опция '-r' или '--no-builtin-rules' опустошвет список суффиксов по умолчанию.

Перед чтение программой make make-файла, определяется переменная SUFFIXES, в качестве значения которой устанавливается список суффиксов, определенный по умолчанию. Вы можете изменить список суффиксов при помощи правила со специальной целью .SUFFIXES, но это не изменяет переменную SUFFIXES.

10.8 Алгоритм поиска неявного правила

Здесь приведена последовательность действий, которую использует программа make при поиске неявного правила для цели t. Эта последовательность действий вступает в действие для каждого правила с двойным двоеточием, у которого нет команд, для каждой цели из обычных правил, ни одно из которых не имеет команд и для каждой зависимости, которая не является целью никакого правила. Кроме того, программа make рекурсивно следует этой последовательности действий при обработке зависимостей, которые происходят из неявных правил, в ходе поиска цепочки правил.

Суффиксные правила не упоминаются в этом алгоритме, поскольку суффиксные правила преобразуются в эквивалентные шаблонные правила, как только программа make считывает их.

Для цели, являющейся элементом архива и представленной в форме 'АРХИВ(ЭЛЕМЕНТ)', данный алгоритм выполняется дважды, сначала с использованием полного имени цели t, а затем с использованием на месте цели t '(ЭЛЕМЕНТА)', если в первом проходе не было найдено ни одного правила.

1. Цель t делится на часть, определяющую каталог, называемую d, и остаток, называемый n. Например, если t - это 'src/foo.o', то d - это 'src/', а n - это 'foo.o'.
2. Создается список всех шаблонных правил, с одной из целей которых сопоставимо t или n. Если шаблон цели содержит символ '/', то он проверяется на соответствие с t, в противном случае - с n.
3. Если хотя бы одно из правил в этом списке не является правилом с произвольным соответствием, то все нетерминальные правила с произвольным соответствием удаляются из списка.
4. Все правила без команд удаляются из списка.
5. Для каждого шаблонного правила из списка:
    1. Ищется основа s, которая представляет собой непустую часть t или n, которую можно сопоставить с символом '%' в шаблоне цели.
    2. Вычисляются имена зависимостей путем подстановки основы s на место символа '%'; если шаблон цели не содержит символа '/', в начало каждого имени зависимости добавляется d.
    3. Проверяется, существуют ли все зависимости или должны ли они существовать. (Если имя файла упоминается в make-файле в качестве цели или в качестве явной зависимости, то мы говорим, что он должен существовать.) Если все зависимости существуют или должны существовать, или же зависимостей нет, то это правило применяется.
6. Если до сих пор не было найдено ни одного шаблонного правила, то производится более сильная попытка поиска. Для каждого шаблонного правила из списка:
    1. Если правило терминальное, то оно игнорируется и происходит переход на следующее правило.
    2. Вычисляются имена зависимостей, как это делалось прежде.
    3. Проверяется, существуют ли все зависимости или должны ли они существовать.

4. Для каждой зависимости, которая не существует, рекурсивно выполняется этот алгоритм, для того, чтобы определить, может ли зависимость быть порождена неявным правилом.
5. Если все зависимости существуют, должны существовать или могут быть порождены с помощью неявных правил, то это правило применяется.

7. Если ни одно неявное правило не применимо, то применяется правило для цели .DEFAULT, если оно есть. В этом случае, цели t передаются те же самые команды, которые есть у цели .DEFAULT. В противном случае, команд для цели t нет.

Как только находится применимое правило, для каждого шаблона цели, отличного от шаблона, которому соответствует t или n, каждое вхождение в шаблон символа '%' заменяется на основу s, и полученное имя файла запоминается до тех пор, пока не будут выполняться команды для порождения заново целевого файла t. После того, как эти команды выполнятся, каждый из тех запомненных имен файлов вводится в базу данных и помечается как обновленный и имеющий такое же время обновления, как и файл t.

Когда для обновления цели t выполняются команды шаблонного правила, автоматические переменные устанавливаются в соответствие с целью и зависимостями. Смотрите раздел 10.5.3 [Автоматические переменные].

11. Использование make для обновления архивных файлов

Архивные файлы представляют собой файлы, содержащие именованные подфайлы, называемые элементами; они обрабатываются с помощью программы ar и основное их использование - в качестве библиотек подпрограмм для компоновки.

11.1 Элементы архивов в качестве целей

Отдельный элемент архивированного файла может использоваться р программе make в качестве цели или зависимости. Вы указываете элемент с именем ЭЛЕМЕНТ в архивном файле с именем АРХИВ следующим образом:

```
АРХИВ(ЭЛЕМЕНТ)
```

Эта структура доступна только в целях и зависимостях, но не в командах! Большинство программ, которые вы можете использовать в командах не поддерживают этот синтаксис и не могут работать непосредственно с элементами архивов. Только программа ar и другие программы, специально созданные для работы с архивами, могут это сделать. Следовательно, корректные команды для обновления цели, являющейся элементом архива, вероятно, должны использовать программу ar. Например, приведенное ниже правило указывает создать в архиве 'foolib' элемент 'hack.o' путем копирования файла 'hack.o':

```
foolib(hack.o) : hack.o ar
        cr foolib hack.o
```

Фактически, почти все цели, являющиеся элементами архивов, обновляются только таким способом и существует неявное правило, которое делает это за вас. Примечание: программе ar требуется опция 'c', если архивный файл еще не существует.

Для того, чтобы указать на несколько элементов одного и того же архива, вы написать вместе внутри скобок все имена элементов. Например, следующий фрагмент make-файла:

```
foolib(hack.o kludge.o)
```

эквивалентен такому фрагменту:

```
foolib(hack.o) foolib(kludge.o)
```

Вы можете также при ссылке на элемент архива использовать шаблоны в стиле командной оболочки. Смотрите раздел 4.2 [Использование шаблонных символов в именах файлов]. Например, 'foolib(*.o)' преобразуется в список всех существующих элементов архива 'foolib', имена которых заканчиваются на '.o', возможно, это будет такой список: 'foolib(hack.o) foolib(kludge.o)'.

