shoumikhin 11 января 2011 в 23:45

Пример Makefile

Разработка веб-сайтов

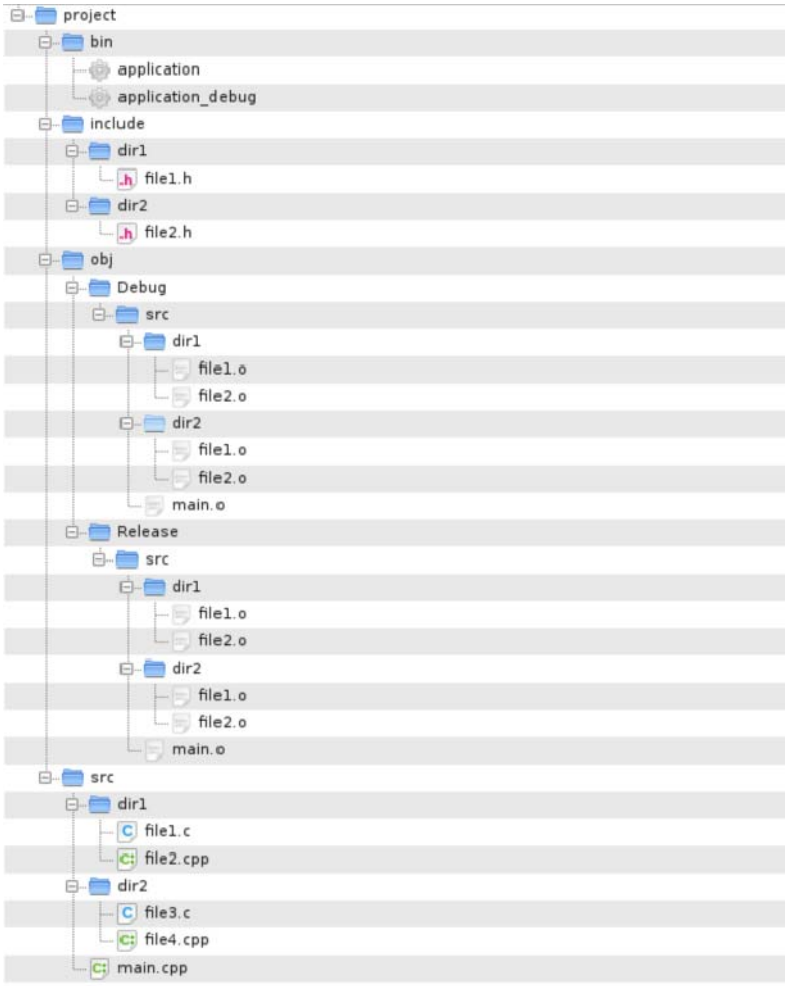
Написание makefile иногда становится головной болью. Однако, если разобраться, все становится на свои места, и написать мощнейший makefile длиной в 40 строк для сколь угодно большого проекта получается быстро и элегантно.

Внимание! Предполагаются базовые знания утилиты GNU make.

Имеем некий типичный абстрактный проект со следующей структурой каталогов:

Пусть для включения заголовочных файлов в исходниках используется что-то типа #include <dir1/file1.h>, то есть каталог project/include делается стандартным при компиляции.

После сборки надо, чтобы получилось так:



То есть, в каталоге bin лежат рабочая (application) и отладочная (application_debug) версии, в подкаталогах Release и Debug каталога project/obj повторяется структура каталога project/src с соответствующими исходниками объектных файлов, из которых и компонуется содержимое каталога bin.

Чтобы достичь данного эффекта, создаем в каталоге project файл Makefile следующего содержания:

```
1. root_include_dir    := include
2. root_source_dir    := src
3. source_subdirs    := . dir1 dir2
4. compile_flags     := -Wall -MD -pipe
5. link_flags          := -s -pipe
6. libraries           := -ldl
7.
8. relative_include_dirs   := $(addprefix ../../, $(root_include_dir))
9. relative_source_dirs   := $(addprefix ../../$(root_source_dir)/, $(source_subdirs))
10. objects_dirs           := $(addprefix $(root_source_dir)/, $(source_subdirs))
11. objects                := $(patsubst ../../%, %, $(wildcard $(addsuffix /*.c*, $(relative_source_dirs))))
```

```
12. objects                 := $(objects:.cpp=.o)
13. objects                 := $(objects:.c=.o)
14.
15. all : $(program_name)
16.
17. $(program_name) : obj_dirs  $(objects)
18.         g++ -o  $@  $(objects) $(link_flags)  $(libraries)
19.
20. obj_dirs  :
21.         mkdir  -p  $(objects_dirs)
22.
23. VPATH := ../../
24.
25. %.o : %.cpp
26.         g++ -o  $@  -c  $< $(compile_flags) $(build_flags) $(addprefix -I, $(relative_include_dirs))
27.
28. %.o : %.c
29.         g++ -o  $@  -c  $< $(compile_flags) $(build_flags) $(addprefix -I, $(relative_include_dirs))
30.
31. .PHONY : clean
32.
33. clean  :
34.         rm  -rf bin  obj
35.
36. include  $(wildcard $(addsuffix /*.d,  $(objects_dirs)))
```

