Тема 7. Инструменты автоматической сборки проектов

7.1 Введение

Команды, с помощью которых мы компилируем и компонуем наши программы на C/C++ и ассемблере, могут быть очень сложными и объемными. Утилита *make* автоматизирует процесс компиляции программ любого размера и сложности так, что одна единственная команда *make* заменяет сотни команд компиляции и компоновки. Более того, *make* сравнивает временные штампы связанных файлов во избежание повторения уже выполненной работы. Еще более важно то, что *make* управляет индивидуальными правилами, которые определяют, как строить различный целевой код и автоматически анализирует зависимости между используемыми файлами. СЛАЙД 2.

Существует целый ряд различных версий *make*, их особенности и использование отличаются в разной степени. Они имеют различные наборы встроенных переменных и задач со специальными значениями. Мы сконцентрируемся на тех особенностях *make*, которые чаще всего используются. В некоторых системах эта утилита может называться иначе, например, *gmake*. Для более подробного изучения *make* рекомендуется ознакомиться с официальной документацией [2] и/или свободно-доступной книге [4].

7.2 Цели, зависимости и команды

Перед тем, как описать *make*, давайте коротко опишем проблему. Чтобы получить исполняемую программу, нам нужно скомпоновать компилированные объектные файлы. Чтобы сгенерировать объектные файлы, мы компилируем исходные файлы, написанные на C, C++ и других языках. Исходные файлы, в частности, на C/C++ в свою очередь нуждаются в препроцессинге заголовочных файлов. И всякий раз, когда мы имеем редактированный исходный текст, все файлы прямо или косвенно генерируемые по нему, должны быть пересобраны.

Утилита *make* организует эту работу в форме **правил**. Для программ на C/C++ эти правила обычно принимают следующую форму: исполняемый файл это **цель**, которая должна быть пересобрана после изменения любого объектного файла. Объектные файлы – это **зависимости**. В то же самое время объектные файлы – это промежуточные цели, которые нужно перекомпилировать, если исходные файлы изменились. Таким образом, исполняемые файлы косвенно зависят от «исходников». Утилита *make* элегантно управляет такими цепочками зависимостей, даже когда они становятся очень запутанными. Правило для каждой цели обычно содержит одну или более команд, называемых **командными скриптами**, которые *make* выполняет для сборки программ. Например, правило для исполняемого файла говорит, что нужно запустить компоновщик. А правило для построения объектных файлов говорит, что нужно запустить препроцессор или компилятор. Иначе говоря, правила зависимостей говорят, **когда** собирать целевую программу, а командный скрипт говорит, **как** это осуществить.

У программы *make* имеется специальный синтаксис правил. Значит, правила для всех операций, которые должна выполнить *make* для сборки нашего проекта крайне

желательно разместить в файле, который утилита сможет прочитать. Это делается в командной строке с помощью опции *-f filename*, чтобы сообщить *make*, в каком файле находятся правила (назовем его ***make*-файл**). Обычно же данная опция не указывается, и *make* пытается найти файл с именем по умолчанию *makefile*, в случае его отсутствия – файл с именем, соответствующим маске *Makefile.**. При чтении *make*-файла следует помнить, что это не просто скрипты, выполняемые в заданном порядке. Утилита первым делом проанализирует все содержимое файла, чтобы сконструировать дерево связей между возможными целями и их зависимостями, а затем будет обходить дерево для построения желаемых целей.

В дополнение к правилам *make*-файлы могут содержать комментарии; присваивания значений переменным; макроопределения; директивы включения содержимого и условные директивы.

На СЛАЙДЕ 3 приведен пример *make*-файла для простой программы, состоящей из двух исходных файлов.

Строка, которая начинается с символа #, это комментарий, его *make* игнорирует. Далее даются определения нескольких переменных, которые используются в дальнейшем. Остальная часть файла состоит из правил, общая форма которых следующая (СЛАЙД 4):

```
цель1  [цель2 ...]  :[:]  [зависимость1 ...]
       [; команды]  [# ...]    [\t команды ]  [# ...]
```

Указанные в скобках компоненты могут быть опущены, цели и зависимости являются цепочками из букв, цифр, символов точки и косой черты. При обработке строки интерпретируются метасимволы оболочки, такие как * и ?. Команды могут быть указаны после точки с запятой в строке зависимостей, или в строках, начинающихся с табуляции, которые следуют сразу за строкой зависимостей. **Команда** – это произвольная цепочка символов, не содержащая знак #, за исключением тех случаев, когда # заключен в кавычки.

Видно, что *цель1* начинается со строки без лидирующих пробелов, а каждую команду размещают после символа табуляции. Каждое правило в *make*-файле гласит: если какая-то *цель* старше, чем любая зависимость, то выполняется командный скрипт. Также утилита проверяет, имеются ли зависимости у зависимостей. Это делается перед тем, как начать выполнение скриптов.