11.2 Неявные правила для целей, являющихся элементами архивов

Напомним, что цель, представленная в виде 'a(m)', означает элемент архивного файла a с именем m.

Когда программа make ищет неявное правило для такой цели, в качестве специальной возможности, он рассматривает неявные правила, с которыми сопоставляется '(m)', наравне с теми правилами, с которыми сопоставляется настоящая цель 'a()'.

Это приводит к сопоставлению с одним специальным правилом, целью которого является '(%)'. Это правило обновляет цель 'a()' с помощью копирования в архив файла m. Например, он обновит цель 'foo.a(bar.o)', являющуюся элементом архива с помощью копирования в архив 'foo.a', в качестве элемента с именем 'bar.o', файла 'bar.o'.

Когда это правило, объединяется в цепочку с другими, это приводит к очень мощному результату. Так, команды 'make "foo.a(bar.o)"' (кавычки необходимы для того, чтобы защитить символы '(' и ')' от специальной интерпретации со стороны командной оболочки), при наличии файла 'bar.c', достаточно для того, чтобы, даже без make-файла, привести к выполнению следующих команд:

```
cc –c bar.c –o bar.o ar
r foo.a bar.o rm –f
bar.o
```

В данном случае программа make увидела в файле 'bar.o' промежуточный файл. Смотрите раздел 10.4 [Цепочки неявных правил].

Неявные правила, как, например, рассмотренное, пишутся с использованием автоматической переменной '$%'. Смотрите раздел 10.5.3 [Автоматические Переменные].

В архиве имя элемента архива не может содержать имени каталога, но в make-файле может быть полезным сделать вид, что оно может это делать. Если вы указываете элемент 'foo.a(dir/file.o)' , являющийся элементом архива, то программа make выполнит автоматическое обновление с помощью такой команды:

```
ar r foo.a dir/file.o
```

действия которой состоят в копировании файла 'dir/file.o' элемент архива с именем 'file.o'. При таком использовании, могут быть полезными автоматические переменные '%D' и 'F'.

Обновление символьных каталогов архивов

Архивный файл, используемый в качестве библиотеки, обычно содержит специальный элемент с именем '__.SYMDEF', который содержит каталог внешних имен символов, определенных всеми другими элементами архива. После того, как вы обновите любые другие элементы архива, вам потребуется обновить '__.SYMDEF', для того, чтобы он содержал корректную суммарную информацию о других элементах архива. Это делается посредством запуска программы ranlib:

```
ranlib АРХИВНЫЙ_ФАЙЛ
```

Обычно вам следует установить эту команду в правило для архивного файла и сделать все элементы архивного файла зависимостями этого правила. Например,

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

Действие этого правила заключается в обновлении элементов архива 'x.o', 'y.o', и т.п., и последующем обновлении элемента '__.SYMDEF', являющегося символьным каталогом, посредством запуска программы ranlib. Правила для обновления элементов здесь не показаны - вероятнее всего, вы можете опустить их и использовать неявное правило, которое копирует файлы в архив, как описано в предыдущем разделе.

Это не является необходимым при использовании GNU-версии программы ar, которая обновляет элемент '__.SYMDEF' автоматически.

11.3 Опасности при использовании архивов

Важно быть внимательным при использовании параллельного выполнения (опция -j, смотрите раздел 5.3 [Параллельное выполнение]) и архивов. Если одновременно запускаются несколько команд ar, работающие с одним и тем же архивным файлом, они не будут знать друг о друге и могут повредить файл.

Возможно, будущая версия программы make обеспечит механизм, для обхода этой проблемы с помощью упорядочивания всех команд, которые работают с одними и теми же архивными файлами. Но в настоящее время вы должны либо писать ваши make-файлы так, чтобы каким-нибудь другим способом избежать этой проблемы, либо не использовать опцию '-j'.

11.4 Суффиксные правила для архивных файлов

Вы можете написать суффиксное правило специального вида, которое бы имело дело с архивными файлами. Смотрите раздел 10.7 [Суффиксные правила], где в полной мере объясняются суффиксные правила. Архивные суффиксные правила являются устаревшими GNU-версии программы make, поскольку шаблонные правила для архивов представляют собой более общий механизм (смотрите раздел 11.2 [Обновление архивов]). Тем не менее, они сохранены для совместимости с другими версиями программы make.

Для того, чтобы написать суффиксное правило для архивов, вы просто пишете суффиксное правило с использованием суффикса цели '.a' (обычный суффикс для архивных файлов). Например, ниже привкдкно устаревшее суффиксное правило для обновления библиотечного архива из исходных C-файлов:

```
.c.a:
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o $(AR)
        r $@ $*.o $(RM) $*.o
```

Этот фрагмент make-файла работает точно так же, как если бы вы написали такое шаблонное правило:

```
(%.o): %.c $(CC)
        $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o $(AR) r
        $@       $*.o $(RM) $*.o
```

Фактически, здесь просто показано то, что делает программа make, когда она видит суффиксное правило с '.a' в качестве суффикса цели. Любое правило двойного суффикса '.x.a' преобразуется в шаблонное правило с шаблоном цели '(%.o)' и шаблоном зависимости '%.x'.

Поскольку вы могли бы захотеть использовать '.a' в качестве для какого-нибудь другого типа файла, программа make также преобразует архивные суффиксные правила в шаблонные правила и обычным способом (смотрите раздел 10.7 [Суффиксные правила]). Таким образом, правило двойного суффикса '.x.a' порождает два шаблонных правила: '(%.o): %.x' и '%.a: %.x'.

12. Особенности GNU-версии программы make

Здесь приведен обзор особонностей GNU-версии программы make, которые отличают ее от других версий, а также заимствованы из других версий программы make. Мы берем за основу особенности программы make в системах 4.2 BSD. Если вы заинтересованы в написании переносимых make-файлов, то вам следует использовать только возможности программы make, не перечисленные здесь и в главе 13 [Недостающие возможности].

Многие особенности происходят из версии программы make для системы System V.

- Переменная VPATH и ее специальный смысл. Смотрите раздел 4.3 "Поиск зависимостей по каталогам". Эта особенность существует в программе make

для системы System V, но является недокументированной. Она документирована в программе make для сисеты 4.3 BSD (где сообщается, что она имитирует возможности переменной VPATH из системы System особенности).

