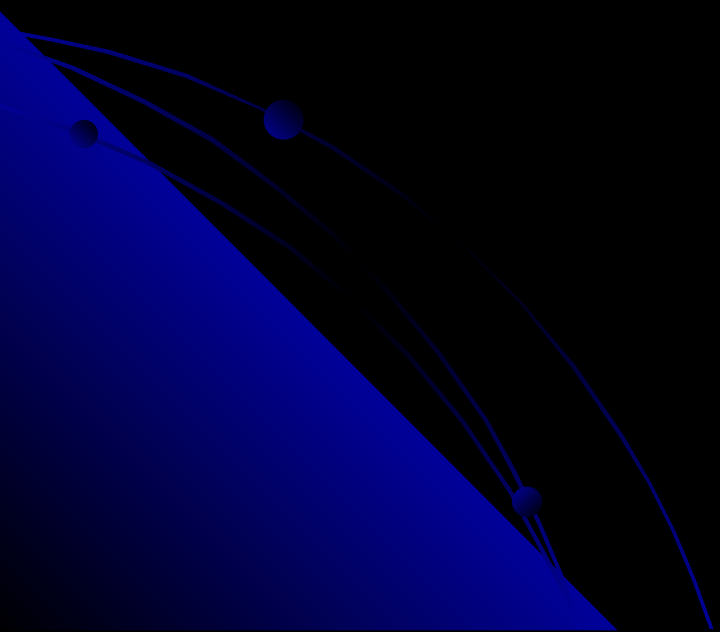


MAKE и другие утилиты



Make

- Утилита `make` автоматически определяет какие части большой программы должны быть перекомпилированы, и выполняет необходимые для этого действия.
- В примерах будут фигурировать программы на языке Си, однако, можно использовать `make` с любым языком программирования для которого имеется компилятор, работающий из командной строки. На самом деле, область применения `make` не ограничивается только сборкой программ. Вы можете использовать ее для решения любых задач, где одни файлы должны автоматически обновляться при изменении других файлов.
- Перед тем, как использовать `make`, вы должны создать так называемый **make-файл (makefile)**, который будет описывать зависимости между файлами вашей программы, и содержать команды для обновления этих файлов. Как правило, исполняемый файл программы зависит от объектных файлов, которые, в свою очередь, получаются в результате компиляции соответствующих файлов с исходными текстами.

Make

- После того, как нужный make-файл создан, простой команды :
- `$make`
- будет достаточно для выполнения всех необходимых перекомпиляций если какие-либо из исходных файлов программы были изменены. Используя информацию из make-файла, и, зная время последней модификации файлов, утилита make решает, каких из файлов должны быть обновлены. Для каждого из этих файлов будут выполнены указанные в make-файле команды.
- При вызове make, в командной строке могут быть заданы параметры, указывающие, какие файлы следует перекомпилировать и каким образом это делать.

Знакомство с make-файлами (makefiles)

- Для работы с утилитой make, вам понадобится так называемый **make-файл (makefile)**, который будет содержать описание требуемых действий. Как правило, make-файл описывает, каким образом нужно компилировать и компоновать программу.
- Обсудим простой make-файл, который описывает, как скомпилировать и скомпоновать программу - текстовый редактор. текстовый редактор будет состоять из восьми файлов с исходными текстами на языке Си и трех заголовочных файлов. Make-файл также может инструктировать make, как выполнять те или иные действия, когда явно будет затребовано их выполнение (например, удалить определенные файлы в ответ на команду "очистка").
- При компиляции текстового редактора, любой файл с исходным текстом, который был модифицирован, должен быть откомпилирован заново. Если был модифицирован какой-либо из заголовочных файлов, то, во избежание проблем, должны быть перекомпилированы все исходные файлы, которые включали в себя этот заголовочный файл.

Знакомство с make-файлами (makefiles)

- Каждая компиляция исходного файла породит новую версию соответствующего ему объектного файла. И, наконец, если какие-либо из исходных файлов были перекомпилированы, то все объектные файлы (как "новые", так и оставшиеся от предыдущих компиляций) должны быть заново скомпонованы для получения новой версии исполняемого файла нашего текстового редактора.
- Простой make-файл состоит из "правил" (rules) следующего вида:
 - *цель ... : пререквизит ... команда*
 - Обычно, **цель (target)** представляет собой имя файла, который генерируется в процессе работы утилиты make. Примером могут служить объектные и исполняемый файлы собираемой программы. Цель также может быть именем некоторого действия, которое нужно выполнить (например, 'clean' - очистить).