Зависимости и командные скрипты опциональны. Правило без скрипта сообщает утилите об отношении зависимости, а правило без зависимостей сообщает *make* о том, **как** построить цель, а не **когда** сделать это. Мы также можем разместить зависимости для заданной цели в одном правиле, а командный скрипт – в другом. Для любой запрошенной цели, задана она в командной строке или как зависимость от другой цели, *make* собирает всю необходимую информацию из **всех** правил для этой цели прежде, чем работать с ними.

Код на СЛАЙДЕ 3 демонстрирует две различные нотации для ссылок на переменные в командном скрипте. Имена переменных, которые состоят из более, чем одного символа (у нас – *CC*, *CFLAGS* и *LDFLAGS*) должны предваряться знаком $ и заключаться в круглые скобки при использовании). Переменные с именами из одного символа (у нас – это автоматические переменные ^, < и @) нуждаются только в знаке $. Следующая команда (*$make -n*) показывает, как *make* подставляет оба вида переменных, чтобы сгенерировать инструкции для компилятора.

Аргумент командной строки *-n* позволяет *make* только печатать команды вместо их выполнения. Эта опция незаменима при тестировании *make*-файлов. При выводе нижняя строка соответствует первому правилу из нашего файла. Как видно, *make* заменяет ссылку на переменную $(*CC*) на текст *gcc*, а $(*LDFLAGS*) на *-lm*. Автоматические переменные $@ и $^ заменяются на цель *circle* и список зависимостей *circle.o circulararea.o*. В первых же двух выведенных строках автоматическая переменная $< заменяется лишь на одну

зависимость – имя компилируемого исходного файла.

Относительно командной части правил *make* важно помнить, что это не *shell*-скрипты. Когда *make* запускает правило, чтобы построить цель, каждая строка в командной секции правила выполняется индивидуально, т.е. в отдельной копии *shell*. Таким образом, мы должны гарантировать, что ни одна из команд не зависит от побочных эффектов на предшествующей строке (СЛАЙД 4). Например, следующие команды не запустят *etags* в подкаталоге *src*.

```
TAGS:
 cd src/
 etags *.c
```

При попытке построения TAGS *make* запускает команду оболочки *cd src/* в текущем каталоге. Когда команда заканчивается, *make* запускает *etags *.c* в новом экземпляре оболочки, но опять в текущем каталоге. Существуют разные способы позволить нескольким командам выполняться в одном процессе оболочки. Можно записать в одну строку, отделив их точкой с запятой, можно добавить обратную косую черту, чтобы помесить их виртуально на одной линии. Например:

```
TAGS:
 cd src/ ;\
 etags *.c
```

Вторая причина для запуска нескольких команд в одной оболочке – ускорение процесса, особенно для построения больших проектов.

Последние две строки в нашем примере показывают повторяющиеся шаблоны. Каждый из объектных файлов зависит от исходных файлов с такими же именами, но уже с суффиксом *.c*. Кроме того, команды для их построения схожи. Утилита *make* позволяет описать такие случаи более экономно с использованием **шаблонных правил**. То есть вместо двух правил мы можем записать одно (СЛАЙД 5).

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

Первая строка правила разбита символом двоеточия на три части вместо двух. Первая часть это список целей, к которым применяется правило. Остаток строки (*%.o: %.c*) это шаблон, объясняющий, как выводить имя зависимости из каждой цели с использованием знака процента в качестве метасимвола. Когда *make* ищет соответствие каждой цели из списка напротив шаблона %.*o*, часть цели, которая соответствует шаблону %, называется **основой**. На нее затем заменяется знак процента в %.*c* для построения зависимости.

Общий вид шаблонных правил имеет простой синтаксис (СЛАЙД 6):

```
[target_list :] target_pattern : prerequisite_pattern
 [command-script]
```

Мы должны гарантировать, что каждая цель в списке соответствует целевому шаблону. В противном случае *make* выдаст сообщение об ошибке.

Если мы подключаем явный целевой список, то правило называется **статическим шаблонным правилом**. Если мы не укажем целевой список, то правило называется **неявным правилом** и применяется к любой цели, чье имя соответствует целевому шаблону. Например, если мы собираемся добавить дополнительные модули к нашей программе *circle*, мы можем указать правило для всех настоящих и будущих объектных файлов в проекте. Правило получится очень простым (средняя часть СЛАЙДА 6):

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

А если некоторым объектом нужно управлять по каким-то причинам иным способом, то мы можем написать статическое шаблонное правило для этого объектного файла. Утилита *make* затем обработает статическое правило для явно заданной цели, и неявное правило для всех остальных объектных файлов. Также *make* выдаст сообщение об ошибке, если для какого-то файла не существует неявной зависимости.

Знак процента обычно используется только один раз в каждом шаблоне. Чтобы использовать этот знак как литерал, его нужно в шаблоне экранировать обратной косой чертой. Например, имя файла *app%3amodule.o* соответствует шаблону *app\%3a%.o*, а результирующая основа – строка *module*. Обратная косая черта как литерал должна экранироваться обратной косой чертой. Таким образом имя файла *app\module.o* должно соответствовать шаблону *app\\%.o*, основа – *module*.

