Building Software Systems

Software systems need to be built / re-built

- during the development phase (change,compile,test,repeat)
- if distributed in source code form (assists portability)

Simple example: systems implemented as multi-module C programs:

- multiple object files (`.o` files) and libraries (`libxyz.a`)
- compiler that only recompiles what it's told to

Multi-module C Programs

Large C programs are implemented as a collection of `.c` and `.h` files.

A pair of `a.c` and `a.h` generally defines a *module*:

- `a.c` contains operations related to one particular kind of data/object
- `a.h` exports definitions for types/operations defined in `a.c`

Why partition a program into modules at all?

- as a principle of good design
- to share development work in a large project
- to create re-usable libraries

Example Multi-module C Program

Imagine a large game program with

- an internal model of the game world (places, objects, actors)
- code to manage graphics (mapping the world to the screen)

The graphics code would ...

- typically be separated from the world code (software eng)
- need to use the world operations (to render current scene)

Then there would be a main program to ...

- accept user commands, modify world, update display

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 Large Programs

Building the example program is relatively simple:

```
% 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
```

For larger systems, 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`)

Classical tool to build software dating to '70s.
Many variants and alternatives but still useful widely used.
More recent tools heavily used for particular languages.
E.g for Java maven or ant very popular.

`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 Makefile #1

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

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:

```
sub parse_makefile( {
    my ($file) = @_;
    open 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:

```
sub build( {
    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

```
sub parse_makefile( {
    my ($file) = @_;
    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:

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

Generic Rules

Can define generic rules based on suffixes:

```
.SUFFIXES: .c .o .java .pl .sh


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

i.e. to make a `.o` file from a `.c` file, use the command ...
Rules for several common languages are built in to `make`.
Pattern-based rules generalise what suffix rules do.
E.g. implementation of `.c.o`

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

Make in Perl - Version #0

Simple Make implementation in Perl.
Parses makefile rules and storing them in 2 hashes.
Building is done with a recursive function.

/home/cs2041/public_html/code/make/make0.pl

```
#!/usr/bin/perl -w
# written by andrewt@cse.unsw.edu.au for COMP2041

# Simple Perl implementation of "make".
#
# It parses makefile rules and stores them in 2 hashes.
#
# Building is done with a recursive function.

$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;
exit 0;

sub parse_makefile {
    my ($file) = @_;
    open MAKEFILE, $file or die "Can not open $file: $!";
    while (<MAKEFILE>) {
        my ($target, $dependencies) = /(\S+)\s*:\s*(.*)/ or next;
        $first_target ||= $target;
        $dependencies{$target} = $dependencies;
        while (<MAKEFILE>) {
            last if !/^\t/;
            $build_command{$target} .= $_;
        }
    }
}

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

Testing make0.pl

```
% cd /home/cs2041/public_html/code/make
% ./make0.pl -f Makefile.simple world.o
% ./make0.pl -f Makefile.simple clean
    rm -f game main.o graphics.o world.o
% ./make0.pl -f Makefile.simple world.o
    gcc -c world.c
% ./make0.pl -f Makefile.simple
    gcc -c main.c
    gcc -c graphics.c
    gcc -o game main.o graphics.o world.o
%  ./make0.pl -f Makefile.simple
% ./make0.pl -f Makefile.simple clean
    rm -f game main.o graphics.o world.o
```

Make in Perl - Version #1

Add a few lines of code and we can handle variables and comments.
A good example of how easy some tasks are in Perl.

/home/cs2041/public_html/code/make/make1.pl

```
#!/usr/bin/perl -w
# written by andrewt@cse.unsw.edu.au for COMP2041

# Add a few lines of code to make0.pl
# and we can handle variables and comments.
#
# A good example of how easy some tasks are in 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;
exit 0;

sub parse_makefile {
    my ($file) = @_;
    open MAKEFILE, $file or die "Can not open $file: $!\n";
    while (<MAKEFILE>) {
        s/#.*//;
        s/\$\((\w+)\)/$variable{$1}||''/eg;
        if (/^\s*(\w+)\s*=\s*(.*)$/) {
            $variable{$1} = $2;
            next;
        }
        my ($target, $dependencies) = /(\S+)\s*:\s*(.*)/ or next;
        $first_target ||= $target;
        $dependencies{$target} = $dependencies;
        while (<MAKEFILE>) {
            s/#.*//;
            s/\$\((\w+)\)/$variable{$1}||''/eg;
            last if !/^\t/;
            $build_command{$target} .= $_;
        }
    }
}