Знакомство с make-файлами (makefiles)

- **Пререквизит (prerequisite)** - это файл, который используется как исходные данные для порождения цели. Очень часто цель зависит сразу от нескольких файлов.
- **Команда** - это действие, выполняемое утилитой make. В правиле может содержаться несколько команд - каждая на свое собственной строке. **Важное замечание:** строки, содержащие команды обязательно должны начинаться с символа табуляции! Это - "грабли", на которые наступают многие начинающие пользователи.
- Обычно, команды находятся в правилах с пререквизитами и служат для создания файла-цели, если какой-нибудь из пререквизитов был модифицирован. Однако, правило, имеющее команды, не обязательно должно иметь пререквизиты. Например, правило с целью `clean` ("очистка"), содержащее команды удаления, может не иметь пререквизитов.

Знакомство с make-файлами (makefiles)

- **Правило (rule)** описывает, когда и каким образом следует обновлять файлы, указанные в нем в качестве цели. Для создания или обновления цели, make исполняет указанные в правиле команды, используя пререквизиты в качестве исходных данных. Правило также может описывать, каким образом должно выполняться некоторое действие.
- Помимо правил, make-файл может содержать и другие конструкции, однако, простой make-файл может состоять и из одних лишь правил. Правила могут выглядеть более сложными, чем приведенный выше шаблон, однако все они более или менее соответствуют ему по структуре.

Пример простого make-файла

- Вот пример простого make-файла, в котором описывается, что исполняемый файл `edit` зависит от восьми объектных файлов, которые, в свою очередь, зависят от восьми соответствующих исходных файлов и трех заголовочных файлов.
- В данном примере, заголовочный файл `defs.h` включается во все файлы с исходным текстом. Заголовочный файл `command.h` включается только в те исходные файлы, которые относятся к командам редактирования, а файл `buffer.h` - только в "низкоуровневые" файлы, непосредственно оперирующие буфером редактирования.
- `edit : main.o kbd.o command.o display.o \
insert.o search.o files.o utils.o`
- `cc -o edit main.o kbd.o command.o display.o \
insert.o search.o files.o utils.o`
- `main.o : main.c defs.h`
- `cc -c main.c`

Пример простого make-файла

- 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

Пример простого make-файла

- clean :
- `rm edit main.o kbd.o command.o display.o \`
- `insert.o search.o files.o utils.o`
- Для повышения удобочитаемости, мы разбили длинные строки на две части с помощью символа обратной косой черты, за которым следует перевод строки.
- Для того, чтобы с помощью этого make-файла создать исполняемый файл `edit`, наберите:
- `$make`
- Для того, чтобы удалить исполняемый и объектные файлы из директории проекта, наберите:
- `make clean`
- В приведенном примере, целями, в частности, являются объектные файлы `main.o` и `kbd.o`, а также исполняемый файл `edit`. К пререквизитам относятся такие файлы, как `main.c` и `defs.h`. Каждый объектный файл, фактически, является одновременно и целью и пререквизитом. Примерами команд могут служить `cc -c main.c` и `cc -c kbd.c`.

Пример простого make-файла

- В случае, если цель является файлом, этот файл должен быть перекомпилирован или перекомпонован всякий раз, когда был изменен какой-либо из его пререквизитов. Кроме того, любые пререквизиты, которые сами генерируются автоматически, должны быть обновлены первыми. В нашем примере, исполняемый файл ``edit'` зависит от восьми объектных файлов; объектный файл ``main.o'` зависит от исходного файла ``main.c'` и заголовочного файла ``defs.h'`.
- За каждой строкой, содержащей цель и пререквизиты, следует строка с командой. Эти команды указывают, каким образом надо обновлять целевой файл. В начале каждой строки, содержащей команду, должен находиться символ табуляции. Именно наличие символа табуляции является признаком, по которому make отличает строки с командами от прочих строк make-файла. Утилита make просто исполняет указанные в правиле команды, если цель нуждается в обновлении.

Пример простого make-файла

- Цель `clean` является не файлом, а именем действия. Поскольку, при обычной сборке программы это действие не требуется, цель `clean` не является пререквизитом какого-либо из правил. Следовательно, make не будет "трогать" это правило, пока вы специально об этом не попросите. Заметьте, что это правило не только не является пререквизитом, но и само не содержит каких-либо пререквизитов. Таким образом, единственное предназначение данного правила - выполнение указанных в нем команд. Цели, которые являются не файлами, а именами действий называются *абстрактными целями (phony targets)*.

Как make обрабатывает make-файл

- По умолчанию, make начинает свою работу с первой встреченной цели (кроме целей, чье имя начинается с символа `.`). Эта цель будет являться **главной целью по умолчанию (default goal)**. Главная цель (**goal**) - это цель, которую стремится достичь make в качестве результата своей работы.
- В примере главная цель заключалась в обновлении исполняемого файла `edit`, поэтому мы поместили данное правило в начало make-файла.
- Таким образом make читает make-файл из текущей директории и начинает его обработку с первого встреченного правила. В нашем примере это правило обеспечивает перекомпиляцию исполняемого файла `edit`. Однако, прежде чем make сможет полностью обработать это правило, ей нужно обработать правила для всех файлов, от которых зависит `edit`. В данном случае - от всех объектных файлов программы. Каждый из этих объектных файлов обрабатывается согласно своему собственному правилу.