Нам не нужно сообщать *make* о том, как делать стандартные операции, например, компиляция объектного файла из исходного. Программа содержит встроенное правило по умолчанию для этой и других подобных операций. Следующий пример показывает более элегантную версию предыдущего *make*-файла. СЛАЙД 7.

Этот *make*-файл не содержит правил для компиляции исходного кода в объектные файлы вместо этого используются встроенные правила *make*. Более того, правило, сообщающее, что программа *circle* зависит от двух объектных файлов, не содержит командных скриптов. Такой эффект достигнут благодаря тому, что у *make* имеется правило для компоновки объектов в исполняемый файл. Бросим свой взгляд на мгновенье на эти встроенные правила. Допустим, мы ввели следующую команду:

```
$ touch *.c ; make circle
```

Выводимая информация не будет отличаться от прежней. Ни одна из этих команд не видна в *make*-файле, даже если какие-то аргументы могут быть заданы присваиванием значений переменным. Чтобы посмотреть встроенные правила и переменные *make* нужно запустить программу с переключателем *-p*. Выводимая информация будет достаточно объемной. Мы приведем только фрагменты, релевантные нашему примеру (СЛАЙД 8):

```
...
# default
OUTPUT_OPTION = -o $@
# default
LINK.o = $(CC) $(LDFLAGS) $(TARGET_ARCH)
# default
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
%: %.o
# commands to execute (built-in):
$(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
 %.o: %.c
# commands to execute (built-in):
 $(COMPILE.c) $(OUTPUT_OPTION) $<
...
```

Отметим, что шаг компоновки синтезирован из двух правил, *make* автоматически применил команду, определенную встроенным правилом с использованием информации о зависимостях, предоставленной правилом в *make*-файле.

Утилита *make* пытается использовать неявные правила (встроенные или шаблонные) из *make*-файла для любой цели, у которой нет явного правила с командным скриптом. Очевидно, что может быть множество правил, которые соответствуют заданной цели. Например, у *make* имеются встроенные правила для генерации объектного файла (шаблон *%.o*) из исходного кода на C (*%.c*), C++ (*%.cpp*), ассемблера (*%.s*) и других. Какое из правил должна использовать *make*? Она выбирает первое правило из списка, для которого зависимости доступны или могут быть созданы применением подходящих правил. В этом случае *make* автоматически применяет цепочку неявных правил для достижения цели. Если *make* генерирует любой промежуточный файл, не упомянутый в *make*-файле, она

удаляет его сразу после того, как они сделали свое дело. Например, допустим, что текущий каталог содержит только файл *square.c* и *make*-файл со следующим содержимым (СЛАЙД 9).

```
%: %.o
 cc -o $@ $^
%.o : %.c
 cc -c -o $@ $<
```

Чтобы отключить все встроенные правила и использовать только два неявных правила, которые есть в *make*-файле, мы должны запустить *make* с опцией *-r*.

```
$ ls
Makefile square.c
$ make -r square
cc -c -o square.o square.c
cc -o square square.o
rm square.o
$ ls
Makefile square square.c
```

Для достижения этой цели *make* найдет два неявных правила и имеющийся исходный файл, создаст цель, а затем автоматически удалит промежуточный объектный файл, потому что он не упоминается ни в *make*-файле, ни в командной строке.

7.3 Правила с двойным двоеточием

Они так называются, потому что между целями и зависимостями указываются не одно, а два двоеточия (СЛАЙД 10).

```
target :: prerequisites
 commands
```

При использовании двойного двоеточия *make* рассматривает правила как альтернативные, а не накапливающиеся. Вместо сбора всех зависимостей в одно множество для некой цели, *make* проверяет по отдельности цель напротив каждой зависимости, чтобы решить, использовать или нет скрипт для соответствующего правила. Пример на СЛАЙДЕ 10 показывает, как использовать правила с двойным двоеточием.

Здесь содержатся две цели для программы *circle*: с опциями отладки (и без них) в командной строке компилятора. В первом правиле цель зависит от исходных текстов. Утилита *make* запускает команду для этого правила, если исходные файлы новее исполняемого. Во втором правиле *circle* зависит от файла по имени *debug* в текущем каталоге. Команда для этого правила совсем не использует зависимость *debug*. Это пустой файл, он располагается в каталоге только ради своей временной метки, которая сообщает *make*, должна ли она строить отладочную версию. Следующая последовательность команд иллюстрирует, как *make* выбирает между двумя альтернативами.

```
$ make clean
rm -f circle
$ make circle
gcc -Wall -std=c99 -o circle -lm circle.c circulararea.c
$ make circle
make: `circle' is up to date.
$ touch debug
$ make circle
gcc -Wall -std=c99 -ggdb -pg -o circle -lm circle.c circulararea.c
$ make circle
make: `circle' is up to date.
$ make clean
rm -f circle
$ make circle
gcc -Wall -std=c99 -o circle -lm circle.c circulararea.c
```

Как видно, *make* использует либо одно правило, либо другое, смотря у какого

правила зависимость новее, чем цель. В обоих случаях они новее цели, поэтому *make* выбирает правило, указанное первым в *make*-файле.