В чистом виде такой makefile полезен только для достижения цели clean, что приведет к удалению каталогов bin и obj. Добавим еще один сценарий с именем Release для сборки рабочей версии:

```
mkdir -p bin
mkdir -p obj
mkdir -p obj/Release
make --directory=./obj/Release --makefile=../../Makefile build_flags="-O2 -fomit-frame-pointer" program_n
ame=../../bin/application
```

И еще один сценарий Debug для сборки отладочной версии:

```
mkdir -p bin
mkdir -p obj
mkdir -p obj/Debug
make --directory=./obj/Debug --makefile=../../Makefile build_flags="-O0 -g3 -D_DEBUG" program_name=../..
/bin/application_debug
```

Именно вызов одного из них соберет наш проект в рабочем, либо отладочном варианте. А теперь, обо всем по-порядку.

Допустим, надо собрать отладочную версию. Переходим в каталог project и вызываем ./Debug. В первых трех строках создаются каталоги. В четвертой строке утилите make сообщается, что текущим каталогом при запуске надо сделать project/obj/Debug, относительно этого далее передается путь к makefile и задаются две константы: build_flags (тут перечисляются важные для отладочной версии флаги компиляции) и program_name (для отладочной версии – это application_debug).

Далее, в игру вступает GNU make. Прокомментируем каждую строку makefile:

1: Объявляется переменная с именем корневого каталога заголовочных файлов.

2: Объявляется переменная с именем корневого каталога исходников.

3: Объявляются переменная с именами подкаталогов корневого каталога исходников.

4: Объявляется переменная с общими флагами компиляции. -MD заставляет компилятор сгенерировать к каждому исходнику одноименный файл зависимостей с расширением .d. Каждый такой файл выглядит как правило, где целью является имя исходника, а зависимостями – все исходники и заголовочные файлы, которые он включает директивой #include. Флаг -pipe заставляет компилятор пользоваться IPC вместо файловой системы, что несколько ускоряет компиляцию.

5: Объявляется переменная с общими флагами компоновки. -s заставляет компоновщик удалить из результирующего ELF файла секции .symtab, .strtab и еще кучу секций с именами типа .debug*, что значительно уменшает его размер. В целях более качественно отладки этот ключ можно убрать.

6: Объявляется переменная с именами используемых библиотек в виде ключей компоновки.

8: Объявляется переменная, содержащая относительные имена каталогов со стандартными заголовочными файлами. Потом такие имена напрямую передаются компилятору, предваряемые ключем -I. Для нашего случая получится ../../include, потому что такой каталог у нас один. Функция addprefix добавляет свой первый аргумент ко всем целям, которые задает второй аргумент.

9: Объявляется переменная, содержащая относительные имена всех подкаталогов корневого каталога исходников. В итоге получим: ../../src/. ../../src/dir1 ../../src/dir1.

10: Объявляется переменная, содержащая имена подкаталогов каталога project/obj/Debug/src относительно текущего project/obj /Debug. То есть, этим мы перечисляем копию структуры каталога project/src. В итоге получим: /src/dir1 src/dir2.
11: Объявляется переменная, содержащая имена исходников, найденных на основе одноименных файлов *.c* (.cpp\.c), безотносительно текущего каталога. Смотрим поэтапно: результатом addsuffix будет ../../src/./*.c* ../../src/dir1/*.c* ../../src/dir2/*.c*. Функция wildcard развернет шаблоны со звездочками до реальных имен файлов: ../../src/./main.cpp ../../src/dir1/file1.c ../../src /dir1/file2.cpp ../../src/dir2/file3.c ../../src/dir2/file4.c. Функция patsubsb уберет префикс ../../ у имен файлов (она заменяет шаблон,

заданный первым аргументом на шаблон во втором аргументе, а % обозначает любое количество символов). В итоге получим: src/./main.cpp src/dir1/file1.c src/dir1/file2.cpp src/dir2/file3.c src/dir2/file4.c.