- Включаемые make-файлы. Смотрите раздел 3.3 "Включение других make-файлов". Расширением в GNU-версии является разрешение на включение нескольких файлов с помошью одной директивы.
- Переменные читаются из командной среды передаются через командную среду. Смотрите раздел 6.9 "Переменные из командной среды".
- Опции для рекурсивных вызовов программы make передаются через переменную MAKEFLAGS. Смотрите раздел 5.6.3 "Опции для связи с порожденным процессом make".
- При работе с архивом, в качестве значения автоматической переменной $% устанавилается имя элемента архива. Смотрите раздел 10.5.3 "Автоматические переменные".
- Автоматические переменные $@ , $*, $<, $%, и $? имеют, соответствующие формы вида $(@F) и $(@D). В качестве напрашивающегося расширения, мы обобщили это и на переменную $^. Смотрите раздел 10.5.3 "Автоматические переменные".
- Замена обращения к переменную. Смотрите раздел 6.1 "Основы обращения к переменным".
- Опции командной строки '-b' и '-m' врспринимаются и игнорируются. В программе make для системы System V эти опции кое-что реально делают.
- Выполнение рекурсивных команд для запуска программы make через переменную MAKE, даже если определены опции '-n', '-q' или '-t'. Смотрите раздел 5.6 "Рекурсивное использование программы make".
- Поддержка суффикса '.a' в суффиксных правилах. Смотрите раздел 11.4 "Архивные суффиксные правила". Эта особенность является устаревшей в GNU-версии программы make, поскольку общая возможность построения цепочки правил (смотрите раздел 10.4 "Цепочки неявных правил") позволяет отдельному шаблонному правилу устанавливать элементы в архиве (смотрите раздел 11.2 "Обновление архива"), и этого достаточно.
- Расположение строк и комбинаций символов backslash и конца строки в командах сохраняются при печати команд, поэтому они появляются в таком же виде, как и в make-файле, за исключением удаления первоначальных пробельных символов.

На реализацию следующих возможностей нас вдохновили всевозможные другие версии программы make. В некоторых случаях точно неясно, какие версии повлияли на развитие каких других версий.

- Шаблонное правило с использованием символа '%'. Это было реализовано в нескольких версиях программы make. Мы не уверены в том, кто изобрел такую конструкцию первым, но она распространилось повсюду. Смотрите раздел 10.5 "Определение и переопределение шаблонных правил".
- Построение цепочки правил и неявных промежуточных файлов. Это было реализовано Стью Фельдманом в его версии программы make для системы AT&T Eighth Edition REsearch Unix, а позже - Андрю Ньюмом из AT&T Bell Labs

в его программа mk (где он называет это "транзитивным замыканием"). Мы, на самом деле, не знаем, взяли ли мы это от кого-либо из них или же одновременно додумались об этом сами. Смотрите раздел 10.4 "Цепочки неявных правил".

- Автоматическая переменная $^, содержащая список всех зависимостей текущей цели. Мы этого не изобретали это, но мы понятия не имеем, кто это сделал. Смотрите раздел 10.5.3 "Автоматические переменные". Автоматическая переменная $+ представляет собой просто расширение переменной $^.
- Опция "А что, если" (`-W' в GNU-версии программы make) была изобретена (насколько нам известно) Андрю Хьюмом и применена в программе mk. Смотрите раздел 9.3 "Вместо исполнения команд".
- Концепция выполнения нескольких работ одновременно (параллелизм) существует во многих реализациях программы make и аналогичных программах, хотя ее нет в реализациях для систем System V и BSD. Смотрите раздел 5.2 "Выполнение команд".
- Модифицированное обращение к переменной ссылки с использованием шаблонной подстановки происходит из системы SunOS 4. Смотрите раздел 6.1 "Основы обращения к переменным". Такая функциональность была обеспечена в GNU-версии программы make посредством функции patsubst, прежде, чем был реализован альтернативный синтаксис для совместимости с системой SunOS 4. Здесь не до конца ясно, кто на кого повлиял, так как в GNU-версии программы make функция patsubst была до того, как былаъ выпущена система SunOS 4.
- Специальное значение символов '+', предшествующих командным строкам (смотрите раздел 9.3 "Вместо исполнения команд") предписано стандартом IEEE 1003.2-1992 (POSIX.2).
- Синтаксис, связанный с конструкцией '+=', предназначенный для добавления к значению переменной, происходит из программы make для системы SunOS 4 Смотрите раздел 6.6 "Добавление дополнительного фрагмента к переменным".
- Синтаксисическая конструкция 'АРХИВ(ЭЛЕМ1 ЭЛЕМ2...)' для перечисления нескольких элементов одного архивного файла происходит из программы make для системы SunOS 4. Смотрите раздел 11.1 "Элементы архива".
- Директива -include для включения make-файлов без генерации ошибки для несуществующего файла происходит из программы make для системы SunOS 4. (Но обратите внимание, что программа make для системы SunOS 4 не позволяет нескольким make-файлам быть указанным в одной директивк -include.)

Оставшиеся возможности - новые изобретения GNU-версии программы make:

- Использование опции '-v' или '--version' опции для вывода на экран версии и информации о copyright.
- Использование опцию '-h' или '--help' для вывода на экран полной полной информации о каждой опции программы make.
- Упрощенно-вычисляемые переменные. Смотрите раздел 6.2 "Две разновидности переменных".