Как make обрабатывает make-файл

- Эти правила говорят, что каждый файл с расширением `.o` (объектный файл) получается в результате компиляции соответствующего ему исходного файла. Такая компиляция должна быть выполнена, если исходный файл или какой-либо из заголовочных файлов, перечисленных в качестве пререквизитов, являются "более новыми", чем объектный файл, либо объектного файла вообще не существует.
- Другие правила обрабатываются потому, что их цели прямо или косвенно являются пререквизитами для главной цели. Если какое-либо правило никоим образом не "связано" с главной целью (то есть ни прямо, ни косвенно не являются его пререквизитом), то это правило не обрабатывается. Чтобы задействовать такие правила, придется явно указать `make` на необходимость их обработки (подобным, например, образом: `make clean`).

Как make обрабатывает make-файл

- Перед перекомпиляцией объектного файла, make рассматривает необходимость обновления его пререквизитов, в данном случае - файла с исходным текстом и заголовочных файлов. В нашем make-файле не содержится никаких инструкций по обновлению этих файлов - файлы с расширениями `.c` и `.h` не являются целями каких-либо правил. Таким образом, утилита make не предпринимает никаких действий с этими файлами.
- После перекомпиляции объектных файлов, которые нуждаются в этом, make принимает решение - нужно ли перекомпоновывать файл `edit`. Это нужно делать, если файла `edit` не существует или какой-нибудь из объектных файлов по сравнению с ним является более "свежим". Если какой-либо из объектных файлов только что был откомпилирован заново, то он будет "моложе", чем файл `edit`. Соответственно, файл `edit` будет перекомпонован.

Как make обрабатывает make-файл

- Так, если мы модифицируем файл `insert.c` и запустим make, этот файл будет скомпилирован заново для обновления объектного файла `insert.o`, и, затем, файл `edit` будет перекомпилирован. Если мы изменим файл `command.h` и запустим make, то будут перекомпилированы объектные файлы `kbd.o`, `command.o` и `files.o`, а затем исполняемый файл `edit` будет скомпилирован заново.
- В приведенном выше примере, в правиле для `edit` нам дважды пришлось перечислять список объектных файлов программы:
- ```
edit : main.o kbd.o command.o display.o \ insert.o search.o
files.o utils.o
```
- ```
cc -o edit main.o kbd.o command.o display.o \  
insert.o search.o files.o utils.o
```
- Подобное дублирование чревато ошибками. При добавлении в проект нового объектного файла, можно добавить его в один список и забыть про другой. Мы можем устранить подобный риск, и, одновременно, упростить make-файл, используя переменные.

Упрощение make-файла с помощью переменных

- **Переменные (variables)** позволяют, один раз определив текстовую строку, затем использовать ее многократно в нужных местах.
- Обычной практикой при построении make-файлов является использование переменной с именем `objects`, `OBJECTS`, `objs`, `OBJS`, `obj`, или `OBJ`, которая содержит список всех объектных файлов программы. Мы могли бы определить подобную переменную с именем `objects` таким образом:
 - `objects = main.o kbd.o command.o display.o \`
 - `insert.o search.o files.o utils.o`
- Далее, всякий раз, когда нам нужен будет список объектных файлов, мы можем использовать значение этой переменной с помощью записи `$(objects)`.
- Вот как будет выглядеть наш простой пример с использованием переменной для хранения списка объектных файлов:

Упрощение make-файла с ПОМОЩЬЮ ПЕРЕМЕННЫХ

- `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`

Упрощение make-файла с помощью переменных

- 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)
- На самом деле, нет необходимости явного указания команд компиляции отдельно для каждого из исходных файлов. Утилита make сама может "догадаться" об использовании нужных команд, поскольку у нее имеется, так называемое, **неявное правило (implicit rule)** для обновления файлов с расширением `.o` из файлов с расширением `.c`, с помощью команды `cc -c`. Таким образом, можно убрать явное указание команд компиляции из правил, описывающих построение объектных файлов.

Упрощение make-файла с помощью переменных

- Когда файл с расширением `.c` автоматически используется подобным образом, он также автоматически добавляется в список пререквизитов "своего" объектного файла. Таким образом, мы вполне можем убрать файлы с расширением `.c` из списков пререквизитов объектных файлов.
- Вот новый вариант нашего примера, в который были внесены оба описанных выше изменения, а также используется переменная `objects`:
- `objects = main.o kbd.o command.o display.o \`
- `insert.o search.o files.o utils.o`
- `edit : $(objects)`
- `cc -o edit $(objects)`
- `main.o : defs.h`
- `kbd.o : defs.h command.h`
- `command.o : defs.h command.h`
- `display.o : defs.h buffer.h`