7.4 Комментарии

В *make*-файле знак диеза (#) в любом месте строки начинает комментарий, за исключением команды. *Make* игнорирует комментарии так, как если бы текста, начиная с этого знака и до конца строки, не существовало. Комментарии и пустые строки между строками не прерывают ее выполнение. Лидирующие пробелы перед знаком диеза игнорируются.

Если строка со знаком диеза является командой (она начинается символом табуляции), то она не может содержать *make*-комментарий. Если соответствующая цель должна быть построена, то *make* передает оболочке для выполнения всю командную строку, естественно, исключив символ табуляции.

7.5 Переменные

Все переменные в *make* одного и того же типа: они содержат последовательности символов и никогда численные значения. Всякий раз применяя правило, *make* вычисляет все переменные, содержащиеся в целях, зависимостях и командах. Переменные в *make* бывают двух видов – рекурсивно-подставляемые и (просто) подставляемые. Какой вид у заданной переменной, определяется конкретным оператором присваивания, используемым при ее определении. Для рекурсивно-подставляемых переменных все вложенные ссылки сохраняются буквально до тех пор, пока переменная не вычислена. Для просто подставляемых переменных ссылки раскрываются непосредственно при выполнении присваивания, и сохраняются их подставленные значения, а не имена.

В именах переменных могут использоваться все символы, за исключением запятой, знаков равенства и диеза. Тем не менее, для надежности и совместимости с оболочкой, настоятельно рекомендуется использовать только литеры, цифры и символы подчеркивания (СЛАЙД 11).

Оператор присваивания ‘=’, который мы используем при определении, указывает на то, что она является рекурсивно-подставляемой. Пример:

```
DEBUGFLAGS = $(CFLAGS) -ggdb -DDEBUG -O0
```

Make сохраняет последовательность символов справа от знака равенства в буквальном виде, вложенная переменная $(*CFLAGS*) не подставляется до тех пор, пока не будет использована переменная $(*DEBUGFLAGS*).

Для создания просто подставляемой переменной мы используем оператор ’:=’.

```
OBJ = circle.o circulararea.o
TESTOBJ := $(OBJ) profile.o
```

В этом случае *make* сохраняет последовательность *circle.o circulararea.o profile.o* как значение $(*TESTOBJ*). Если последующие присваивания модифицируют значение $(*OBJ*), то на $(*TESTOBJ*) это не отражается.

Мы еще можем определить рекурсивно-подставляемые и просто подставляемые переменные не только в *make*-файле, но и в командной строке при запуске *make*. Например, так:

```
$make CFLAGS=-ffinite-math-only circulararea.o
```

Каждое такое присваивание должно указываться в одном аргументе командной строки. Если присваивание содержит пробелы, нужно экранировать их, или же заключить весь оператор в кавычки. Любая переменная, определенная в командной строке или в оболочке, может быть отменена оператором присваивания внутри *make*-файла, если

оператор начинается с ключевого слова *override*, например:

```
override CPPLFAGS = -DDEBUG
```

Использовать такую форму оператора присваивания следует с осторожностью, если мы не хотим сконфузить или расстроить будущих пользователей нашего *make*-файла. См. СЛАЙД 12.

Есть еще две разновидности присваивания: ‘+=’ и ‘?=’. Первая – это оператор добавления, вторая – оператор условного присваивания. См. в соответствующей литературе.

Во время присваивания *make* игнорирует любой пробел между знаком операции и первым непробельным символов в присваиваемом значении. Тем не менее, окружающие пробелы вплоть до конца строки с оператором присваивания или до начала комментария, находящегося на этой строке, становятся частью значения переменной. Обычно такое поведение несущественно, поскольку большая часть ссылок на *make*-переменные задаются в командной строке, где дополнительные пробелы значимыми не являются.

Однако мы можем использовать переменные для конструирования имен файлов или каталогов, и непреднамеренно сделанные пробелы в конце строки с присваиванием могут привести к фатальным последствиям.

С другой стороны, иногда создаются сложные *make*-файлы, и появляется нужда в пробельном символе, который *make* не игнорирует или интерпретирует как разделитель элементов списка. Самый легкий способ решения проблемы состоит в использовании переменной, чье значение – это одиночный пробел, но определение такой переменной сложно. Заключение пробела в апострофы или кавычки, как в языке C, нужного эффекта не дает. Рассмотрим следующее присваивание (СЛАЙД 13):

```
ONESPACE := ' '
TEST = Does$(ONESPACE)this$(ONESPACE)work?
```

В этом случае ссылка на $(*TEST*) приведет к появлению следующего текста:

```
Does' 'this' 'work?
```

Апострофы становятся частью значения переменной. Чтобы определить переменную, содержащую только пробел и ничего более, нам поможет следующий нехитрый прием:

```
NOTHING :=
ONESPACE := $(NOTHING) # This comment terminates the variable’s value.
```

Ссылочная переменная $(*NOTHING*) содержит нуль символов, но она заканчивается лидирующим пробелом, который *make* вырежет после выполнения оператора присваивания. Если мы не вставим комментарий после пробельного символа за $ (*NOTHING*), мы можем в некоторых случаях при редактировании *make*-файла получить определенные проблемы при поиске строки, содержащей пробел как желательный символ.