- Автоматическая передача через переменную MAKE присваиваний значений переменным в командной строке рекурсивным вызовам программы make. Смотрите раздел 5.6 "Рекурсивное использование программы make".
- Использование опции командной строки '-C' или '--directory' для измения каталога. Смотрите раздел 9.7 "Обзор опций".
- Реализация определений многостроковых переменных при помощи директивы define. Смотрите раздел 6.8 "Определение многостроковых переменных".
- Объявление целей, являющихся именами действий, при помощи специальной цели .PHONY. Андрю Хьюм из AT&T Bell Labs в своей программе mk реализовал аналогичную возможность с помощью другого синтаксиса. Это похоже на случай параллельного открытия. Смотрите раздел 4.4 "Цели-имена действий".
- Манипуляции с текстом с помощью вызывов функций. Смотрите главу 8 "Функции преобразования текста".
- Использование опции '-o' или '--old-file' для иммитации того, что время изменения файла старое. Смотрите раздел 9.4 "Предотвращение перекомпиляции некоторых файлов".
- Условное выполнение. Эта возможность была реализована много раз в различных версиях программы make; она выглядит естественным расширением, наследуемым из возможностей C-препроцессора и аналогичных макроязыков и не является революционной концепцией. Смотрите главу 7 "Условные части make-файла".
- Определение пути поиска для включаемых make-файлов. Смотрите раздел 3.3 "Включение других make-файлов".
- Определение дополнительных make-файлов, подлежащих чтению, с помощью переменной командной среды. Смотрите раздел 3.4 "Переменная MAKEFILES".
- Удаление из имен файлов ведущей последовательности подстрок './', поэтому './file' и 'file'рассматриваются как один и тот же файл.
- Использование специального матода поиска для библиотечных зависимостей, записанных в форме '-lname' Смотрите раздел 4.3.5 "Поиск по каталогам библиотек для компоновки".
- Разрешение суффиксам в суффиксных правилах (смотрите раздел 10.7 "Устаревшие суффиксные правила") содержать любые символы. В других версиях программы make они должны начинаться с символа '.' и не содержать ни одного символа '/'.
- Отслеживание текущего уровня рекурсии программы make с использованием переменной MAKELEVEL. Смотрите раздел 5.6 "Рекурсивное использование программы make".
- Определение статических шаблонных правил. Смотрите раздел 4.10 "Статические шаблонные правила".
- Поддержка выборочного поиска на основе значения переменной vpath. Смотрите раздел 4.3 "Поиск зависимостей по каталогам".

- Поддержка вычисляемых ссылок на переменные. Смотрите раздел 6.1 "Основы обращения к переменным".
- Обновление make-файлов. Смотрите раздел 3.5 "Как переделываются make-файлы". Программа make для системы System V имеет очень, очень ограниченную форму такой функциональности, заключающуюся в том, что она сравнит выходные SCCS-файлы с make-файлами.
- Различные новые встроенные неявные правила. Смотрите раздел 10.2 "Перечень неявных правил".
- Встроенная переменная MAKE_VERSION показывает номер версии программы make.

13. Несовместимость и недостающие возможности

Программы make на различных других системах поддерживают некоторые возможности, которые не реализованы в GNU-версии программы make. Стандарт POSIX.2 (стандарт IEEE 1003.2-1992), который специфицирует программу make, не требует ни одной из этих возможностей.

- Цель в форме 'ФАЙЛ((ВХОЖДЕНИЕ))' означает элемент архивного файла с именем ФАЙЛ. Элемент выбирается не по имени, а по тому, является ли он объектным файлом, который определяет символ компоновщика с именем ВХОЖДЕНИЕ. Эта особенность не была утановлена в GNU-версию программы make, из-за того, что добавление в программу make знаний о внутреннем формате таблицы символов архивного файла противоречит принципу модульности. Смотрите раздел 11.2.1 [Обновление символьных каталогов архивов].

- Суффиксы (используемые в суффиксных правилах), которые заканчиваются символом '~' имеют специальное значение в программе make для системы System V - они ссылаются на SCCS-файл, который соответствует файлу, который получился бы, если бы убрали символ '~'. Например, суффиксное правило '.c~.o' породило бы файл 'n.o' из SCCS-файла 's.n.c'. Для полноты охвата всех требуемых случаев, требуется целый ряд таких суффиксных правил. Смотрите раздел 10.7 "Устаревшие суффиксные правила". В GNU-версии программы make, вся эта совокупность случаев обрабатывается двумя шаблонными правилами для извлечения файла из SCCS, в комбинации с общей возможностью построения цепочки правил. Смотрите раздел 10.4 [Цепочки неявных правил].

- В программе make для системы System V строка '$$@ ' имеет странный смысл, а именно, в зависимостях правила с несколькими целями, она означает конкретную цель, которая обрабатывается. Такая возможность не предусмотрена в GNU-версии программы make, поскольку '$$' всегда означает просто '$'.

  Такая функциональность может быть получена через использование статических шаблонных правил (смотрите раздел 4.10 [Статические шаблонные правила]. Такое правило для программы make из системы System V:

```
$(targets): $$<htmlurl name="@.o" url="mailto:@.o"> lib.a
```

может быть заменено на статическое шаблонное правило для GNU-версии программы make:

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

- В программе make для систем System V и 4.3 BSD, имена файлов, найденных при помощи поиска с использованием переменной VPATH (смотрите раздел 4.3 "Поиск зависимостей по каталогам") изменяются внутри командных строк. Нам кажется, что намного более ясно является использование во всех случаях автоматических переменных и, в результате, эта возможность становится устаревшей.
- В программах make для некоторых Unix-систем, автоматическая переменная $*, появляющаяся среди зависимостей правила, имеет удивительно странную "особенность", заключающуюся подстановке полного имени цели этого правила. Мы не можем представить, что пришло в головы разработчикам программ make для тех Unix-систем, заставив их сделать это; такая особенность полностью противоречит с обычнымим определением переменной $*.
- В программах make для некоторых Unix-систем, поиск неяного правила (смотрите главу 10 "Использование неявных правил"), по-видимому, делается для всех целей, а не только тех, что не имеют команд. Это означает Вы можете написать так:

```
foo.o:

    cc -c foo.c
```

и программы make для тех Unix-систем догадаются, что файл 'foo.o' зависит от файла 'foo.c'. Нам кажется, что такую конструкцию не используют. Состав зависимостей в программе make легко определяется (по крайней мере, в GNU-версии программы make), и работа с такой конструкцией просто не укладывается в общую схему.

