How to Make?

A hands-on walkthrough of the Make Build System

Meet and greet

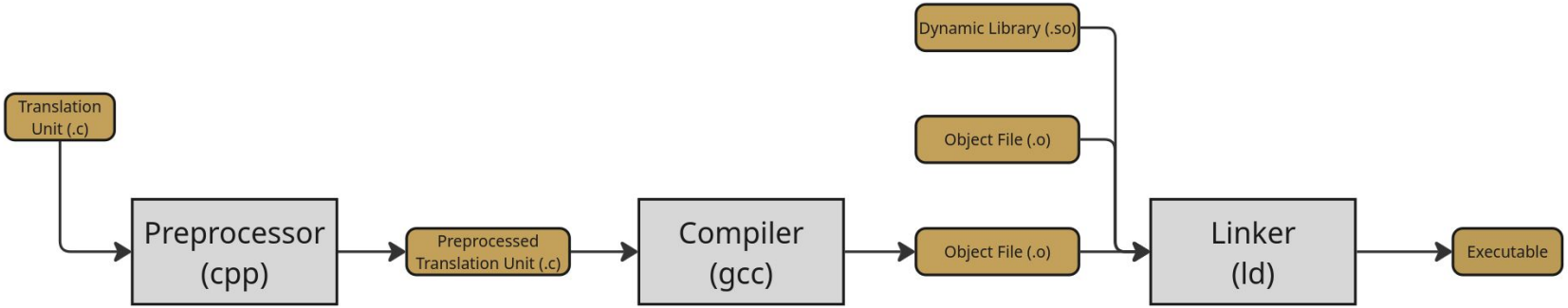
Λίγα λόγια για μένα

Γεια σας! Λέγομαι Πέτρος. Είμαι:

- Δευτεροετής φοιτητής του τμήματος
- Βοηθός του μαθήματος (κάποιοι λογικά με ξέρετε)
- Τεράστιος fan των Linux και του open-source λογισμικού
- Μου αρέσουν οι γάτες
- Και…i use vim btw (well neovim but same spiel)

C Build Process

Με τι έχουμε να κάνουμε;



Say ‘Aye’ if you know what this is!

Παράδειγμα: math.h και libm.so

```c
#include <stdio.h>

int main(void) {
      printf(“%.2f\n”, sqrt(3));
      return 0;
}
```

Τι έγινε εδώ; Τι είδους error είναι αυτό;

Είναι compiler error, καθώς ο compiler δεν μπορεί να βρει την δήλωση της sqrt στο αρχείο μας.
Παρατηρήστε ότι ο compiler μας λέει ακριβώς πως να το διορθώσουμε.

```
$ gcc -o main main.c
main.c: In function ‘main’:
main.c:4:20: error: implicit declaration of function ‘sqrt’
[-Wimplicit-function-declaration]
    4 |   printf("%.2f\n", sqrt(3));
      |                    ^~~~
main.c:2:1: note: include ‘<math.h>’ or provide a declaration of ‘sqrt’
    1 | #include <stdio.h>
  +++ |+#include <math.h>
    2 |
```

Παράδειγμα: math.h και libm.so

```c
#include <stdio.h>
#include <math.h>

int main(void) {
      printf("%.2f\n", sqrt(3));
      return 0;
}
```

```
$ gcc -o main main.c
/usr/bin/ld: /tmp/ccaw9D8e.o: in function `main':
main.c:(.text+0x1f): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
```

Τι έγινε εδώ; Τι είδους error είναι αυτό τώρα;

Είναι linking error. Ο linker δεν μπορεί να εντοπίσει μέσα στο linking scope συνάρτηση με όνομα sqrt. Παρατηρήστε ότι σε αντίθεση με τον compiler, δεν είναι και πολύ βοηθητικό αυτό το μήνυμα. Έχει κανείς ιδέα γιατί;

Παράδειγμα: math.h και libm.so

Η συμπερίληψη του `math.h`, απλά μας δίνει τις δηλώσεις που χρειαζόμαστε. Θα πρέπει κάπως να πούμε στον linker που να βρει την υλοποίηση της sqrt. Χρειαζόμαστε ένα object file.