Мы можем выполнить любую из операций присваивания применительно к определенной цели (или шаблону цели) включением в *make*-файл строки следующей формы (СЛАЙД 13):

```
target_list: [override] assignment
```

Когда *make* строит заданную цель или ее зависимости, специфичная для цели или шаблона переменная заменяет любое другое определение этой же переменной в другом месте *make*-файла.

Пример на СЛАЙДЕ 14 иллюстрирует разные виды присваиваний. Для целей *debug* и *symbols*, этот *make*-файл использует оператор добавления, что прибавить значение *DEBUGCFLAGS* к значению *CFLAGS*, чтобы сохранить уже заданные компилятору флаги.

Присваивание *SYMTABS* иллюстрирует еще одну особенность *make*-переменных: мы можем выполнять подстановки во время их использования. Для этого используется следующая форма (СЛАЙД 15):

```
$(name:ending=new_ending)
```

Когда мы таким способом ссылаемся на переменную, *make* делает подстановку, потом проверяет конец каждого слова в проверяемом значении (слово – это последовательность непробельных символов заканчивающаяся пробелом или конечным элементом значения) до завершения строки. Если слово заканчивается *ending*-ом, то *make* заменяет эту часть *new_ending*-ом. В нашем примере результирующим значением $ (*SYMTABS*) станет *circle.sym circulararea.sym*.

Переменная *CFLAGS* определена в начале файла с помощью безусловного присваивания. Подстановка вложенной переменной $(*ASMFLAGS*), будет отложена до тех пор, пока *make* не выполнит подстановку $(*CFLAGS*), чтобы задать команды компилятору. Значением $(*ASMFLAGS*) могут быть, например, *-Wa,-as=circle.sym,-L* или пустая строка. Когда *make* создает символы для цели, команды компилятору будут рекурсивно раскрыты в следующие строки:

```
gcc -c -Wall -std=c99 -Wa,-as=circle.sym,-L -ggdb -O0 -o circle.o circle.c
gcc -c -Wall -std=c99 -Wa,-as=circulararea.sym,-L -ggdb -O0 -o
circulararea.o circulararea.c
```

Как видно, если не задана переменная с именем *CPPFLAGS* в момент подстановки переменной, *make* просто заменит $(*CPPFLAGS*) на пустую строку.

Как и во многих реальных *make*-файлах, в нашем примере создаются переменные для часто используемых утилит как *mkdir* и *rm* с заданными стандартными опциями. Такой подход позволяет избавиться от повторения командных скриптов, а также облегчает обслуживание и портирование.

Командные скрипты у нас еще содержат ряд односимвольных переменных: $@, $<, $^ и $*. Это **автоматические переменные**, которые *make* определяет и подставляет при выполнении каждого правила. Приведем список автоматических переменных и их значения (СЛАЙД 16).

$@ – Имя файла цели.

$* – Основа имени файла цели, т.е. часть, представленная метасимволом % в шаблонном правиле.

$< – Первая зависимость.

$^ – Список зависимостей, исключая дублирующиеся элементы.

$? – Список зависимостей, которые новее цели.

$+ – Полный список зависимостей, включая дубликаты.

$% – Если цель является элементом архива, то переменная $% позволяет получить имя элемента без указания имени архива, а для получения последнего предназначена $@.

Эта последняя автоматическая переменная используется для особых целей. Поскольку многие программы зависят не только от исходного кода, но и от библиотечных модулей, у *make* имеется специальная нотация для тех целей, которые являются элементами архива:

```
archive_name(member_name): [prerequisites]
  [command_script]
```

Имя элемента архива заключается в круглые скобки немедленно после имени файла-архива. Например:

```
AR = ar -rv
libcircularmath.a(circulararea.o): circulararea.o
$(AR) $@ $%
```

Это правило выполняется, когда следующая команда добавляет или удаляет объектный файл из архива:

```
ar -rv libcircularmath.a circulararea.o
```

Когда автоматические переменные преобразуются в список (например, список имен файлов), элементы отделяются пробелами (СЛАЙД 18). Чтобы отделить имена файлов от каталогов, существуют еще две версии каждой автоматической переменной в списке, чьи имена формируются суффиксами *D* и *F*.

Поскольку результирующее имя переменной состоит из двух символов, требуются круглые скобки. Например, $(@*D*) в любом правиле заменяется на имя каталога для заданной цели, тогда как $(@*F*) дает только имя файла без имени каталога. То же самое можно сделать с помощью функций, однако их использование (так же как и многое другое) мы оставим на самостоятельное изучение.

Переменные, которые *make* использует для своих внутренних нужд, описаны в списке на СЛАЙДЕ 19. Их также можно использовать в своих *make*-файлах. Значения этих переменных можно получить на стандартный вывод путем запуска *make -p*.

Далее мы покажем, что для автоматизации создания *make*-файлов и решения других задач можно использовать *Autotools,* т.е. набор программных средств, предназначенных для поддержки переносимости исходного кода программ между *NIX-подобными системами. Также отметим, что *Autotools* является элементом GNU *Toolchain*.