- GNU-версия программы make не включает никаких встроенных неявных правил для и обработки препроцессором программ на языке EFL. Если мы услышим о ком-нибудь, кто использует использует язык EFL, мы охотно добавим эти правила.
- Оказывается, что в программе make для системы SVR4 суффиксное правило может быть определено без команд, и оно обрабатывается точно так же, как если бы к него были пустые команды (смотрите раздел 5.8 "Пустые команды"). Например, такое правило:

```
.c.a:
```

перекрывает встроенное суффиксное правило '.c.a'. Нам кажется, что для правил без команд более ясным было бы просто добавлять всегда символ ';' в список зависимостей цели. Вышеприведенный пример может легко быть переписан для получения требуемого поведения GNU-версии программы

make:

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

- Некоторые версии программы make вызывают командную оболочку с опцией '-e', за исключением работы с действующей опцией '-k' (смотрите раздел 9.6 "Проверка компиляции программ"). Опция '-e' указывает командной оболочке заканчивать работу, как только любая запущенная ею программа возвращает ненулевой результат. Нам кажется, что было бы более ясным писать каждую командную строку командной оболочки так, чтобы она стояла на своей собственной строке в make-файле и не требовать этой специальной обработки.

14. Соглашения о make-файлах

Эта глава описывает соглашения по написанию make-файлов для GNU-программ.

14.1 Общие соглашения о make-файлах

Каждый make-файл должен содержать такую строку:

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

во избежании проблем при работе на тех системах, где переменная SHELL могла бы быть унаследована из среды. (Это никогда не является проблемой при работе с GNU-версией программы make.)

Различные версии программы make имеют несовместимые списки суффиксов и неявные правила, и это иногда создает неразбериху или неправильное поведение. Поэтому хорошей идеей является явная установка списков суффиксов с использованием только тех суффиксов, которые вам нужны в конкретном make-файле, как показано ниже:

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

Первая строка очищает список суффиксов, вторая представляет все суффиксы, которые могут быть предметом неявных правил в этом make-файле.

При выполнении команд не закладывайтесь на то, что символ '.' находится в пути. Когда во время работы программы make вам требуется запускать программы, которые являются частью вашего пакета, убедитесь, пожалуйста, что используется './', если построение программы происходит в рамках части программы make или '$(srcdir)/', если файл является неизменяемой частью исходного кода. Если нет ни одного из этих префиксов, то используется текущий путь поиска.

Различие между './' и '$(srcdir)/' важно при использовании опции '--srcdir', установленной в значение 'configure'. Правило в такой форме:

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

не сможет быть выполнено, когда текущий каталог не является исходным каталогом, поскольку файлов 'foo.man' и 'sedscript' нет в текущем каталоге.

При использовании GNU-версии программы make полагаться на переменную VPATH для поиска исходного файла можно в том случае, когда есть один файл зависимости, поскольку автоматическая переменная программы make '$<' будет представлять исходный файл, где бы она ни находилась. (Многие версии программы make устанавливать переменную '$<' только в неявных правилах.) Цель make-файла, представленная в такой форме:

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

должно вместо этого быть записана в таком виде:

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

для того, чтобы позволить переменной VPATH корректно работать. Когда цель имеет несколько зависимостей, явное использование ссылки на переменную '$ (srcdir)'представляет собой самый легкий способ, заставляющий правило работать правильно. Например, лучше всего, чтобы приведенное выше правило для файла 'foo.1' было написано в таком виде:

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

14.2 Использование утилит в Makefile.

Команды, которые пишутся в Makefile (как, впрочем, и в любых других shell-скриптах вроде configure), должны работать в sh, а не в csh. Не следует использовать каких-либо специальных особенностей ksh или bash.

Скрипт configure и правила Makefile для построения и установки программы не должны использовать никаких утилит кроме следующих:

```
cat cmp cp echo egrep expr grep
ln mkdir mv pwd rm rmdir sed test touch
```

Следует использовать только общепринятые опции для этих программ. К примеру, не следует использовать 'mkdir -p', поскольку эта опция хоть и удобна, но большинство систем не поддерживают ее.

Правила Makefile для построения и установки могут также использовать компиляторы и другие необходимые программы, но их использование должно выполняться через make-переменные для того, чтобы пользователь имел возможность заменить их определение на собственную альтернативу. Вот некоторые из программ, которые мы имеем ввиду:

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

Когда Вы используете ranlib, Вы должны проверять его существование в системе, и использовать его только в том случае, если он присутствует. Это необходимо для того, чтобы можно было выполнять построение на системах, не имеющих ranlib.

Если Вы используете символические ссылки, Вы должны поддержать возможность использования Вашего пакета в системах, которые не поддерживают символических ссылок.

Возможно использование других утилит во фрагментах Makefile или скриптах, которые предназначены только для использования в данной конкретной системе, и для которых Вы уверены в их существовании.

14.3 Стандартные цели в Make

Все программы в GNU должны иметь следующие цели в своих Makefile'ах.

'all'	Компиляция всей программой. Эта цель должна быть целью по умолчанию. Эта цель не должна перестраивать никакие файлы документации; info-файлы должны включаться в поставку, DVI файлы должны формироваться только по явному запросу.
'clean'	Выполняется удаление тех файлов из текущего каталога, которые были созданы при построении программы. Не удаляются файлы, в которых сохранена конфигурация. Так же сохраняются файлы, которые могут быть получены при построении, но которые тем не менее входят в поставку. Следует удалять .dvi файлы, если они не являются частью поставки.
'distclean'	Удаляются все файлы из текущего каталога, которые были созданы при конфигурировании или построении программы. Если Вы распакуете исходные тексты программ, после чего построите программу не создавая самостоятельно каких-либо файлов, то 'make distclean' должно оставить только те файлы, которые входили в поставку.
'mostyclean'	Работает так же, как и clean, но оставляет неудаленными некоторые файлы, которые обычно нежелательно перекомпилировать. Например, цель 'mostyclean' для GCC не удаляет файл 'libgcc.a', поскольку его перекомпиляция редко когда бывает нужна, и к тому же занимает много времени.

'realclean'	Выполняется удаление из текущего каталога всего, что может быть построено с помощью Makefile. Обычно это включает в себя все то, что удаляется по distclean, исходные файлы на C, полученные с помощью построителя синтаксических анализаторов Bison, таблицу тегов, info-файлы и т.д. Имеется одно исключение: 'make realclean' не должен удалять 'configure', даже если 'configure' может быть построен используя правило в Makefile. Более того, 'make realclean' не должен удалять ничего из того, чье существование требуется для выполнения 'configure' и начального исполнения программы.