12: В переменной с именами исходников расширения .cpp заменяется на .o.

13: В переменной с именами исходников расширения .c заменяется на .o.

15: Первое объявленное правило – его цель становится целью всего проекта. Зависимостью является константа, содержащая имя программы (../../bin/application_debug мы ее передали при запуске make из сценария).

17: Описание ключевой цели. Зависимости тоже очевидны: наличие созданных подкаталого в project/obj/Debug, повторяющих структуру каталога project/src и множество объектных файлов в них.

18: Описано действие по компоновке объектных файлов в целевой.

20: Правило, в котором цель – каталог project/obj/Debug/src и его подкаталоги.

21: Действие по достижению цели – создать соответствующие каталоги src/., src/dir1 и src/dir2. Ключ -p утилиты mkdir игнорирует ошибку, если при создании какого-либо каталога, таковой уже существуют.

23: Переменная VPATH принимает значение ../../. Это необходимо для шаблонов нижеследующих правил.

25: Описывается множество правил, для которых целями являются любые цели, соответствующие шаблону %.o (то есть имена которых оканчиваются на .o), а зависимостями для этих целей являются одноименные цели, соответствующие шаблону %.cpp (то есть имена которых оканчиваются на .cpp). При этом под одноименностью понимается не только точное совпадение, но также если имя зависимости предварено содержимым переменной VPATH. Например, имена src/dir1/file2 и ../../src/dir1/file2 совпадут, так как VPATH содержит ../../.

26: Вызов компилятора для превращения исходника на языке C++ в объектный файл.

28: Описывается множество правил, для которых целями являются любые цели, соответствующие шаблону %.o (то есть имена которых оканчиваются на .o), а зависимостями для этих целей являются одноименные цели, соответствующие шаблону %.c (то есть имена которых оканчиваются на .c). Одноименность как в строке 23.

29: Вызов компилятора для превращения исходника на языке C в объектный файл.

31. Некоторая цель clean объявлена абстрактной. Достижение абстрактной цели происходит всегда и не зависит от существования одноименного файла.

32: Объявление абстрактной цели clean.

33: Действие по ее достижению заключается в уничтожении каталогов project/bin и project/obj со всем их содержимым.

36: Включение содержимого всех файлов зависимостей (с расширением .d), находящихся в подкаталогах текущего каталога. Данное действие утилита make делает в начале разбора makefile. Однако, файлы зависимостей создаются только Значит, при первой сборке ни один такой файл включен не будет. Но это не страшно. Цель включения этих файлов – вызвать перекомпиляцию исходников, зависящих от модифицированного заголовочного файла. При второй и последующих сборках утилита make будет включать правила, описанные во всех файлах зависимостей, и, при необходимости,
достигать все цели, зависимые от модифицированного заголовочного файла.

Теги: makefile, make

Хабы: Разработка веб-сайтов

Удачи!

Комментарии 59

 **Alexroid** 11 января 2011 в 23:45

Спасибо за информацию.

 **kibizoidus** 11 января 2011 в 23:51 **+3**

Спасибо. Новичкам будет очень полезно, особенно когда начинаешь задумываться, а как же на самом деле правильно организовать проект.послекомпиляции.

 **soloweb 12 января 2011 в 00:32 0**

Разжевал — спасибо :-)

 12 января 2011 в 00:38

Спасибо, но я за cmake

 **Mad_Fish** 12 января 2011 в 00:53

Пожалуйста, разработчики, НИКОГДА не используйте самописные makefile-ы. Это pain in the ass в поддержке, и для любого, кому понадобится их читать. Используйте CMake, или на худой конец autotools.

Что у autotools, что у CMake стандартизированный набор параметров, которые они принимают при конфигурировании, с которыми достаточно разобраться один раз, и потом каждый проект на них будет казаться вам чем-то знакомым. У самописных мейкфайлов — чёрт знает что, каждый раз разное, и не всегда корректно работающее на различных конфигурациях, с различными компиляторами, с различным расположением внешних зависимостей, и т.д.