7.6 Фиктивные цели

Пример *make*-файла со СЛАЙДА 14 иллюстрировал несколько различных способов использования целей. Цели *debug*, *testing*, *production*, *clean* и *symbols* не являются именами генерируемых файлов. Тем не менее, эти команды четко определяют поведение команды типа *make production* и *make clean symbols debug*. Цели, которые не являются именами генерируемых файлов, называются **фиктивными**.

В нашем примере фиктивная цель *clean* имеет командный скрипт, но не имеет зависимостей. Более того, этот скрипт в действительности ничего не создает: наоборот, он удаляет файлы, сгенерированные другими целями. Мы можем использовать такую цель, что стереть все, что мешает начать сборку нашей программы с «чистого листа». Таким образом, цели *testing* и *production* гарантируют, что исполняемый модуль компонуется из объектных файлов с желаемыми опциями компилятора путем указания *clean* в качестве одной из зависимостей.

Мы также можем считать фиктивной такую цель, для которой не предполагается устаревание содержимого. Иными словами, ее скрипт должен выполняться, когда бы не указывалась данная цель. Это как раз случай цели *clean*, при условии, что в каталоге проекта нет файла с именем *clean*.

Довольно часто, тем не менее, имя фиктивной цели появляется в виде имени существующего в проекте файла. Например, в нашем проекте создаются или используются два каталога *bin* и *doc*. Тогда мы вправе указать эти имена в качестве фиктивных целей. Но мы должны гарантировать, что *make* пересобирает содержимое этих каталогов в случае устаревания, даже если каталоги сами по себе уже существуют.

В подобных случаях *make* позволяет указывать цели как фиктивные, невзирая на существование того или иного файла. Один способ сделать это заключается в добавлении следующей строки к нашему *make*-файлу (СЛАЙД 20). Теперь цель *clean* является зависимостью для специальной встроенной цели *.PHONY*.

Для примера с подкаталогами мы можем добавить к *make*-файлу еще несколько строк (СЛАЙД 20). Данное правило с целью *bin* в действительности создает каталог в

проекте. Однако, поскольку данная цель явно помечена как фиктивная, то *bin* никогда не устаревает. Утилита *make* копирует *circle* в подкаталог *bin*, даже если *bin* новее, чем *circle*.

В общем случае все фиктивные цели следует объявлять в явном виде, это может сохранить массу времени. Для тех целей, которые объявлены как фиктивные, *make* не беспокоится о поиске файлов с подходящими именами, как это происходит с неявными правилами по созданию файла с именем цели. Старомодный и чуть менее интуитивно понятный способ сделать то же самое показан в нижней части СЛАЙДА 20. Такого эффекта мы добьемся, если создадим одноименную цель без зависимостей и команд.

Цель *.PHONY* является предпочтительным вариантом хотя бы потому, что фиктивность задается явно, но мы, очевидно, можем предпочесть другие приемы, например, автоматическую генерацию правил и зависимостей.

Существуют и другие атрибуты, которые можно задавать определенным в *make*-файле целям. Это выполняется путем назначения их зависимостями для встроенной цели *.PHONY*. Некоторые из этих встроенных целей представлены на СЛАЙДЕ 21.

Существуют и другие атрибуты, о назначении которых можно прочитать в соответствующей литературе.

7.7 Введение в автоматическую сборку проекта

Ранее мы уже должны были осуществлять сборку некоторых нужных нам программ из их исходных текстов (в частности, *CUnit*). Общепринятый подход сводится к трем командам (СЛАЙД 22):

1. *./configure*
2. *make*
3. *make install*

Случается, что используются и иные команды, но обычно для сборки достаточно этих трех.

На первом шаге (это *shell*-скрипт) анализируется ОС и окружения с тем, чтобы узнать какие используются программы и библиотеки. Таким способом можно узнать, как наилучшим образом осуществлять сборку. На втором шаге программа, собственно говоря, и собирается программа, а на третьем – выполняется ее установка в работающую систему. Очень просто, не так ли?

Мы рассмотрим, какие действия и средства обеспечивают эти три шага. Для этого используется пакет (иногда используют термин «инфраструктура») GNU *Autotools*. Его компонентами являются:

1. *Autoconf* – его используют для генерации скрипта *configure*. Именно в нем определяется, например, какой компилятор использовать: *cc* или *gcc*.
2. *Automake* – автоматическое создание *make*-файлов на основе информации, созданной *Autoconf*.
3. *Libtool* – средство для создания разделяемых библиотек платформенно-независимым образом. *Libtool* очень тесно работает совместно с *Automake*, и для включения сборки разделяемых библиотек достаточно лишь внести небольшие изменения в шаблон *make*-файлов.
4. *Shtool* является одним большим *shell*-скриптом, который умеет выполнять порядка 15 команд разного назначения.

Мы сосредоточимся на двух первых пунктах, т.к. 3 и 4 стоят некоторым особняком.

7.8 Автоматическое создание файла конфигурирования

Пример кода программы, а также *make*-файл для демонстрации показан на СЛАЙДЕ 23. Пока ничего для нас нового.