Упрощение make-файла с помощью переменных

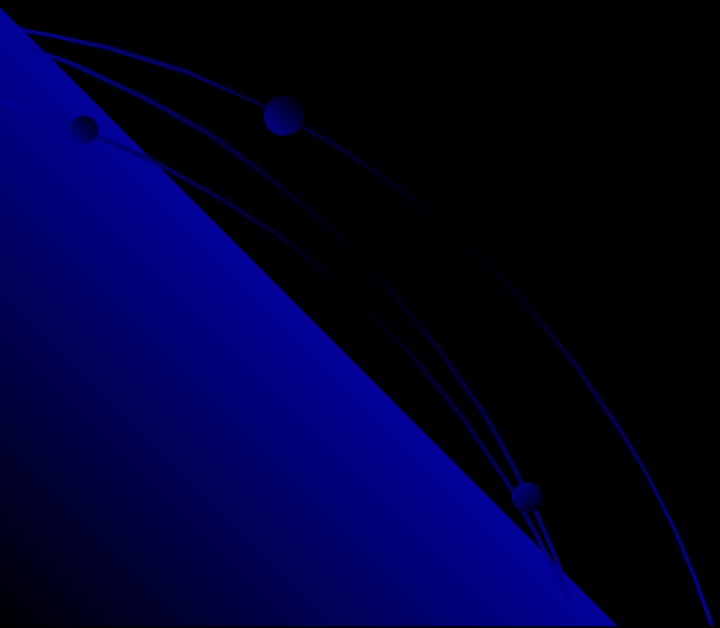
- insert.o : defs.h buffer.h
- search.o : defs.h buffer.h
- files.o : defs.h buffer.h command.h
- utils.o : defs.h.
- PHONY : clean
- clean :
 - -rm edit \$(objects)
- **Правило для очистки каталога**
- Компиляция программы - не единственная вещь, для которой вы, возможно, захотите написать правила. Часто, в make-файле указывается, каким образом можно выполнить некоторые другие действия, не относящиеся к компиляции программы. Таким действием, например, может быть удаление все объектных и исполняемых файлов программы для очистки каталога.

Упрощение make-файла с помощью переменных

- Вот как можно было бы написать правило для очистки каталога в нашем проекте текстового редактора:
- clean:
- `rm edit $(objects)`
- На практике, скорее всего, мы бы записали это правило чуть более сложным способом, предполагающим возможность непредвиденных ситуаций:
- `.PHONY : clean`
- clean :
- `-rm edit $(objects)`
- Такая запись предотвратит возможную путаницу если, вдруг, в каталоге будет находится файл с именем `clean`, а также позволит make продолжить работу, даже если команда `rm` завершится с ошибкой.

Упрощение make-файла с помощью переменных

- Подобное правило не следует помещать в начало make-файла, поскольку мы не хотим, чтобы оно запускалось "по умолчанию"! В нашем примере, мы помещаем данное правило в конец make-файла, чтобы главной целью по умолчанию оставалась сборка файла `edit`.
- Поскольку `clean` не является пререквизитом цели `edit`, это правило не будет выполняться, если вызывать `make` без аргументов. Для запуска данного правила, нужно будет набрать `make clean`.



Конфигурирование

- Факт создания авторами исходных кодов представляет собой чисто технический интерес для *портирования* программ. Свободное программное обеспечение, разработанное для UNIX®-систем, может быть использовано во всех существующих системах UNIX® (как свободных, так и собственных) с незначительными изменениями или вообще без изменений. Для этого требуется сконфигурировать ПО непосредственно перед его компиляцией.
- Существует несколько систем конфигурирования. Вы должны использовать ту, которую требует автор программного обеспечения (а иногда и несколько). Обычно вы можете:
 - Использовать AutoConf, если в родительском каталоге дистрибутива имеется файл с именем configure.
 - Использовать imake, если в родительском каталоге дистрибутива имеется файл с именем Imakefile.
 - Запустить shell-скрипт (например, install.sh) согласно содержимому файла INSTALL (или файла README).

Autoconf

- Программа AutoConf используется для корректной настройки ПО. Она создаст необходимые для компиляции файлы (например, Makefile) и иногда изменяет непосредственно сами исходные тексты (например, при помощи файла config.h.in).
- Принцип AutoConf прост:
- Разработчик ПО знает, какие проверки необходимы для настройки его программы (например: «какую версию этой библиотеки вы используете?»). Он записывает их в файл с именем configure.in, используя определённый синтаксис.
- Он запускает AutoConf, которая создаст из файла configure.in конфигурационный скрипт с именем configure. Этот скрипт выполняет тесты, необходимые при настройке программы.
- Конечный пользователь запускает скрипт и AutoConf настраивает всё, что необходимо для компиляции.