Мало того, CMake позволит вам ещё и нормально использовать IDE при работе с кодом (Qt Creator, KDevelop).

 **a_v** 12 января 2011 в 01:00 **+1**

Конечно, сейчас использовать make стоит только для маленьких (из пары-тройки исходных файлов) проектов или для развлечения. Если же пытаться применить make к чему-то большему, то в итоге получится тот же autotools или CMake, только вот с наполовину меньшим функционалом и кучей ошибок.

googol 12 января 2011 в 01:13 **+1**

BTW Android использует голый Make

naryl 12 января 2011 в 01:17 **+2**

make — универсальная утилита. cmake и autotools — специально заточены для C и C++, они не покрывают всех сценариев использоания make.

 **Gorthauer87** 12 января 2011 в 01:23 **+1**

Они расширяемы.

naryl 12 января 2011 в 02:40 **0**

Вы предлагаете мне расширять не связанную с текущим проектом систему вместо того чтобы взять make и делать дело? cmake и autotools не вчера появились. Если бы кто-нибудь положительно оценил их для использования с другими языками, их бы уже расширили.

Вот CL был оценен как язык для веб и на нём сразу появились и веб-сервера и фреймворки. :]

 **Restorer** 12 января 2011 в 02:02 **+5**

поддерживаю. недавно перевёл свой проект (не большой, около 60 c/cpp файлов и около 100 h), первое впечатление — лучше бы оставил на make.

make-файл я написал часа за пол, ещё столько же понадобилось чтоб он нормально заработал на freebsd и pegasos (странная amiga с ppc процессором).
в случае с cmake я убил пол дня только на то, чтобы убедить cmake препроцессить некоторые .c файлы специальнам препроцессором (ну… так надо).

make простой как топор — собственно как и все classic-unix утилиты.
cmake сверх мощный и очень сложный. мне он напоминает продукты Microsoft — стандартные вещи делаются в пол строчки, но если надо что-то нестандартное, готовтесь потратить пол дня (а то и больше).

naryl 12 января 2011 в 11:53 **+1**

Судя по полученным минусам кто-то успешно использует cmake или autotools для всех своих проектов, независимо от языка разработки. Сделайте же бескорыстное доброе дело — поделитесь знанием.

 **Bas1l** 12 января 2011 в 01:23 **0**

Верно. Для полноты картины добавлю, что есть еще QMake и PreMake. Лично мне очень понравился PreMake, потому что используется стандартный скриптовый язык Lua.

googol 12 января 2011 в 02:03 **+1**

PreMake выглядит интересным.

Насчет расширяемости CMake я бы так не спешил говорить. Как раз пытаюсь добавить Vala support в CMake и если честно то я не очень впечатлен внутренностями cmake. BASIC-like синтаксис для макросов с кучей variables бррр Довольно сложно читать этот код.

googol 12 января 2011 в 23:26 **0**

Рыская по просторам интернета нашел еще один интересный build tool

gittup.org/tup/make_vs_tup.html

Отличительная способность — быстрый incremental build. Использует inotify + sqlite где хранит граф зависимостей.

 **Sannis** 14 января 2011 в 22:28 **0**

Если писать Makefile правильно, используя *%.o: %.c*, то всё будет аналогично и при использовании make. И судя по описанию файл tup'а можно по простым правилам преобразовать в Makefile.

googol 14 января 2011 в 22:39 **0**

Фишка Tup не в синтаксисе (он очень близок к Makefile) а в том как эффективно он собирает incremental builds. Почитайте по ссылке что я привел — не пожалеете.

> И судя по описанию файл tup'а можно по простым правилам преобразовать в Makefile.
Вы хотели сказать *ИЗ* Makefile?

Хотя я нашел Tup только лишь пару дней назад — мне он очень нравится. Уже собрал LLVM/Clang/Bison этой тулзой. Работает великолепно.

pomozok.wordpress.com/2011/01/13/66/

 **Sannis** 14 января 2011 в 23:04 **0**

Я прочитал, прежде чем говорить :) На первый взгляд магии не видно и автор сам говорит о том, что вариант с Makefile был некорректным. Про корректность варианта с Tup ни слова. Да и в целом тест выглядит довольно синтетически, никто не будет использовать в проекте 10000 файлов разложенных по сбалансированному дереву.

googol 16 января 2011 в 20:58 **0**

Не совсем понял в чем синтетичность теста. То есть если разбросать файлы случайным образом (разное количество в разных папках) то Make сразу же быстро заработает? Сильно в этом сомневаюсь.