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

Testing make1.pl

```
% cd /home/cs2041/public_html/code/make
% ./make1.pl -f Makefile.variables world.o graphics.o
    gcc-4.3 -O3 -Wall -c world.c
    gcc-4.3 -O3 -Wall -c graphics.c
% ./make1.pl -f Makefile.variables
    gcc-4.3 -O3 -Wall -c main.c
    gcc-4.3 -O3 -Wall -o game main.o graphics.o world.
```

Make in Perl - Version #2

Add a few lines of code and we can handle some builtin variables
and an implicit rule.
Another good example of how easy some tasks are in Perl.

/home/cs2041/public_html/code/make/make2.pl

```
#!/usr/bin/perl -w
# written by andrewt@cse.unsw.edu.au for COMP2041

# Add a few lines of code to make1.pl and we can handle some
# builtin variables and an implicit rule.
#
# Another  good example of how easy some tasks are in Perl.

$makefile_name = "Makefile";
if (@ARGV >= 2 && $ARGV[0] eq "-f") {
    shift @ARGV;
    $makefile_name = shift @ARGV;
}
%variable = (CC => 'cc', CFLAGS => '');
parse_makefile($makefile_name);
push @ARGV, $first_target if !@ARGV;
build($_) foreach @ARGV;
exit 0;

sub parse_makefile {
    my ($file) = @_;
    open MAKEFILE, $file or die "Can not open $file: $!";
    while (<MAKEFILE>) {
        s/#.*//;
        s/\$\((\w+)\)/$variable{$1}||''/eg;
        if (/^\s*(\w+)\s*=\s*(.*)$/) {
            $variable{$1} = $2;
            next;
        }
        my ($target, $dependencies) = /(\S+)\s*:\s*(.*)/ or next;
        $first_target = $target if !defined $first_target;
        $dependencies{$target} = $dependencies;
        while (<MAKEFILE>) {
            s/#.*//;
            s/\$\((\w+)\)/$variable{$1}||''/eg;
            last if !/^\t/;
            $build_command{$target} .= $_;
        }
    }
}

sub build {
    my ($target) = @_;
    my $build_command = $build_command{$target};
    if (!$build_command && $target =~ /(.*)\.o/) {
        $build_command = "$variable{CC} $variable{CFLAGS} -c \$< -o \$@\n";
    }
    die "*** No rule to make target $target\n" if !$build_command && !-e $target;
    return if !$build_command;
    my $target_build_needed = ! -e $target;
    foreach $dependency (split /\s+/, $dependencies{$target}) {
        build($dependency);
        $target_build_needed ||= -M  $target > -M $dependency;
    }
    return if !$target_build_needed;
    my %builtin_variables;
    $builtin_variables{'@'} = $target;
    ($builtin_variables{'*'} = $target) =~ s/\.[^\.]*$//;
    $builtin_variables{'^'} = $dependencies{$target};
    ($builtin_variables{'<'} = $dependencies{$target}) =~  s/\s.*//;
    $build_command =~ s/\$(.)/$builtin_variables{$1}||''/eg;
    print $build_command;
    system $build_command;
}
```

Testing make2.pl

```
% cd /home/cs2041/public_html/code/make
% ./make2.pl -f Makefile.builtin_variables clean
    rm -f game main.o graphics.o world.o
% ./make2.pl -f Makefile.builtin_variables
    gcc-4.3 -O3 -Wall -c main.c
    gcc-4.3 -O3 -Wall -c graphics.c
    gcc-4.3 -O3 -Wall -c world.c -o world.o
    gcc-4.3 -O3 -Wall -o game main.o graphics.o world.
% ./make2.pl -f Makefile.implicit  clean
    rm -f game main.o graphics.o world.o
% ./make2.pl -f Makefile.implicit
    gcc-4.3 -O3 -Wall -c main.c -o main.o
    gcc-4.3 -O3 -Wall -c graphics.c -o graphics.o
    gcc-4.3 -O3 -Wall -c world.c -o world.o
    gcc-4.3 -O3 -Wall -o game main.o graphics.o world.
```

Configuration

Systems need to be tailored to fit into specific computing environments.