Συγκεκριμένα, το `math.h` περιλαμβάνει τις δηλώσεις για το math library του συστήματος, που έχει όνομα libm.so. Για να πούμε στον linker να χρησιμοποιήσει αυτή την βιβλιοθήκη κατά το linking αρκεί απλά να προσθέσουμε στην εντολή linking ένα flag ως εξής:

```c
#include <stdio.h>
#include <math.h>

int main(void) {
      printf("%.2f\n", sqrt(3));
      return 0;
}
```

```
$ gcc -o main main.c -l:libm.so # or -lm for short
$ ./main
1.73
```

A simple project

Ένα δικό μας Project

Μέχρι τώρα έχουμε ασχοληθεί μόνο με projects που αποτελούνται από 1 αρχείο. Στον πραγματικό κόσμο όμως, τα projects αποτελούνται από πολλά αρχεία, που πρέπει να κάνουν compile με όμοιο τρόπο και link.

Έστω λοιπόν το παρακάτω project:

```c
/* file: main.c */
typedef unsigned long long ull;
ull *n_primes(ull n);

int main(int argc, const char **argv) {
      // …
      ull *primes = n_primes(n);
      // …
}
```

```c
/* file: primes.c */
typedef unsigned long long ull;
ull *n_primes(ull n) {
      // …
}
```

Πώς το κάνουμε compile?


```
$ gcc main.c primes.c -o main
$ ./main 21
Found primes: 2 3 5 7 11 13 17 19
```

```
$ cpp main.c -o main.C
$ cpp primes.c -o primes.C
$ gcc main.C -c
$ gcc primes.C -c
$ ld -o main --dynamic-linker /lib64/ld-linux-x86-64.so.2
/lib/crt1.o /lib/crti.o main.o primes.o -lc /lib/crtn.o
$ ./main 21
Found primes: 2 3 5 7 11 13 17 19
```

Πώς το κάνουμε compile?

Η στρατηγική μας ως τώρα, έχοντας εισάγει πλέον κι άλλα αρχεία πέραν του ενός έχει 3 προβλήματα.

1. long long compilation commands

2. Many compilation commands

Θέλω το primes.c translation unit να μην έχει το standard enforcement επειδή θέλω να χρησιμοποιήσω την reallocarray που είναι non-standard.

Πώς θα το κάνω αυτό;

Το compilation πρέπει να σπάσει σε βήματα.

```
$ gcc -Wall -Wextra -Werror -std=c99 -pedantic -c main.c
$ gcc -Wall -Wextra -Werror -pedantic -c primes.c
$ gcc -o main primes.o main.o
```

You probably are interested...

```
$ gcc -Os -c -Wall -Wextra -Werror -pedantic recurse.c
$ gcc -Os -c -Wall -Wextra -Werror -pedantic brute.c
$ gcc -Os -c -Wall -Wextra -Werror -pedantic memoize.c
$ gcc -Os -c -Wall -Wextra -Werror -pedantic dp.c
$ gcc -Os -c -Wall -Wextra -Werror -pedantic elevate.c
$ gcc -Os -o elevate recurse.o brute.o memoize.o dp.o elevate.o
```

Βλέπετε ότι κάτι δεν πάει καλά

Έχω ήδη ένα πολύ απλό project το οποίο χρειάζεται να γράψω υπερβολικά πολλά για να το κάνω compile.

Δεν ξέρω για εσάς αλλά εγώ δεν μπορώ να θυμάμαι απ' έξω όλα τα commands που χρειάζομαι για να κάνω compile ένα project.

Χειρότερο: ακόμη κι αν τα ήξερα ο χρόνος που θα έχανα για να τα τρέχω 1-1 σε ένα μεγάλο project(θα δούμε σύντομα) θα ήταν περισσότερος από το χρόνο που θα μας έπαιρνε να κάνουμε μια μικρή αλλαγή στον κώδικα.

Μπορώ με κάποιον τρόπο να το αποφύγω όλο αυτό;

Introducing the revolutionary technology of: bash

Η λύση σε αυτή την φάση είναι ιδιαίτερα απλή και έχει βρεθεί από τα ‘70s . Αρκεί απλά να πάρω όλο compilation process και να το βάλω μέσα σε ένα bash script.

```
#!/usr/bin/env bash
# file: build.sh
set -xe
gcc -Wall -Werror -Wextra -pedantic -std=c99 -c main.c
gcc -Wall -Werror -Wextra -pedantic -c primes.c
gcc -o main main.o primes.o
```

```
$ chmod +x ./build.sh
$ ./build.sh 2> /dev/null
$ ./main 42
Found primes: 2 3 5 7 11 13 17 19
23 29 31 37 41
```

Αφού γίνει αυτό χρειάζεται απλά να τρέξουμε το script και ξαφνικά έφυγε πολλή πίεση από πάνω μας.