Пример работы Tup — инкрементная сборка Linux distro (похожая на Gentoo) gittup.org/gittup/

```
[marf@captainfalcon gittup]$ vi nethack/src/spell.c
[marf@captainfalcon gittup]$ vi busybox/coreutils/ls.c
[marf@captainfalcon gittup]$ time tup upd -j2
Executing Commands
```

```
[ 0/9  ] busybox/coreutils/CC  ls.c
[ 1/9  ] nethack/CC src/spell.c
[ 2/9  ] busybox/coreutils/LD built-in.o
[ 3/9  ] busybox/LD busybox
[ 4/9  ] nethack/LD nethack
[ 5/9  ] initrd/bin/CP busybox
[ 6/9  ] initrd/bin/CP nethack
[  7/9   ] initrd/MKINITRD
[ 8/9  ] initrd/GZIP initrd
[ 9/9  ]

real    0m1.571s
user    0m1.888s
sys 0m0.269s
```

Полторы секунды. Вот это я и подразумеваю быстрая инкрементрая сборка. gmake только stat() будет делать секунд 10.

 **Sannis** 17 января 2011 в 05:00 **0**

> Пример работы Tup — инкрементная сборка Linux distro (похожая на Gentoo) gittup.org/gittup/
Всё-таки это довольно специфичная задача. И в любом случае это не 6 вложенных папок по 10 файлов, как в случае тестирование tup на 1000000 файлах.

> Полторы секунды. Вот это я и подразумеваю быстрая инкрементрая сборка. gmake только stat() будет делать секунд 10.
А tup не делает его как будто, и получает информацию об изменённых файлах астральным путём? ;) Что-то я сомневаюсь.

Попробую на досуге подкрепить свои слова аргументами.

googol 17 января 2011 в 05:44 **0**

> А tup не делает его как будто, и получает информацию об изменённых файлах астральным путём? ;) Что-то я сомневаюсь.

Tup делает но только один лишь раз. Далее использует inotify. По ссылке все описано как это работает.

 **Sannis** 17 января 2011 в 17:44 **0**

Только если запустить демон `tup monitor`. При этом в описании теста gittup.org/tup/make_vs_tup.html не сказано, что перед вторым запуском запускался монитор. Это и вызывает сомнения, ибо в таком случае tup upd будет сканировать всё и вызывать stat-ы

В остальном же получается, что кроме удобного синтаксиса мы выигрываем только за счёт отсутствия stat-а и только при запущенном мониторе.

zone19 12 января 2011 в 01:26 **0**

Для полноты картины добавлю также Scons и Waf. Первый более развитый, второй более быстрый и архитектурно более правильный. Для написания кода в обоих используется Python, что не может не радовать.

naryl 12 января 2011 в 02:45 **+1**

bash стандартнее чем Python и Lua (Premake из соседнего коммента) когда-либо будут.

НЛО прилетело и опубликовало эту надпись здесь

naryl 12 января 2011 в 04:36 **0**

Скажите это сюда: habrahabr.ru/blogs/development/111691/#comment_3565108

И сюда: habrahabr.ru/blogs/development/111691/#comment_3565190

 **GreyCat** 12 января 2011 в 07:22 **0**

Вы это _серьёзно_? Подскажете, где посмотреть _стандарт_ на _bash_?

naryl 12 января 2011 в 12:02 **0**

ru.wikipedia.org/wiki/Де-факто

-_- Вы же не глупый человек, понимаете что _скрипты пишут на bash_, как это ни прискорбно. Вы ведь не будете предлагать использовать Lua или Python только для билд-скриптов. Tcl — хороший кандидат. И формальнй стандарт присутствует, и система компактная, и синтаксис умещается на страничку. Только вот нет на нём билд-систем.