If we're lucky, this is handled by the execution engine.

- although we'll may still need to change e.g. file/user names

If we're not so lucky, we need to ...

- embded stuff in the code to cater for each environment
- ensure the compiler uses the right set of switches

Best to do this automatically ...

Autotools

GNU build system for assisting with portability of open source systems.
Still widely used but many people dislike it.
Many newer build tools available (e.g. Cmake , ninja)

Autotools

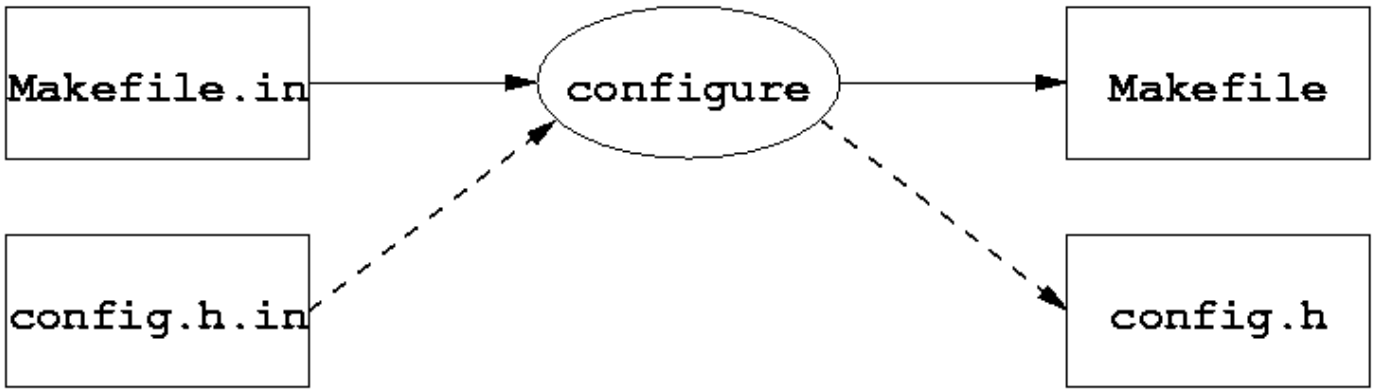
Uses a generic `configure` command to

- interrogate the local system and determine configuration settings
- read simple templates for `Makefile` and/or `config.h`
- generate customised `Makefile` and/or `config.h`

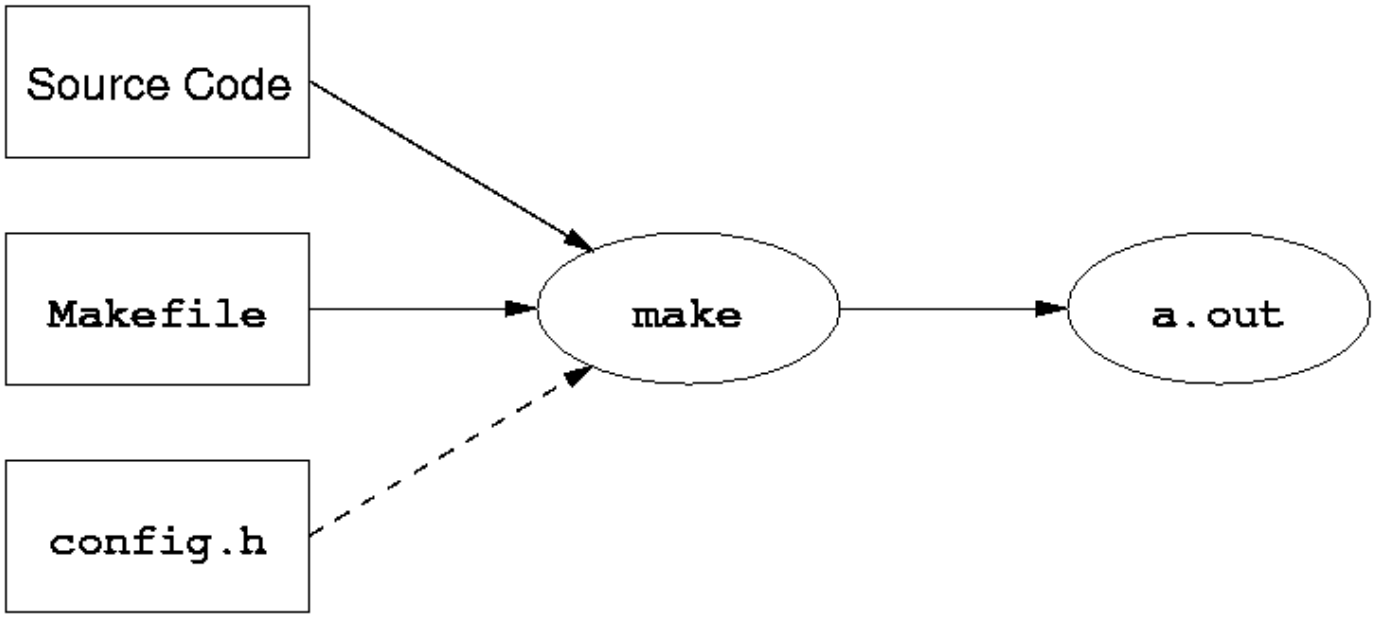
Once system is configured, simply run `make` to build it