'dvi'	Выполняется построение DVI-файлов для всей TeXinfo-документации. Пример правил:
	`dvi: foo.dvi`
	
	`foo.dvi: foo.texi chap1.texi chap2.texi`
	`  $(TEXI2DVI) $(srcdir)/foo.texi`
	Вы должны определить переменную TEXI2DVI в Makefile. Она должна запускать программу texi2dvi, которая является частью поставки пакета Texinfo. Можно указать просто зависимости, тогда GNU Make сам предоставит эту команду

'dist'	Выполняется создание tar-файла, содержащего дистрибутивную поставку этой программы. tar-файл должен быть создан таким образом, чтобы имена файлов в нем начинались с подкаталога, имя которого являлось бы именем поставляемого пакета. Имя может включать в себя номер версии. Например, поставка дистрибутивного архива для GCC версии 1.40 должна распаковываться в каталог с именем 'gcc-1.40'. Простейший способ выполнить это состоит в создании названного таким образом каталога, и использовании ln или cp для установки соответствующих файлов в него. После чего необходимо выполнить tar для этого подкаталога. Цель dist должна явно зависеть от всех файлов, которые не являются исходными, но должны входить в поставку, для того, чтобы убедиться в их актуальности.
'TAGS'	Обновляет таблицу тегов для программы.

'info'	Выполняется построение всех требуемых info-файлов. Лучший всего написать правила по следующему образцу:
	`info: foo.info`
	
	`foo.info: foo.texi chap1.texi chap2.texi`
	`  $(MAKEINFO) $(srcdir)/foo.texi`
	Вы должны определить в Makefile переменную MAKEINFO. Она должна запускать программу makeinfo, которая входит в поставку пакета Texinfo.

'install'	Компиляция программы и копирование исполнымых файлов, библиотек и т.д. туда, где они должны располагаться для их обычного использования. Если возможно, по этой цели должна выполняться простая проверка того, что программа была правильно установлена.
	
	Команды должны создать все каталоги, в которых файлы будут установлены, если эти каталоги уже не существуют. Сюда входят каталоги, указанные как значения переменных prefix и exec_prefix, так же, как и все их требуемые подкаталоги. Другой способ выполнить это подразумевает использование цели installdirs, описанной ниже.
	
	Используйте символ '-' перед любой командой для установки файлов с man-страницами для того, чтобы игнорировались все ошибки. Это нужно для того, чтобы можно было устанавливать программу на систему, в которой не установлена Unix-система man-документов.
	
	Для установки info-файлов необходимо скопировать их в $(infodir) с $ (INSTALL_DATA) (см. Переменные для исполнения команд), и затем выполнить программу install-info (если она имеется). install-info - это скрипт, который выполняет редактирование файла 'dir' системы Info для того, чтобы добавить или обновить элемент меню для данного Info-файла; этот скрипт является частью пакета Texinfo. Далее приводится пример правила для установки Info-файла:
	
	`$(infodir)/foo.info: foo.info`
	`# There may be a newer info file in . than in srcdir`
	`-if test -f foo.info; then d=.; \`
	`else d=$(srcdir); i;`
	`$(INSTALL_DATA) $$d/foo.info $@ ; \`
	`# Run install-info only if it exists.`
	`# Use 'if' instead of just prepending '-' to the`
	`# line so we notice real errors from install-info.`
	`# We use '$(SHELL) -c' because some shells do`
	`# fail gracefuly when there is an unknown command.`
	`if $(SHELL) -c 'install-info --version' \`
	`>/dev/null 2>&1; then \`
	`install-info --infodir=$(infodir) $$d/foo.info; \`
	`else true; fi`

'uninstall'	Выполняется удаление всех установленных файлов, созданных при исполнении цели 'install' (но не тех файлов, которые создаются при исполнении цели 'all').
'check'	Выполняет самотестирование (если предусмотрено). Пользователь должен построить программу перед запуском тестов, но не должен устанавливать программу; Вы должны написать тесты таким образом, чтобы они работали когда программа построена, но не установлена. Следующие цели в тех программах, в которых они нужны, должны иметь следующие стандартные имена.

'installcheck'	Выполняет проверку правильности установки (если соответствующие тесты предусмотрены). Пользователь должен построить и установить программу перед запуском этих тестов. Вы не должны считать, что $(bindir) входит в путь поиска программ.

'installdirs'	Полезно добавить цель с именем 'installdirs' для создания структуры каталогов, в которых будут установлены файлы. Имеется скрипт, названный 'mkinstalldirs', который подходит для этой цели. Он находится в пакете Texinfo. Вы можете написать правило по следующему образцу:
	
	`# Make sure all installation directories (e.g. $ (bindir))`
	`# actually exists by making them if necessary.`
	`installdirs: mkinstalldirs`
	`$(srcdir)/mkinstalldirs $(bindir) $(datadir) \`
	`$(libdir) $(infodir) \`
	`$(mandir)`

14.4 Переменные для указания команд.

Makefile должен предоставлять переменные для того, чтобы можно было перекрыть некоторые команды, опции и т.д.

В частности, Вы должны запускать большинство утилит через переменные. Так, если Вы используете Bison, введите переменную с именем BISON, чье значение по умолчанию установлено как 'BISON=bison', и ссылайтесь на нее $(BISON) везде, где Вам нужно использовать Bison.

Утилиты управления файлами (такие, как ln, rm, mv и т.д.) не должны выполняться через переменные, поскольку пользователям нет необходимости заменять их другими программами.

Каждой переменной с именем программы должна соответствовать переменная с опциями, которые должны передаваться этой программе. Следует добавлять 'FLAGS' к имени переменной для программы, чтобы получить имя переменной для опций - например, BISONFLAGS. (Имя CFLAGS является исключением из этого правила, но мы сохраняем его, поскольку оно общепринято.) Используйте CPPFLAGS в любой команде компиляции, которая запускает препроцессор, и LDFLAGS в любой команде компиляции, которая выполняет редактирование связей, так же, как и при явном использовании ld.