3. Recompilation ΟΛΟΚΛΗΡΟΥ του Project

Το bash script, όσο κι αν μας λύνει τα χέρια, δεν είναι **φτιαγμένο** για να κάνει compile projects. Είναι μια απλή scripting γλώσσα, που τυχαίνει και εμείς την χρησιμοποιήσαμε για να κάνουμε compile.

Ως αποτέλεσμα, κάθε φορά που τρέχουμε το script, ανεξάρτητα από το αν έχει αλλάξει κάποιο αρχείο, αυτό θα κάνει recompile όλο το πρόγραμμα από την αρχή.

Και;

Glibc

Ας πούμε ότι θέλουμε λοιπόν με bash script να κάνουμε compile την GNU Lib C, την πιο διαδεδομένη υλοποίηση του C standard library + διαφόρων άλλων προτύπων (POSIX, GNU C, System V etc.).

Πρόκειται για ένα project με 1.524.568 γραμμές κώδικα μοιρασμένες σε πάνω από 14.000 αρχεία.

Αν θέλαμε απλά να βάλουμε comments σε ένα αρχείο C, θα έπρεπε να κάνουμε recompile 14.000 αρχεία από την αρχή. Ακούγεται για καλή χρήση του χρόνου μας;

Το σίγουρο είναι ότι θα προλαβαίνουμε να κάνουμε πολλά πράγματα ανάμεσα σε compilations…(εγώ προσωπικά ανοίγω Frieren episodes)

Πώς μπορούμε να το αποφύγουμε αυτό;

Θα κάνω recompile με το χέρι τα αρχεία που αλλάζω.

Θα κάνω link με ένα bash script.

Κι αν θέλω να κάνω compile όλο το project, θα έχω ένα άλλο script για αυτό.

Κι αν ξεχάσεις να κάνεις κάτι recompile;

Enter Make and Makefiles

Make originated with a visit from Steve Johnson (author of yacc, etc.), storming into my office, cursing the Fates that had caused him to waste a morning debugging a correct program (bug had been fixed, file hadn't been compiled, `cc *.o` was therefore unaffected). As I had spent a part of the previous evening coping with the same disaster on a project I was working on, the idea of a tool to solve it came up. It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make that weekend. Use of tools that were still wet was part of the culture. Makefiles were text files, not magically encoded binaries, because that was the Unix ethos: printable, debuggable, understandable stuff.

—Stuart Feldman, *The Art of Unix Programming*, Eric S. Raymond 2003

Make Basics

Make

Πρόκειται για πρόγραμμα που μπορεί να εκτελέσει εντολές με βάση κάποιες προκαθορισμένες σχέσεις εξάρτησης.

It comes in many flavors.

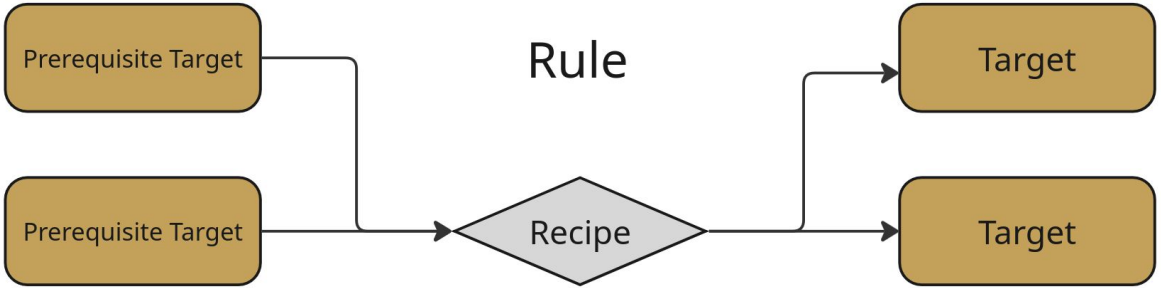
Εμείς θα μιλήσουμε για το GNU Make, αν και σημαντικό μέρος των πραγμάτων που θα πούμε ισχύει και για τα υπόλοιπα Make variants.