 **GreyCat** 12 января 2011 в 14:01 **0**

Вот даже в контексте разговора про системы сборки, я ни разу не вижу какого-либо доминирования bash (или даже sh, если вы имели в виду на самом деле его). Например, для тех, кто планирует быть портабельными под Windows, вероятность запуститься на Windows с cmake-системой или даже с какими-нибудь скриптами на python — радикально выше, чем если там что-то, написанное на bash, даже еще наверняка и с жестокой привязкой к какой-то определенной файловой иерархии, механизмам ldconfig и т.п.

naryl 12 января 2011 в 20:02 **0**

Да, конечно sh. Все до единого известные мне портабельные проекты всё-равно собираются под mingw или MSYS. Ну и в конце концов ответьте:

> вероятность запуститься на Windows с cmake-системой или даже с какими-нибудь скриптами на python — радикально выше, чем если там что-то, написанное на bash

КАК ИСПОЛЬЗОВАТЬ cmake ДЛЯ ПРОЕКТОВ БЕЗ СИ И ПЛЮСОВ?!? Уже третий раз вижу предложение использовать cmake, а как собрать им, например код на D, так никто и не рассказал. Python неприемлем по причинам писанным выше.

 **GreyCat** 12 января 2011 в 20:48 **0**

> Все до единого известные мне портабельные проекты всё-равно собираются под mingw или MSYS

Вы либо обладаете достаточно узким кругозором, либо, что скорее — хотели сказать что-то не то, что сказали. Например, масса проектов на Qt (хотя бы на том же qt-apps.org посмотрите — их не так мало) собираются с помощью qmake, которая не требует использовать никаких sh-скриптов, не требует никакого UNIX toolkit'a на собирающей машине и имеют возможность прекрасно собраться на Windows-машине с только Microsoft Visual Studio и их чудным C-компилятором.

> КАК ИСПОЛЬЗОВАТЬ cmake ДЛЯ ПРОЕКТОВ БЕЗ СИ И ПЛЮСОВ?!? Уже третий раз вижу предложение использовать cmake, а как собрать им, например код на D, так никто и не рассказал.

Во-первых, пожалуйста, не кричите, во-вторых, сэкономьте мне немного времени и расскажите тогда уж сразу, чем именно вам не нравится CMakeD?

naryl 13 января 2011 в 02:06 **0**

Нет, я обладаю узким кругозором. По счастливой случайности никогда не пробовал под виндовс собрать что-то, требующее большего чем mingw.

> чем именно вам не нравится CMakeD?

Спасибо, посмотрю, но D — тоже не единственный в мире язык. Вы продолжаете игнорировать мою главную мысль, высказанную в самом начале ветки. Все любимые вами утилиты требуют допиливания под *каждый* сценарий использования. make — нет.

Mezomish 12 января 2011 в 23:03 **0**

>Да, конечно sh. Все до единого известные мне портабельные проекты всё-равно собираются под mingw или MSYS

Притащить на Винду MinGW это совсем не то же самое, что притащить туда целый Cygwin.

kekoz 13 января 2011 в 01:21 **+1**

Про “скрипты пишут на bash” я бы поспорил. Во-первых, их много на чём пишут :) но это я не о том. Во-вторых, если уж говорить о стандартном unix-shell, то таковым является не bourne-again shell, а тот, который описан в IEEE Std 1003.2 (с ногами, растущими из korn shell). bash стал де-фактом по понятной причине — linux. Фактически навязан. Оно бы и не беда, если бы не одно “но” — количество людей, уверенных, что sh == bash огромно (благодаря тому, что в linux'ах он таки называется /bin/sh). Когда бы все они были просто юзерами, то не было бы проблемы — пусть каждый пользует то, что ему нравится. Но когда на нём начинают писать портабельные (якобы) скрипты, вот здесь веселье и начинается. Я уже не возьмусь сосчитать, сколько раз приходилось объяснять людям, почему у них не работают массивы :)

Люто ненавижу Makefile с bash'евскими конструкциями.

GreyCat 13 января 2011 в 05:43 **+1**

В Linux'ах всё на самом деле по-разному. В том же Debian/Ubuntu, например, /bin/sh — это dash.

Проблема даже не в башизмах так таковых, а более общая — в том, что сам стандарт и на POSIX shell, и на *utils — объективно плохой или, как минимум, устаревший. Как следствие — мы имеем несколько тысяч реализаций, в которых «shell is 98% POSIX compatible», а в utils нескольких утилиток нет, а еще несколько — работают с определенными отклонениями (в лучшую сторону, конечно же, по мнению автора) от стандарта.