Autoconf

- Пример использования AutoConf:
- `$./configure`
- loading cache `./config.cache`
- checking for gcc... Gcc
- checking whether the C compiler (`gcc`) works... Yes
- checking whether the C compiler (`gcc`) is a cross-compiler... No
- checking whether we are using GNU C... Yes
- checking whether gcc accepts `-g`... yes
- checking how to run the C preprocessor... `gcc -E`
- checking for ANSI C header files... yes
- checking for `unistd.h`... yes
- checking for working `const`... yes
- updating cache `./config.cache`
- creating `./config.status`
- creating `lib/Makefile`

Autoconf

- creating src/Makefile
- creating Makefile
- Для лучшего управления информацией, генерируемой **configure**, из командной строки или через переменные окружения могут быть добавлены некоторые опции. Пример:
- `$./configure --with-gcc --prefix=/opt/GNU`
- или (при помощи bash):
- `$ export CC=`which gcc``
- `$ export CFLAGS=-O2$`
- `./configure --with-gcc`
- или:
- `$ CC=gcc CFLAGS=-O2`
- `./configure`

Autoconf

- Обычны ошибки подобного вида: `configure: error: Cannot find library guile.`
- Это означает, что скрипт `configure` не смог найти библиотеку (в этой примере библиотеку `guile`). Принцип заключается в том, что скрипт `configure` компилирует небольшую тестовую программу, использующую эту библиотеку. Если компиляция этой программы завершится неудачей, то невозможно будет откомпилировать и весь программный пакет. Затем возникает ошибка.
- Причину ошибки ищите в конце файла `config.log`, содержащего отчёт обо всех этапах конфигурирования. Компилятор `C` выводит довольно чёткие сообщения об ошибках. Обычно это поможет вам при решении возникающих проблем.
- Проверьте, правильно ли установлена названная библиотека. Если это не так - установите её (из исходных кодов или в виде откомпилированного бинарного файла) и ещё раз запустите `configure`.

Autoconf

- Эффективным способом проверки является поиск файла, содержащего символы библиотеки, коим всегда будет `lib<имя>.so`. Например,
- `$ find / -name 'libguile*' или, как вариант:`
- `$ locate libguile`
- Проверьте, доступна ли библиотека для компилятора. Это означает, что она находится в одном из каталогов: `/usr/lib`, `/lib`, `/usr/X11R6/lib` (или в одном из тех, что определены переменной окружения `LD_LIBRARY_PATH`). Проверьте, является ли этот файл библиотекой, набрав **file libguile.so**.
- Проверьте, правильно ли установлены заголовочные файлы библиотеки (обычно в `/usr/include`, `/usr/local/include` или `/usr/X11R6/include`). Если вы не знаете, какие файлы заголовков вам нужны, проверьте, установлена ли у вас development-версия (для разработки) требуемой библиотеки (например, `libgtk+2.0-devel` вместо `libgtk+2.0`). Версия библиотеки для разработки предоставляет файлы «include», необходимые для компиляции ПО, использующего эту библиотеку.

Autoconf

- Проверьте, достаточно ли у вас свободного дискового пространства (для скрипта `configure` требуется некоторый объём для временных файлов). Воспользуйтесь командой `df -h` для вывода списка разделов вашей системы и обратите внимание на заполненные или почти заполненные разделы.
- Если вы не понимаете сообщения, сохранённые в файле `config.log`, не стесняйтесь попросить помощи у сообщества свободного ПО.
- Кроме того, проверьте, существует ли библиотека, даже если `configure` говорит, что её нет. В этом случае, вероятно, повреждена переменная окружения `LD_LIBRARY_PATH`!

Создание скриптов configure

- Для создания скриптов Autoconf требует наличия программы GNU m4. Он пользуется возможностями, которых нет в некоторых UNIX-версиях программы m4. Он также превышает внутренние ограничения некоторых версий m4, включая GNU m4 версии 1.0. Вам необходимо использовать версию 1.1 (или более позднюю) программы GNU m4. Версии 1.3 и более поздние будут работать гораздо быстрее, чем версии 1.1 или 1.2.
- Скрипты конфигурации, создаваемые Autoconf, по принятым соглашениям называются configure. При запуске configure создает несколько файлов, заменяя в них параметры конфигурации на соответствующие системе значения. configure создает следующие файлы:
 - один или несколько файлов `Makefile`, по одному на каждый подкаталог пакета;
 - если задано, создается заголовочный файл для языка C, имя которого можно задать при создании скрипта и который содержит директивы `#define`;

Создание скриптов configure

- скрипт командного процессора с именем ``config.status'`, который при запуске заново создаст вышеперечисленные файлы;
- скрипт командного процессора с именем ``config.cache'`, который сохраняет результаты выполнения многих тестов;
- файл с именем ``config.log'`, который содержит все сообщения, выданные компиляторами. Этот файл может использоваться при отладке, если `configure` работает неправильно.
- Для того, чтобы с помощью `Autosconf` создать скрипт `configure`, вам необходимо написать входной файл с именем ``configure.in'` и выполнить команду `autosconf`. Если вы напишите собственный код тестирования возможностей системы, в дополнение к поставляемым с `Autosconf`, то вам придется записать его в файлы с именами ``aclocal.m4'` и ``acsite.m4'`. Если вы используете заголовочный файл, который содержит директивы `#define`, то вы также должны создать файл ``acconfig.h'`, и вы сможете распространять с пакетом созданный с помощью `Autosconf` файл ``config.h.in'`.