GNU Make

Make: Βασικές Έννοιες

Η χρήση του Make απαιτεί την εισαγωγή κάποιων καινούργιων εννοιών, θεμελιωδών για αυτό:

- Target: κάποιο αποτέλεσμα (αρχείο ή/και κάτι άλλο) που μπορεί να παραχθεί
- Recipe: η διαδικασία με την οποία μπορεί να κατασκευαστεί ένα target
- Rule: πρόκειται για τρόπο ορισμού σχέσεων εξάρτησης ανάμεσα σε targets

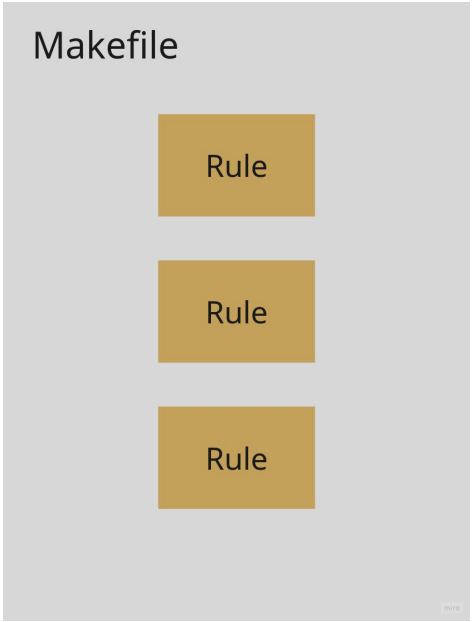


Make: Makefile

Ένα Makefile δεν είναι παρά ένα σύνολο από rules το οποίο αποσκοπεί στην παραγωγή διαφόρων target.

Για να οριστεί ένα rule μέσα σε ένα Makefile, χρησιμοποιούμε την ακόλουθη σύνταξη:

```
# rule
target_name1 target_name2: prerequisite1 prerequisite2 # ...
      # recipe
      command1
      command2
```



Make: Makefile

```
# rule
target_name1 target_name2: prerequisite1 prerequisite2 # … more targets
	# recipe
	command1
	command2
```

Το παραπάνω ορίζει ένα rule που φτιάχνει τα targets με ονόματα `target_name1 target_name2` τρέχοντας τα commands `command1  command2` σε shell και δηλώνει ότι για να μπορέσει να φτιάξει το `target_name` χρειάζεται να έχει φτιάξει τα targets `prerequisite1` και `prerequisite2`.

Μεγάλη προσοχή: το indentation είναι σημαντικό(αν κάτι είναι indented, το Make θεωρεί ότι είναι μέρος recipe) και πρέπει να είναι consistent σε όλο το αρχείο.

Make: make

Εφόσον σε ένα directory υπάρχει ένα αρχείο με το όνομα Makefile(ή καμιά 5 άλλα πιθανά ονόματα), προκειμένου να χρησιμοποιήσουμε το Make, αρκεί απλά να τρέξουμε το παρακάτω από ένα terminal:

```
$ make target_name
```

Αυτό θα οδηγήσει στην εκτέλεση του προγράμματος `make`, το οποίο θα προσπαθήσει να κατασκευάσει το target `target_name` ακολουθώντας το τελευταίο rule που ορίζει recipe με αυτό το όνομα μέσα στο Makefile.

Αν δεν περαστεί όνομα από target, το make θα θεωρήσει ως build target το target που ορίζεται από το πρώτο rule του αρχείου.