Теперь к этой программе мы добавим *autoconf*. Сначала мы создадим файл “*configure.ac*”. Этот файл инструктирует *Autoconf*, как следует генерировать скрипт "*configure*".

Создание этого файла вручную возможно, но утомительно. В *Autoconf* имеется программа, которая автоматизирует процесс – *autoscan*. СЛАЙД 24.

Она просканирует наши «исходники» и создаст файл "*configure.scan*". Нам останется только переименовать файл. Почему этот файл не создается напрямую? Так сделано, чтобы нечаянно не перезаписать уже существующий "*configure.ac*". Теперь нужно запустить *Autoconf*, и на выходе мы получим подходящий скрипт *configure*.

Игнорируем все дополнительные файлы, созданные *Autoconf*, однако удалять их следует, т.к. они могут понадобиться в будущем.

Мы могли бы теперь запустить полученный скрипт, но нам ведь еще понадобится *make*-файл, который может быть получен по шаблону, содержащемуся в файле *Makefile.in*. В нашем случае мы сделаем это простым копированием файлов, но есть и иные способы.

Теперь можно запускать конфигурирование и сборку (СЛАЙД 25). Тем самым мы блестяще справились с созданием нашей первой *Autoconf*-программы. Однако пока еще ничего не сделало нашу программу более портируемой, чем раньше. Чтобы это проиллюстрировать, нам понадобится внести изменения в файлы, где должны быть указаны соответствующие директивы компилятору. Однако их использование невозможно, пока у нас нет констант для проверки. Ну и когда они появятся, мы разместим их в файле "*config.h*". Затем с помощью директивы *#include* файл должен быть подключен.

Но у нас нет этого файла, и он может быть сгенерирован *Autoconf*, если мы укажем ему на это. Это очень просто: запускается *autoheader*, а затем скрипт *configure* (СЛАЙД 26).

Далее *autoheader* сгенерирует "*config.h.in*". Затем "*config.h.in*" используется *configure* для генерации "*config.h*". Это общий подход *Autoconf*: часть программ генерирует "<*something*>*.in*", а затем "<*something*>*.in*" будет использовано *Autoconf* для генерации "<*something*>".

Давайте взглянем, какие константы мы получили в файле "*config*.h" (СЛАЙД 27).

Ну да, есть какие-то константы, но ни одна из них не полезна для портабельности. Кажется, наша программа оказалась слишком простой для этого. Усложним ее, чтобы мы получили что-то интересное в "*config.h*".

Сделаем нашу программу более сложной для того, чтобы могли протестировать переносимость с *Autoconf*. Теперь наша программа будет выводить количество секунд с начала Эпохи, используя вызов *gettimeofday* (СЛАЙД 28). Теперь будем ее «автоконфигурировать»! СЛАЙД 29.

Фрагмент содержимого файла "*config.h*" на СЛАЙДАХ 30-31.

Замечательно, не так ли?! Используя этот заголовочный файл, мы можем проверить несколько вещей. Однако для простоты, мы проверим только две из них: существует ли функция *gettimeofday* (используя *HAVE_GETTIMEOFDAY*), и существует ли заголовочный файл "*sys*/*time.h*" (используя *HAVE_SYS_TIME_H*). Чтобы эту информацию использовать, модифицируем наш «исходник» еще раз (СЛАЙД 32). Видно, что подключается требуемый заголовочный файл.

Мы оформили проверку времени отдельной функцией. Так мы можем группировать функции, у которых могут возникать проблемы с портируемостью, в одну секцию кода, и отрегулировать их в будущем при необходимости.

Функция пытается использовать системный вызов *gettimeofday*(), но она вернется назад к вызову *time*(), если *gettimeofday*() недоступен. Вызов *gettimeofday*() более полезен, т.к. предоставляет информацию о микросекундах, но *time*() мы все же будем использовать, если нет доступа к *gettimeofday*().

Не забудем про директиву *#include* "*config.h*".

Теперь нам снова понадобится *Autoconf*, для того чтобы проверить, не сделали ли мы какие-то изменения, приводящие к другим проблемам (СЛАЙД 33).

7.9 Автоматическое создание *make*-файла

Для этого выполняются три шага, очень похожие на создание файла конфигурации.

1. Создается файл с именем “*Makefile.am*”.
2. “*Makefile.am*” будет использован *Automake* для создания нового “*Makefile.in*”.
3. “*Makefile.in*” будет использован скриптом *configure* для создания “*Makefile*”.

Файл *Makefile.am* может быть создан вручную, его пример на СЛАЙДЕ 34.

Ведь просто, правда? Еще бы, мы же столько времени потратили на ручное написание *make*-файлов ранее.

Переменная *bin_PROGRAMS* указывает на имя нашей программы (или нескольких переменных). В нашем случае это “*hello*”. Со своей стороны *hello_SOURCES* указывает, какие исходные файлы используются для создания нашей “*hello*”. У нас такой файл один – “*hello.c*”, но при желании можно указывать несколько файлов, разделенных пробелами.