Если имеются опции C-компилятора, которые должны быть использованы для правильной компиляции некоторых файлов, не включайте их в CFLAGS. Пользователи ожидают, что они могут свободно установить CFLAGS сами. Вместо этого, упорядочите передачу таких опций компилятору независимо от CFLAGS, указывая их явно в командах компиляции или определяя неявное правило как здесь:

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

Не следует включать опцию -g в CFLAGS, поскольку она не требуется для правильной компиляции. Вы можете рассматривать это умолчание как рекоменуемое. Если пакет установлен так, что он компилируется с использованием GCC по умолчанию, то вы можете включить '-O' в умалчиваемое значение переменной CFLAGS.

Помещайте CFLAGS последним в команде компиляции, после всех других переменных, содержащих опции компилятора, для того, чтобы пользователь мог использовать CFLAGS для перекрытия опций, указанных в других переменных.

Каждый Makefile должен определять переменную INSTALL, которая обозначает бызовою команду для установки файла в системе.

Каждый Makefile должен также определять переменные INSTALL_PROGRAM и INSTALL_DATA. (Умолчание для этих переменных должно быть $(INSTALL).) В дальнейшем, эти переменные должны использоваться для установки соответственно исполнимых и неисполнимых файлов. Используйте эти переменные как в примере:

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

Следует всегда указывать в качестве второго аргумента имя файла (не имя каталога). Надо использовать отдельную команду для каждого устанавливаемого файла.

14.5 Переменные для каталогов

Каталоги для установки должны всегда именоваться посредством переменных, поскольку это упрощает установку программы в нестандартное место. Стандартные имена для таких переменных следующие:

'prefix'	Префикс, используемый для построения умолчательных значений для переменных, перечисленных ниже. Значение по умолчанию для переменной prefix должно быть '/usr/local' (по крайней мере сейчас).
'exec_prefix'	Префикс, используемый при построении значений по умолчанию для некоторых переменных, перечисленных ниже. Значение по умолчанию для переменной exec_prefix должно быть $(prefix). Как правило, $(prefix) используется для каталогов, которые содержат машиннозависимые файлы (такие, как исполнимые файлы и библиотеки процедур), в то время как $(prefix) используется для остальных каталогов.

'bindir'	Каталог для установки исполнимых программ, которые могут быть запущены пользователем. Обычно, это '/usr/local/bin', но должно быть записано как '$(exec_prefix)/bin'
'libdir'	Каталог для установки исполняемых файлов, которые будут запускаться другими программами, а не пользователем. Объектные файлы и библиотеки объектного кода должны так же попадать в этот каталог. Идея состоит в том, что этот каталог используется для файлов, которые зависят от конкретной архитектуры машины, но не должны находится в пути для команд. Значение для libdir обычно '/usr/local/lib', но должно быть записано как '$(exec_prefix)/lib'.
'datadir'	Каталог для установки файлов с неизменяемыми данными, которые используются программами во время их работы. Этот каталог используется для файлов, которые не зависят от используемого типа машины. Значение этой переменной обычно '/usr/local/lib', но должно быть записано как '$(prefix)/lib'.
'statedir'	Каталог для установки файлов с данными, которые программы могут изменять в процессе своей работы. Эти файлы должны быть независимыми от типа используемой машины, и должны допускать разделение их между машинами при сетевой установке. Значение этой переменной обычно '/usr/local/lib', но должно быть записано как '$(prefix)/lib'
'includedir'	Каталог для установки заголовочных файлов (header-файлов), которые могут быть включены другими пользовательскими программами с помощью директивы препроцессора '#include'. Значение этой переменной обычно '/usr/local/include', но должно быть записано как '$prefix/include'. Большинство компиляторов отличных от GCC не выполняют поиск заголовочных файлов в '/usr/local/include', поэтому установка заголовочных файлов в этот каталог целесообразна только для GCC. Иногда это не представляет из себя проблему, так как некоторые библиотеки предназначены для использования исключительно с GCC. Но имеются так же и библиотеки, предназначенные для работы и с другими компиляторами. Они должны устанавливать свои включаемые файлы в два места, одно из которых определено переменной includedir, а другое - oldincludedir.
'infodir'	Переменная должна содержать имя каталога для установки info-файлов для данного пакета. По умолчанию, ее значение должно быть '/usr/local/info', но должно быть записано как '$(prefix)/info'.

'oldincludedir'	Каталог для установки заголовочных файлов для использования с компиляторами, отличными от GCC. Значение этой переменной обычно '/usr/include'. Команды Makefile должны проверить, не пусто ли значение переменной oldincludedir. Если оно пусто, они не должны пытаться использовать ее и выполнять повторную установку включаемых файлов. Пакет не должен замещать существующие заголовочные файлы в этом каталоге, в случае, если заголовочный файл пришел не из того же пакета. Так, если Ваш пакет Foo предоставляет заголовочный файл 'foo.h', то он должне установить заголовочный файл в каталог, заданный oldincludedir, если (1) foo.h не существует, или (2) foo.h существует и пришел из пакета Foo. Для того, чтобы проверить, что foo.h пришел из пакета Foo, поместите специальную строку в этот файл - часть комментария - и проверьте наличие этой строки с помощью команды grep.
'mandir'	Каталог для установки man-страниц (если они есть) для этого пакета. Переменная должна включать суффикс для соответсвующей секции руководства - обычно '1' для утилит. Обычно значение этой переменной '/usr/local/man/man1', но должно быть записано как '$(prefix)/man/man1'
'man1dir'	Каталог для установки в раздел 1 man-страниц.
'man2dir'	Каталог для установки в раздел 2 man-страниц. Используйте переменные такого рода вместо 'mandir', если пакет должен устанавливать man-страницы более чем в один раздел.
'manext'	Расширение для имени файла для устанавливаемых man-страниц. Переменная должна содержать точку, за которой следует соответствующая цифра, обычно '.1'.
'man1ext'	Расширение имени файла для установки в раздел 1 man-страниц.
'man2ext'	Расширение имени файла для установки в раздел 2 man-страниц. Используйте переменные такого рода вместо 'manext', если пакет должен устанавливать man-страницы более чем в один раздел.
'srcdir'	В этой переменной должно находится имя каталога, содержащего компилируемые исходные тексты. Значение этой переменной обычно вставляется скриптом configure.