Automake

- Automake читает файл `Makefile.am` и создает на его основе файл `Makefile.in`. Специальные макросы и цели, определенные в `Makefile.am`, заставляют Automake генерировать более специализированный код; например, макроопределение `bin_PROGRAMS` заставит создать цели для компиляции и компоновки программ.
- Давайте предположим, что мы только что закончили писать `zardoz` --- программу, от которой у всех кружится голова. Вы использовали `Autosconf` для обеспечения переносимости, но ваш файл `Makefile.in` был написан бессистемно. Вы же хотите сделать его пуленепробиваемым, и поэтому решаете использовать Automake.
- Сначала вам необходимо обновить ваш файл `configure.in`, чтобы вставить в него команды, которые необходимы для работы automake. Проще всего для этого добавить строку `AM_INIT_AUTOMAKE` сразу после `AC_INIT`:

Automake

- AM_INIT_AUTOMAKE(zardoz, 1.0)
- Поскольку ваша программа не имеет никаких осложняющих факторов (например, она не использует gettext и не будет создавать разделяемые библиотеки), то первая стадия на этом и заканчивается. Это легко!
- Теперь вы должны заново создать файл `configure`. Но для этого нужно сказать autosconf, где найти новые макросы, которые вы использовали. Для создания файла `aclocal.m4` удобнее всего будет использовать программу aclocal. Программа aclocal позволяет вам поместить ваши собственные макросы в файл `acinclude.m4`, так что для сохранения вашей работы просто переименуйте свой файл с макросами, а уж затем запускайте программу aclocal:

Automake

- `mv aclocal.m4 acinclude.m4`
- `aclocal`
- `autoconf`
- Теперь пришло время написать свой собственный файл ``Makefile.am'` для программы `zardoz`. Поскольку `zardoz` является пользовательской программой, то вам хочется установить ее туда, где располагаются другие пользовательские программы. Вдобавок, `zardoz` содержит в комплекте документацию в формате `Texinfo`. Ваш скрипт ``configure.in'` использует `AC_REPLACE_FUNCS`, так что вам необходимо скомпоновать программу с ``@LIBOBJS@'`. Вот что вам необходимо написать в ``Makefile.am'`.
- `bin_PROGRAMS = zardoz`
- `zardoz_SOURCES = main.c head.c float.c vortex9.c gun.c`
- `zardoz_LDADD = @LIBOBJS@`
- `info_TEXINFOS = zardoz.texi`

GNU Autotools - инфраструктура для сборки программ

- Ранее мы обсуждали два "основных" пакета: Automake и Autoconf. Здесь мы кратко обсудим оставшихся два пакета: Libtool и Shtool.
- **Libtool -- разделяемые библиотеки**
- Необходимость появления пакета Libtool стала очевидной после того, как доказали свою жизнеспособность пакеты Autoconf и Automake. Разработчики проекта GNU постоянно вставали перед проблемой создания разделяемых библиотек (shared libraries) на различных платформах.
- К сожалению, практически каждый из производителей UNIX-совместимых операционных систем изобретал свой собственный способ создания разделяемых библиотек..
- После того, как несколько существующих к тому времени пакетов были признаны неудовлетворительными, началась работа над пакетом Libtool, который и является на текущий момент самым функциональным.

Libtool -- разделяемые библиотеки

- Для иллюстрации использования Libtool мы будем использовать многострадальную программу Hello, world. Надо разбить эту программу на два файла, в одном из которых находится наша функция `print_hello()`. Нашей задачей будет превратить этот файл в полноценную разделяемую библиотеку `libhello`.
- Создадим основной каталог проекта, а в нем подкаталог `src/`. Создадим `src/hello.c`, содержащий коротенькую функцию `print_hello()`:
 - `=== src/hello.c ===`
 - `#include <stdio.h>`
 - `#include "hello.h"`
 - `void print_hello(void) {`
 - `printf("Hello, world!\n");`
 - `}`
 - `=== src/hello.c ===`
- Еще создадим файл `src/main.c` с функцией `main()`, которая использует функцию `print_hello()`:

Libtool -- разделяемые библиотеки

- `=== src/main.c ===`
- `#include "hello.h"`
- `int main(void) {`
- `print_hello();`
- `exit(0);}`
- `=== src/main.c ===`
- Еще создадим заголовочный файл `src/hello.h`, содержащий объявление функции `print_hello()`:
- `=== src/hello.h ===`
- `#ifndef HELLO_H`
- `#define HELLO_H`
- `void print_hello(void);`
- `#endif`
- `=== src/hello.h ===`