Существует множество дополнительных команд, которые мы можем использовать в “*Makefile.am*”: сборка библиотек, переход по файловой системе и т.д. В нашем случае достаточно только этих двух команд. За интересующими подробностями следует обратиться к руководству по *Automake*.

Итак, теперь мы можем автоматически создать *make*-файл. Для этого потребуется два шага:

1. *automake*. Исполнение этой команды вероятнее всего будет приводить к ошибкам. Следует проверить каждую из них и исправить.

2. *aclocal*. На 1-м шаге мы добавим несколько *Automake*-макросов в файл “*configure.ac*”. Это может обеспокоить *Autoconf*, т.к. он не будет в состоянии понять эти новые макросы. Но мы, к счастью, в состоянии успокоить *Autoconf*, создав файл “*aclocal.m4*”, содержащим определения для вновь добавленных макросов. Автоматически это за нас сделает *aclocal*.

Теперь запускаем automake и видим ошибки, которые следует исправить (СЛАЙД 34). В зависимости от используемой версии формулировки ошибок, поэтому нам придется разбираться с ними самостоятельно. По поводу некоторых из них можно сделать следующие рекомендации.

Первая ошибка исправляется легко: можно просто заменить *AC_CONFIG_HEADER* на *AM_CONFIG_HEADER* в файле “*configure.ac*”. *Automake* просто сообщает о том, что он предпочитает собственные версии макросов вместо того предлагаемых *Autoconf* (СЛАЙД 35).

Что касается ошибок с *PACKAGE* и *VERSION*, то эта информация может быть добавлена в "*configure.ac*" несколькими способами. Самое простое – добавить *AM_INIT_AUTOMAKE*(*hello*, 1.0) сразу после*AC_INIT*(…).

Это может выглядеть так, как будто мы взяли это макроопределение, как говорится, с потолока. Но фактически это один из нескольких стандартных макросов, о которых мы должны знать, используя *Automake*. Первый аргумент для *AM_INIT_AUTOMAKE* – это имя пакета, а второй аргумент – его версия. Лучше взять это себе на заметку!

Дальше идут сообщения о том, что пропущены какие-то файлы. Их можно создать вручную, например *NEWS*, *README*, *AUTHORS* и *ChangeLog*. Что касается других, то можно скопировать из любого известного места. В частности, у *Automake* есть каталог с такими файлами.

Например, мы можем взять “*install-sh*”, “*mkinstalldirs*” и “*missing*” из “/*usr*/*share*/*automake*/”. Это стандартные для *Autoconf* и *Automake* скрипты. Мы просто берем и копируем их в каталог нашего проекта. Их можно поискать утилитами *locate*, *find* или другими.

INSTALL – это документ, описывающий процесс установки программы. В нашем случае мы тоже скопировали его из “/*usr*/*share*/*automake*/”, но такой практике не нужно следовать повсеместно. Файл *COPYING* (файл лицензии) можно также скопировать из

“/*usr*/*share*/*automake*/”, если мы пишем GPL-программу, или придется переписать, в противном случае.

Следующая ошибка связана с багом взаимодействия *Autoconf* и *Automake*. Если открыть “*configure.ac*”, мы увидим следующую строку (СЛАЙД 35). Мы ее ранее модифицировали путем замены *AC_…* на *AM_…*

Измененный варианта на СЛАЙДЕ 35.

В любом случае придется внести корректировки.

Последние две ошибки подразумевают, какие строки кода хочет добавить Automake в файл “configure.ac”. И на самом деле эти строки будут добавлены макросами из “configure.ac” ранее. Так что эти сообщения можно игнорировать.

Теперь запустим по новой *Automake* (СЛАЙД 35). Все полученные ошибки следует исправлять!

Т.к. мы модифицировали “*configure.ac*”, то придется по новой запустить *autoconf*, но перед этим нужно запустить *aclocal*.

Как известно, *Autoconf* знает только о своих собственных макросах, которые начинаются с префикса *AC_…* Т.к. мы добавили макросы *Automake* к *Autoconf* (макросы с префиксами *AM_…*), то нам нужно где-то их взять и положить их туда, где их сможет найти *Autoconf*. Это за нам может сделать программа *aclocal* (берет их из Automake и кладет их в “*aclocal.m*4”). СЛАЙД 36.

Ничего захватывающего…

Теперь мы можем запустить *Autoconf*, скрипт *configure*, а затем сборку проекта. СЛАЙДЫ 37-38.

Литература и дополнительные источники к Теме 7

1. *Automake* - http://www.gnu.org/software/automake/
2. *Make* - http://www.gnu.org/software/make/
3. Makefiles. A tutorial by example - http://mrbook.org/blog/tutorials/make/
4. Managing Projects with GNU *Make*, Third Edition By Robert Mecklenburg - http://oreilly.com/openbook/make3/book/index.csp
5. The GNU *configure* and build system - http://airs.com/ian/configure/
6. Using *make* and writing Makefiles - http://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html
7. Работа с утилитой *make* - http://www.net4me.net/docs/pdf/Linux/make.pdf
8. GNU *M*4 - http://www.gnu.org/software/m4/