Building Software Systems

- Even small software systems need to to use tools to control builds.
- Many, many tools available
- Tools popular with developers often changing, and specific to platform/language.
- We'll look at a classic tool **make** which is still widely used e.g. Linux kernel
- If you want current alternative: cmake + ninja
- But you should know **make**

make

`make` allows youto

- document intra-module dependencies
- automatically track of changes

`make` works from a file called `Makefile` (or `makefile`)
A `Makefile` contains a sequence of rules like:

```
    target : source1 source2 ...
            commands to create target from sources
```

Beware: *each command is preceded by a single* tab *char.*
Take care using cut-and-paste with`Makefiles`

Dependencies

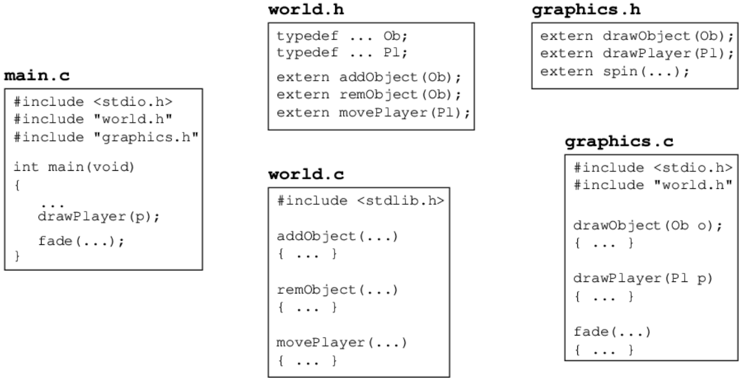
The `make` command is based on the notion of *dependencies*.
Each rule in a `Makefile` describes:

- dependencies between each target and its sources
- commands to build the target from its sources

`Make` decides that a target needs to be rebuilt if

- it is older than any of its sources (based on file modification times)

Example Multi-module C Program



main.c

```
#include <stdio.h>
#include "world.h"
#include "graphics.h"

int main(void)
{
   ...
   drawPlayer(p);

   fade(...);
}
```

world.h

```
typedef ... Ob;
typedef ... Pl;

extern addObject(Ob);
extern remObject(Ob);
extern movePlayer(Pl);
```

world.c

```
#include <stdlib.h>

addObject(...)
{ ... }

remObject(...)
{ ... }

movePlayer(...)
{ ... }
```

graphics.h

```
extern drawObject(Ob);
extern drawPlayer(Pl);
extern spin(...);
```

graphics.c

```
#include <stdio.h>
#include "world.h"

drawObject(Ob o);
{ ... }

drawPlayer(Pl p)
{ ... }

fade(...)
{ ... }
```

Building with incremental compilation:

```
$ gcc -c -g -Wall world.c
$ gcc -c -g -Wall graphics.c
$ gcc -c -g -Wall main.c
$ gcc -Wall -o game main.o world.o graphics.o
```

Building Large C Program

For systems like Linux kernel with 50,000 files building is either

- inefficient (recompile everything after any change)
- error-prone (recompile just what's changed + dependents)
  - module relationships easy to overlook (e.g. `graphics.c` depends on a `typedef` in `world.h`)
  - you may not know when a module changes (e.g. you work on `graphics.c`, others work on `world.c`)

Example Makefile #1

A Makefile for the earlier example program:

```
game : main.o graphics.o world.o
        gcc -Wall -o game main.o graphics.o world.o

main.o : main.c graphics.h world.h
        gcc -c main.c

graphics.o : graphics.c world.h
        gcc -c -g -Wall graphics.c

world.o : world.c
        gcc -c -g -Wall world.c
```

Example Makefile #1

Easily parsed in Perl:

```
open my $makefile, '<', $file or die;
while (<$makefile>) {
  my ($target, $depends) = /(\S+)\s*:\s*(.*)/
     or next;
  $first_target = $target if !defined $first_target;
  $depends{$target} = $depends;
  while (<$makefile>) {
    last if !/^\t/;
    $build_cmd{$target} .= $_;
  }
}
```

How make Works

The make command behaves as:

```
make(target, sources, command):
    # Stage 1
    FOR each S in sources DO
        rebuild S if it needs rebuilding
    END
    # Stage 2
    IF (no sources OR
        any source is newer than target) THEN
       run command to rebuild target
    END
```