Libtool -- разделяемые библиотеки

- Два файла `hello.c` и `main.c` будут компилироваться в объектные файлы `hello.o` и `main.o`, соответственно, которые будут слинкованы в исполняемый файл, который называется просто `hello`.
- Вообще, Libtool создает так называемые Libtool-библиотеки -- это целый набор файлов. Главный файл, содержащий описание библиотеки, имеет расширение `.la`. Объектные файлы, используемые при компиляции, а также результирующие статические и динамические библиотеки, помещаются в подкаталог `.libs/`. При установке Libtool автоматически обрабатывает эти файлы, помещая их в нужные каталоги.
- Заметьте, что некоторые устаревшие операционные системы не поддерживают динамических библиотек — Libtool автоматически будет создавать при этом только статические библиотеки.

Libtool -- разделяемые библиотеки

- Для того, чтобы добавить поддержку Libtool в вашем проекте, используется макрос AC_PROG_LIBTOOL в файле configure.in:
- `=== configure.in ===`
- `AC_INIT([hello.c])`
- `AM_INIT_AUTOMAKE([hello], [0.3])`
- `AC_PROG_CC`
- `AC_PROG_LIBTOOL`
- `AM_CONFIG_HEADER(config.h)`
- `AC_OUTPUT([Makefile src/Makefile])`
- `=== configure.in ===`
- Этот макрос не входит в стандартную поставку Autoconf, поэтому его нужно добавить в пакет с помощью программы `aclocal`. Кроме того, для поддержки Libtool требуется еще несколько файлов, которые можно автоматически установить с помощью программы `automake`:

Libtool -- разделяемые библиотеки

- \$ aclocal
- \$ autoconf
- \$automake -ac
- В дистрибутив добавятся следующие файлы: config.guess, config.sub, ltmain.sh, ltconfig. В процессе работы скрипта configure будет создан файл libtool, который и будет обеспечивать всю поддержку сборки.
- Для создания Libtool-библиотек используется основная переменная Automake `_LTLIBRARIES`. Для того, чтобы создать разделяемую библиотеку libhello, напишем в src/Makefile.am:
 - `=== src/Makefile.am ===`
 - `lib_LTLIBRARIES = libhello.la`
 - `libhello_la_SOURCES = hello.c hello.h`
 - `=== src/Makefile.am ===`
- Эта запись означает, что будет создана Libtool-библиотека, которая называется libhello.la и компилируется из двух исходных файлов: hello.c и hello.h.

Libtool -- разделяемые библиотеки

- Выполним программы `automake`, `autoconf`, затем выполним скрипт `./configure`. Теперь попробуем собрать библиотеку (несколько строчек из выдачи программы `make` удалено для краткости):
 - `$cd src/`
 - `$make libhello.la`
- Заглянув в каталог `.libs/`, можно увидеть там как статическую библиотеку с расширением `.a`, так и динамические библиотеки с расширением `.so`, со всеми необходимыми символьными ссылками.
- **Сборка программ с Libtool-библиотеками**
- Довольно часто ваш пакет состоит не только из разделяемой библиотеки, но также и из набора программ, использующих эту библиотеку. Libtool автоматически обрабатывает этот распространенный случай.
- Давайте сделаем так, чтобы программа `hello` использовала разделяемую библиотеку `libhello`:

Сборка программ с Libtool-библиотеками

- `=== src/Makefile.am ===`
- `bin_PROGRAMS=hello`
- `hello_SOURCES=main.c`
- `hello_LIBADD = $(top_builddir)/src/libhello.la`
- `=== src/Makefile.am ===`
- Переменная `$(top_builddir)` содержит каталог сборки верхнего уровня. Есть еще похожая переменная `$(top_srcdir)`, содержащая верхний уровень каталога с исходными текстами. Если вам нужно обратиться к собираемому объекту, используйте первую из них; в противном случае, чтобы обратиться к объекту, существующему в дистрибутиве (например, файлу данных или каталогу с заголовочными файлами), используйте вторую переменную.
- Итак, мы опять обновляем файлы `Makefile.in` и `Makefile`, собираем библиотеку `libhello.la`, затем собираем программу `hello`:

Сборка программ с Libtool-библиотеками

- `$ cd src/`
- `$ make libhello.la`
- `$ make hello`
- Если мы посмотрим на файл `hello`, то увидим, что он на самом деле представляет собой `shell`-скрипт. Настоящий исполняемый файл находится в подкаталоге `.libs/`. Он будет установлен как следует, при выполнении `make install`.
- Впрочем, этот скрипт можно запустить прямо из каталога сборки, как если бы это была обычная программа: `$./hello Hello, world!`