Субъективно — на сейчас — UNIX shell — один из самых непортабельных языков вообще. Ситуация с ним напоминает ситуацию с языком C до середины 90-х годов, но с C ее хоть как-то порешали (хотя и тоже плохо и сумбурно), а за UNIX shell, видимо, не стоят такие гиганты индустрии…

googol 13 января 2011 в 01:26 **0**

Вспомнилась фраза «It's easier to port a shell than a shell script. — Larry Wall»

mikhailian 12 января 2011 в 12:02 **+3**

Вы видно с autotools мало работали…

damnerd 13 января 2011 в 00:41 **0**

Работал мало и больше не хочу. GNU Autohell, GNU Autocrud, GNU Autocrap…

Overdese 12 января 2011 в 00:55 **0**

Спасибо! Побольше бы подобных статей!

 **a_v** 12 января 2011 в 00:56 **+3**

www.gnu.org/software/make/manual/html_node/Implicit-Variables.html — иначе получается, что компилировать C++ мы можем только командой g++, C — почему-то тоже этой же командой, а файлы удалять — только rm.

zone19 12 января 2011 в 01:22 **+1**

Я для подсветки использовал этот сайт. Надо только не забыть поставить галочку «Use tag (for Habrahabr)» и «Line number», а в результирующем коде внутрь тега blockquote поставить тег code.

Единственный минус — там нет подсветки для Make, но можно использовать либо «Bash», либо «Plain text».

zone19 12 января 2011 в 01:30 **0**

Забыл самое главное — полный список доступных «оформителей» доступен в этом посте

 **TiGR** 12 января 2011 в 10:11 **+3**

Зачем, если на хабре есть <source />?

 **icc** 12 января 2011 в 04:39 **0**

я бы еще добавил в этот makefile код, который бы учитывал библиотеки как с ключом *L*, так и с ключом *l*.

 **icc** 12 января 2011 в 04:41 **0**

И все таки в один clean объединять все rm'ы, ИМХО это как-то неправильно.

tkirill128 12 января 2011 в 08:26 **+2**

У вас нумерация списка в статье какая-то непоследовательная :)

 **Mezomish** 12 января 2011 в 22:12 **+1**

Автор не только руками пишет мейкфайлы, но и руками нумерует списки :)

 **proxor** 12 января 2011 в 09:34 **0**

Спасибо, я лучше SCons возьму.

 **andrewsh** 12 января 2011 в 09:48 **0**

Извините, но это плохой, негодный мейкфайл.

 **Captcha** 12 января 2011 в 10:49 **0**

Отвергая — предлагай.

andrewsh 12 января 2011 в 10:51 **0**

Запустите make -p и редуцируйте мейкфайл, используя уже определённые правила и переменные.

НЛО прилетело и опубликовало эту надпись здесь

andrewsh 12 января 2011 в 11:01 **+1**

В нём слишком много велосипедизма, если можно так выразиться. А скрипты Debug/Release — зачем они сделаны отдельно от мейкфайла?

НЛО прилетело и опубликовало эту надпись здесь

Sannis 12 января 2011 в 13:07 **+1**

```
make --directory=./obj/Release --makefile=../../Makefile build_flags="-O2 -fomit-frame-pointer" program_name
=../../bin/application
```

Таки вам не кажется, что это очень грязный хак?

erley 12 января 2011 в 13:32 **0**

У вас всё сломается если вдруг в одной директории обнаружатся два файла dummy.c и dummy.cpp
А ещё встречаются расширения .cxx .cc .hxx .hh .hpp
Кстати, иногда приходится собирать за раз несколько бинарников с разными клюджами -DWITH_abc
И чтоб один скрипт сборки понимал что под ним линукс или винда или вообще эмулятор встраиваемой платформы.
Но это всё поправимо при желании, просто каждый затачивает систему сборки под свой проект сам. И это, не люблю почему-то VPATH, непомню конкретно, но что-то там криво было у make с ним, может сейчас прошло уже...

А вот autotools и прочие свистелки годятся только для проектов на sf.org, т.е. там, где люди сами толком не знают какие библиотеки у них в системе стоят и вообще, как у них там всё работает.
Когда грамотный человек что-то разрабатывает, то он будет сам контролировать какая библиотека у него подцепилась и какой инклюд откуда сработал. Я не буду разбираться кто виноват, я или этот autotools — просто исключу его из цепочки и спокойно буду работать. autotools можно будет добавить позже, но в работе он только мешает

icc 12 января 2011 в 14:39 **+1**

вот поэтому лучше не лениться и забить все имена файлов в makefile

НЛО прилетело и опубликовало эту надпись здесь

shoumikhin 13 января 2011 в 14:51 **0**

Что-то тут нечисто. Попробуйте мой.