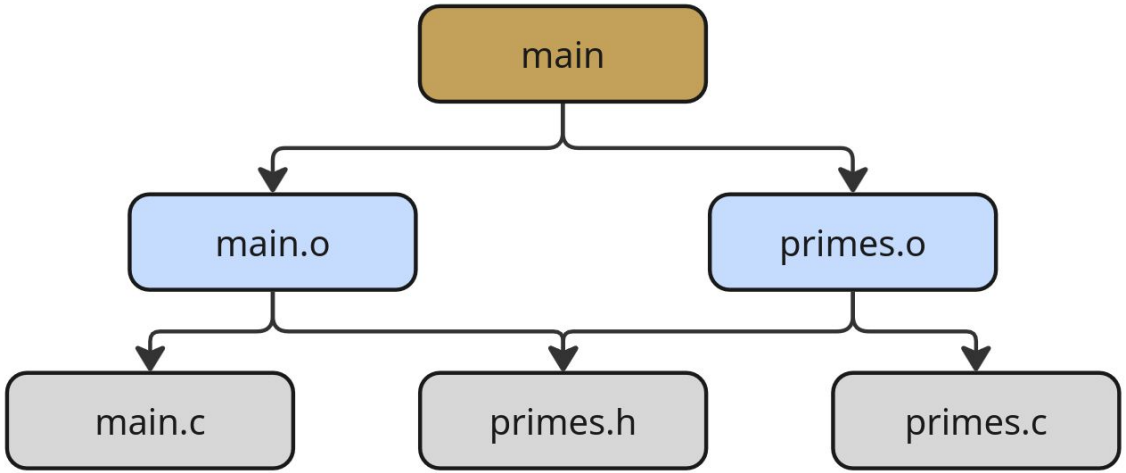
```
$ make
```

Make: build procedure

Αφού τρέξουμε το Make και επιλεχθεί το build target, ξεκινάει η διαδικασία παρασκευής του. Ακολουθείται περίπου ο εξής απλός αλγόριθμος, ξεκινώντας από το build target:

```
proc Make-Target(target):
     foreach prerequisite of target:
          if prerequisite is out of date:
               mark target out of date
               Make-Target(prerequisite)

     if target is out of date:
          run recipe of target
endproc
```



Οι εξαρτήσεις του project που θέλουμε να κάνουμε build.

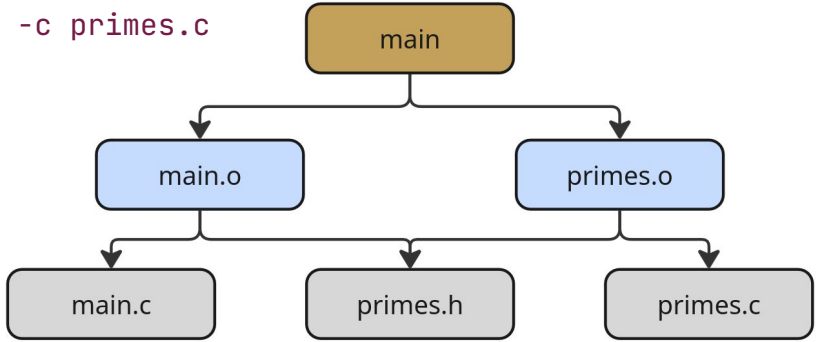
Improved Makefiles

Που βρισκόμαστε;

```
main: main.o primes.o
    gcc -o main main.o primes.o

main.o: main.c primes.h
    gcc -Wall -Wextra -Werror -pedantic -std=c99 -c main.c

primes.o: primes.c primes.h
    gcc -Wall -Wextra -Werror -pedantic -c primes.c
```



Λειτουργεί; Ναι. Είναι καλό; Όχι

Ο φίλος μου (ο ίδιος με πριν), θέλει να κάνει compile στο machine του αλλά δεν χρησιμοποιεί gcc αλλά clang, έναν διαφορετικό C compiler (Mac users, you know who you are…).

Πρέπει τώρα να αλλάζουμε 1-1 τις εμφανίσεις του gcc μέσα στο Makefile, το οποίο αν το project μας είναι μεγάλο, θα οδηγήσει σε υπερβολικά πολλές αλλαγές.

Όταν είχαμε στην C το αντίστοιχο πρόβλημα, η λύση ήταν να χρησιμοποιήσουμε μεταβλητές. Ας κάνουμε το ίδιο και εδώ.

Make: Macros

Στο Make αντί για variables έχουμε macros, τα οποία στην πραγματικότητα έχουν συμπεριφορά παρόμοια με variables. Τα macros περιέχουν strings. Τιμές αποδίδουμε με αναθέσεις, η σύνταξη για τις οποίες ακολουθεί:

```
MACRO_NAME <assignment-operator> string value
```

Συνήθως τα ονόματα από macros είναι με κεφαλαία. Οι τιμή που ανατίθεται σε αυτά είναι string χωρίς αυτάκια (“ , ‘).
Η τελική τιμή που θα έχει ένα macro, όταν χρησιμοποιηθεί, είναι η τελευταία που αποδίδεται σε αυτό μέσα σε ένα Makefile(ή μέσα σε ένα rule, όπως θα δούμε σύντομα).

Make: Macro Assignments

Υπάρχουν 2 είδη από αναθέσεις που μπορούν να γίνουν σε macros:

- Lazily evaluated ή recursive macros: αποτιμούνται μόνο κατά την χρήση τους
- Eagerly evaluated ή simple macros: αποτιμούνται κατά την ανάθεση σε αυτά

Υπάρχουν διάφοροι τελεστές ανάθεσης. Οι πιο σημαντικοί είναι οι παρακάτω:

1. = : Κάνει lazy assignment
2. := : Κάνει eager assignment
3. ?= : Κάνει lazy assignment μόνο αν δεν έχει ήδη οριστεί τιμή για την αριστερή του μεριά
4. += : Προσθέτει την δεξιά μεριά στο τέλος της αριστερής κι αποθηκεύει στην αριστερή το αποτέλεσμα lazily.