Пример:

```
 # Common prefix for installation directories.
 # NOTE: This directory must exist when you start the
install.
 prefix = /usr/local
 exec_prefix = $(prefix)
 # Where to put the executable for the command 'gcc'.
 bindir = $(exec_prefix)/bin
 # Where to put the directories used by the compiler.
 libdir = $(exec_prefix)/lib
 # Where to put the Info files.
 infodir = $(prefix)/info
```

Если Ваша программа устанавливает большое количество файлов в один из стандартных каталогов, указанных пользователем, целесообразно сгруппировать их в один подкаталог, относящийся к этой программе. Если Вы делаете это, Вы должны создать такие подкаталоги в правиле для цели install.

Не ожидайте от пользователя указания имени такого подкаталога в заданном им значении переменных, перечисленных выше. Наличие однообразного набора имен переменных для каталогов, в которые будет установлена программа, позволяет пользователю указать точно такие же значения для нескольких различных GNU-пакетов. Для того, чтобы это было полезным, все пакеты должны быть разработаны таким образом, чтобы они правильно использовали значения переменных.

15. Приложение. Комплексный пример Make-файла.

В этом приложении приводится текст Make-файла для программы tar. Это сравнительно сложный Make-файл.

Целью по умолчанию в этом Make-файле является 'all', поскольку это первая цель. Интересной особенностью этого Make-файла является то, что файл 'testpad.h' генерируется автоматически с помощью программы 'testpad', которая в свою очередь получается компиляцией файла 'testpad.c'.

Если Вы наберете 'make' или 'make all', то make создаст исполнимый модуль 'tar', демон 'rmt', который предоставляет доступ к накопителям на магнитной ленте, расположенным на других машинах, и info-файл 'tar.info'.

Если Вы наберете 'make install', то make не только создаст 'tar', 'rmt' и 'tar.info', но и установит их.

Если Вы наберете 'make clean', то make удалит все '.o'-файлы, а так же файлы 'tar', 'rmt', 'testpad', testpad.h' и 'core'.

Если Вы наберете 'make distclean', то make удалит не только файлы, которые удаляются при исполнении цели 'clean', но также и файлы 'TAGS', 'Makefile' и 'config.status'. (Хоть это и неочевидно, файлы 'Makefile' и 'config.status' генерируются пользователем с помощью программы 'configure', которая входит в состав поставки пакета 'tar', но не показана здесь.)

Если Вы наберете 'make realclean', то make удалит файлы, удаляемые при исполнении цели 'distclean', и info-файлы, которые строятся из исходного файла 'tar.texinfo'.

Наконец, цели 'shar' и 'dist' строят дистрибутив, используемый для распространения.

```
# Generated automatically from Makefile.in by configure.
# Un*x Makefile for GNU tar program.
# 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

#### Start of system configuration section. ####

srcdir = .

# If you use gcc, you should either run the
# fixincludes script that comes with it or else use
# gcc with the -traditional option.  Otherwise ioctl
# calls will be compiled incorrectly on some systems.
CC = gcc -O
YACC = bison -y
INSTALL = /usr/local/bin/install -c
INSTALLDATA = /usr/local/bin/install -c -m 644

# Things you might add to DEFS:
# -DSTDC_HEADERS        If you have ANSI C headers and
#                       libraries.
# -DPOSIX               If you have POSIX.1 headers and
#                       libraries.
# -DBSD42               If you have sys/dir.h (unless
#                       you use -DPOSIX), sys/file.h,
#                       and st_blocks in `struct stat'.
# -DUSG                 If you have System V/ANSI C
#                       string and memory functions
#                       and headers, sys/sysmacros.h,
#                       fcntl.h, getcwd, no valloc,
#                       and ndir.h (unless
#                       you use -DDIRENT).
# -DNO_MEMORY_H         If USG or STDC_HEADERS but do not
#                       include memory.h.
# -DDIRENT              If USG and you have dirent.h
#                       instead of ndir.h.
# -DSIGTYPE=int         If your signal handlers
```

```
#                       return int, not void.
# -DNO_MTIO             If you lack sys/mtio.h
#                       (magtape ioctls).
# -DNO_REMOTE           If you do not have a remote shell
#                       or rexec.
# -DUSE_REXEC           To use rexec for remote tape
#                       operations instead of
#                       forking rsh or remsh.
# -DVPRINTF_MISSING     If you lack vprintf function
#                       (but have _doprnt).
# -DDOPRNT_MISSING      If you lack _doprnt function.
#                       Also need to define
#                       -DVPRINTF_MISSING.
# -DFTIME_MISSING       If you lack ftime system call.
# -DSTRSTR_MISSING      If you lack strstr function.
# -DVALLOC_MISSING      If you lack valloc function.
# -DMKDIR_MISSING       If you lack mkdir and
#                       rmdir system calls.
# -DRENAME_MISSING      If you lack rename system call.
# -DFTRUNCATE_MISSING   If you lack ftruncate
#                       system call.
# -DV7                  On Version 7 Unix (not
#                       tested in a long time).
# -DEMUL_OPEN3          If you lack a 3-argument version
#                       of open, and want to emulate it
#                       with system calls you do have.
# -DNO_OPEN3            If you lack the 3-argument open
#                       and want to disable the tar -k
#                       option instead of emulating open.
# -DXENIX               If you have sys/inode.h
#                       and need it 94 to be included.

DEFS =  -DSIGTYPE=int -DDIRENT -DSTRSTR_MISSING \
        -DVPRINTF_MISSING -DBSD42
# Set this to rtapelib.o unless you defined NO_REMOTE,
# in which case make it empty.
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
# Prefix for each installed program,
# normally empty or `g'.
binprefix =
```

```
# The directory to install tar in.
bindir = $(prefix)/bin

# The directory to install the info files in.
infodir = $(prefix)/info

#### End of system configuration section. ####

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`
        -rm -rf `cat .fname` .fname
        tar chZf `cat .fname`.tar.Z `cat .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
```