Дополнительные возможности

- Если вы хотите отладить вашу программу, то не сможете сделать это простым наивным методом:
- `$gdb hello`
- `"/home/hello/src/hello": not in executable format: File format not recognized`
- Libtool обеспечивает специальный режим для отладки:
- `$ libtool gdb hello This GDB was configured as "i686-pc-linux-gnu"...`
- `(gdb) run`
- `Starting program: /home/hello/src/.libs/lt-hello`
- `Hello, world!`
- `Program exited normally.`

Shtool

- Shtool является одним большим shell-скриптом, который умеет выполнять полтора десятка подкоманд разного назначения. Основная причина использования этого скрипта -- невероятная переносимость. Shell-код, который используется внутри этих скриптов, работает на огромном количестве командных интерпретаторов различных операционных систем, каждая из которых, как всегда, имеет свои, скажем так, особенности.
- Вообще, в системах сборки пакетов, использующих Automake и Autoconf, почти нет места, где можно было бы использовать Shtool: разве что можно использовать подкоманды mkdir и install. Зачем бы это было нужно? -- спросите вы, ведь команды mkdir и install есть на всякой системе? Нет. Например, команда mkdir имеет флаг -p только в GNU-версии и еще в некоторых. Ощутимое количество коммерческих операционных систем не умеют создавать каталоги, только если все его родительские каталоги уже существуют. Некоторые программы install также не поддерживают ряда полезных ключей.

Shtool

- Скрипты shtool проявляют себя в полной мере, когда их используют для различных дополнительных, нестандартных целей. В этой статье мы лишь кратко опишем подкоманды, которые обеспечивает Shtool, чтобы вы представляли себе, что можно не реализовывать снова и снова, а взять уже готовое. Подробности, как обычно, можно найти на великолепной странице руководства к Shtool.
- Вывод на экран:
- - echo: выдает на экран строку, в которой можно использовать служебные конструкции; поддерживает ключ -n, который умеет не всякая программа echo;
- - mdate: печатает в указанном формате время последней модификации файла или каталога;
- - table: печатает список значений в виде таблицы;
- - prog: рисует красивый "пропеллер" во время выполнения длинной операции;

Shtool

- Работа с файлами:
- - `move`: позволяет переименовывать файлы с поддержкой маски: например, `shtool move -e *.txt *.asc` переименует все файлы с расширением `.txt` в `.asc`; такой возможности обычно не хватает многим, привыкшим к досовской команде `rename`;
- - `install`: устанавливает программу, скрипт или обычный файл; обеспечивает все флаги BSD-версии программы `install`, даже если системная программа `install` их не поддерживает или вообще не существует;
- - `mkdir`: создает каталог; позволяет также создать все необходимые родительские каталоги -- эта возможность часто отсутствует в системных программах `mkdir`;
- - `mkln`: создает ссылку (жесткую или символическую) на файл; при создании символической ссылки старается, чтобы ссылка была относительной;

Shtool

- - `mkshadow`: создает копию дерева исходных текстов с помощью символьных ссылок; полезно, если хочется внести несколько исправлений в существующее дерево, чтобы потом создать `diff`-файл;
- - `fixperm`: "исправляет" права доступа к файлам перед созданием дистрибутива; обычно при этом устанавливаются недостающие права доступа и выполнения для "группы" и для "всех остальных";
- - `tarball`: создает правильный `tar`-файл с дистрибутивом. Этот `tar`-файл будет распаковываться в отдельный подкаталог; список файлов в этом `tar`-файле будет отсортирован, чтобы можно было пользоваться командой `tar tvf`;
- Системные возможности:
- - `guessos`: определяет операционную систему и архитектуру; обычно лучше использовать стандартный GNU-скрипт `config.guess`;
- - `arx`: расширенный вариант команды `ar`;

Shtool

- - slo: специальная обертка к системному компоновщику, позволяющая удобно работать со специальными динамическими библиотеками;
- - scpp: специальная обертка к препроцессору языка C, облегчающая написание библиотек на ANSI C;
- - version: обеспечивает создание файлов описания версии пакета для нескольких различных языков программирования;
- - path: утилита для работы с переменной окружения \$PATH.
- С помощью программы shtoolize можно создать скрипт shtool, содержащий необходимое подмножество подкоманд.
- Учтите, лицензия на все эти пакеты позволяет использовать их даже в коммерческих программных продуктах.
- Документация, а также списки рассылки и группы новостей помогут вам добиться эффективного использования Autotools в повседневной практике.

Shtool

- **Ссылки**
- Основная страница проекта Libtool:
<http://www.gnu.org/software/libtool/libtool.html>
- Основная страница проекта Shtool:
<http://www.gnu.org/software/shtool/shtool.html>
- Русская страница по Autotools: <http://squadette.ru/autotools-ru/>
- Книга "GNU Autoconf, Automake, and Libtool":
<http://2read.ru/2006/02/20/gary-v-vaughan-et-al-gnu-autoconf-automake-an...>