How make Works - Implementation in Perl

```perl
my ($target) = @_;
my $build_cmd = $build_cmd{$target};
die "*** No rule to make target $target\n" if
    !$build_cmd && !-e $target;
return if !$build_cmd;
my $target_build_needed = ! -e $target;
foreach $dep (split /\s+/, $depends{$target}) {
    build $dep;
    $target_build_needed ||= -M  $target > -M $dep;
}
return if !$target_build_needed;
print $build_cmd;
system $build_cmd;
```

Additional functionalities of Makefiles

```
# string-valued variables/macros
CC = gcc
CFLAGS = -g
LDFLAGS = -lm
BINS = main.o graphics.o world.o

# implicit commands, determined by suffix
main.o     : main.c graphics.h world.h
graphics.o : graphics.c world.h
world.o    : world.c

# pseduo-targets
clean :
        rm -f game main.o graphics.o world.o
            # or ... rm -f game $(BINS)
```

Additional functionalities of Makefiles

```
# multiple targets with same sources
stats1 stats2 : data1 data2 data3
        perl analyse1.pl data1 data2 data3 > stats1
        perl analyse2.pl data1 data2 data3 > stats2

# creating subsystems via make
parser:
    cd parser && $(MAKE)
        # assumes parser directory has own Makefile
```

Parsing Variables and comments in Perl

```perl
open MAKEFILE, $file or die;
while (<MAKEFILE>) {
  s/#.*//;
  s/\$\((\w+)\)/$variable{$1}||''/eg;
  if (/^\s*(\w+)\s*=\s*(.*)$/) {
    $variable{$1} = $2;
    next;
  }
  my ($target, $depends) = /(\S+)\s*:\s*(.*)/ or next;
  $first_target = $target if !defined $first_target;
  $depends{$target} = $depends;
  while (<MAKEFILE>) {
    s/\$\((\w+)\)/$variable{$1}||''/eg;
    last if !/^\t/;
    $build_cmd{$target} .= $_;
  }
}
```

Command-line Arguments

If `make` arguments are targets, build just those targets:

```
$ make world.o
$ make clean
```

If no args, build first target in the `Makefile`.
The `-n` option instructs `make`

- to tell what it would do to create targets
- but don't execute any of the commands

Command-line Arguments - Implementation in Perl

```perl
$makefile_name = "Makefile";
if (@ARGV >= 2 && $ARGV[0] eq "-f") {
    shift @ARGV;
    $makefile_name = shift @ARGV;
}
parse_makefile $makefile_name;
push @ARGV, $first_target if !@ARGV;
build $_ foreach @ARGV;
```

Example Makefile #2

Sample Makefile for a simple compiler:

```
CC     = gcc
CFLAGS = -Wall -g
OBJS   = main.o lex.o parse.o codegen.o

mycc : $(OBJS)
        $(CC) -o mycc $(OBJS)

main.o : main.c mycc.h lex.h parse.h codegen.h
        $(CC) $(CFLAGS) -c main.c

lex.o : lex.c mycc.h lex.h
        $(CC) $(CFLAGS) -c lex.c

parse.o : parse.c mycc.h parse.h lex.h
codegen.o : codegen.h mycc.h codegen.h parse.h

clean :
        rm -f mycc $(OBJS) core
```

Abbreviations

To simplify writing rules, `make` provides default abbreviations:

`$@`	full name of target
`$*`	name of target, without suffix
`$<`	full name of first source
`$?`	full names of all newer sources

Examples:

```
# one of above rules, re-written
lex.o : lex.c mycc.h lex.h
        $(CC) $(CFLAGS) -c $*.c -o $@
            # or ... $(CC) $(CFLAGS) -c $<< -o $@

# update a library archive
lib.a: foo.o bar.o lose.o win.o
    ar r lib.a $?
```

Abbreviations - Implementation in Perl

```
my %builtin;
$builtin{'@'} = $target;
($builtin{'*'} = $target) =~ s/\.[^\.]*$//;
$builtin{'^'} = $depends{$target};
($builtin{'<'} = $depends{$target}) =~  s/\s.*//;
$build_command =~ s/\$([@*^<])/$builtin{$1}||''/eg;
```