Make: Using Macros

Για να χρησιμοποιήσουμε ένα macro σε οποιοδήποτε σημείο του Makefile μας, αρκεί απλά να γράψουμε το αποτιμήσουμε εκείνη την χρονική στιγμή. Αυτό το κάνουμε με τον εξής τρόπο

```
$(MACRO_NAME)
```

Μεγάλη προσοχή: η αποτίμηση των lazily assigned macros, θεωρούμε ότι γίνεται μετά την ανάγνωση όλου του Makefile, ενώ των eagerly κατά την διάρκεια της ανάγνωσης.

Make: Automatic Variables

Σε ένα recipe, μπορούν να χρησιμοποιηθούν πέραν από macros και μεταβλητές των οποίων τα περιεχόμενα αλλάζουν αυτόματα από rule σε rule, που ονομάζονται automatic variables. Οι πιο σημαντικές είναι:

- $@: το όνομα του target που προσπαθεί να παραχθεί αυτή την στιγμή
- $^: λίστα από τα prerequisites
- $<: το πρώτο prerequisite της λίστας από prerequisites

Make it beautiful & simple

Παρατηρήστε ότι πλέον έχουμε δύο σχεδόν πανομοιότυπα rules που κατασκευάζουν τα αρχεία που θέλουμε. Όπως και στον προγραμματισμό, όταν βλέπουμε ότι κάτι επαναλαμβάνεται, ψάχνουμε κάποιον τρόπο να το αφαιρέσουμε(to abstract it). Με την χρήση των automatic variables, των macros έχουμε φτάσει πολύ κοντά.

Αυτή την στιγμή φαίνεται ότι όλα τα object files, απλά θα επαναλαμβάνουν το ίδιο recipe.

Θα ήταν καλό αν μπορούσαμε να ορίσουμε ένα γενικό rule και recipe για object files…

Make: Wildcards

Μπορούμε να ορίσουμε wildcard rules, τα οποία δημιουργούν οικογένειες από rules που επιδρούν πάνω σε όμοιες οντότητες. Συγκεκριμένα εμάς μας ενδιαφέρει να ορίσουμε ότι όλα τα object files, έχουν prerequisite το αντίστοιχο translation unit και παράγονται με το ίδιο recipe.

Το Make μας δίνει την δυνατότητα να το κάνουμε αυτό με wildcard rules. Χρησιμοποιώντας τον ειδικό χαρακτήρα ‘%’, μπορούμε να ορίσουμε μια οικογένεια από rules ως εξής:

```
%.o: %.c
     recipe
```

Το παραπάνω snippet, ορίζει ακριβώς αυτό που περιγράψαμε.

Make: Built-in Rules

Ας θυμηθούμε ότι το Make φτιάχτηκε για να κάνει compile C projects. Αυτό σημαίνει ότι για καλή μας τύχη, πολλά από τα πράγματα που θέλουμε να κάνουμε, ήθελαν και οι δημιουργοί του να τα κάνουν. Συγκεκριμένα:

- Το rule που έχουμε για να κάνουμε compile object files από translation units.
- Το rule που έχουμε για να κάνουμε link object files, σε executables.
- και πολλά άλλα rules που μπορείτε να βρείτε εδώ.

χρησιμοποιούνται τόσο συχνά που το ίδιο το Make τα έχει ορισμένα by default.

Τι σημαίνει αυτό για εμάς;

Μπορούμε να κάνουμε το Makefile μας, 2-3 γραμμές.

Και πού είστε ακόμη;

Λόγω της εξαιρετικής απλότητάς του αλλά και της δύναμης που παρέχει, το Make μπορεί να κάνει πολλά παραπάνω:

1. Το Linux Kernel (~35M γραμμές κώδικα) χρησιμοποιεί make για την ιδιαίτερα σύνθετη διαδικασία μετάφρασης του.
2. Τα περισσότερα πιο advanced εργαλεία(cmake, premake, automake, autoconf) στην ουσία είναι χτισμένα πάνω στο Make και το χρησιμοποιούν. Έτσι στην πραγματικότητα, όλα τα μεγάλα project χρησιμοποιούν Make, είτε άμεσα είτε έμμεσα.
3. Μπορεί να δουλέψει και για άλλες γλώσσες πέραν της C.

42 Slides...

Σας ευχαριστώ. Keep Coding :)