`Configure` shell script is distributed with the source code.

Configure/Autoconf



Configure/Autoconf



Configure/Autoconf

If the `configure` script was completely general

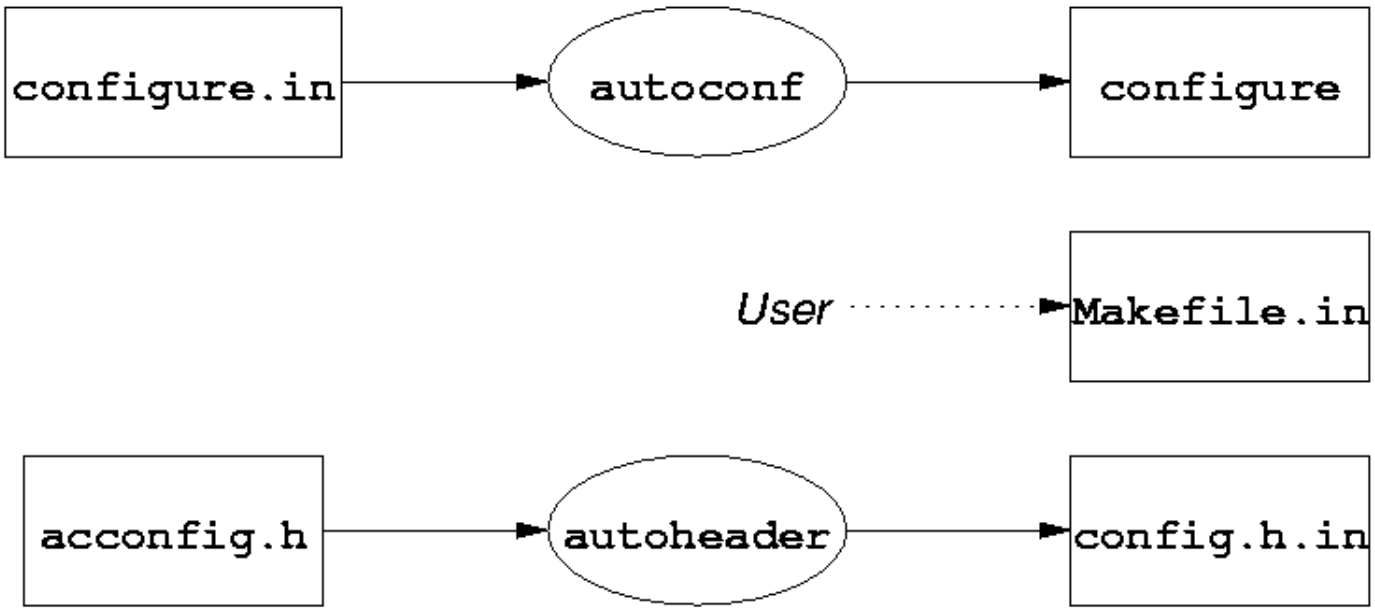
- it would be huge
- most of it would be irrelevant for a given system
- developers would need to tailor it for new systems

The solution: generate the `configure` script automatically ...

Configure/Autoconf

The `autoconf` command

- generates a `configure` script from a file called `configure.in`
- `configure.in` contains information such as
  - names of all source code files for this system
  - where should binaries and manuals be installed
  - which libraries/processors (e.g. `flex`) are needed



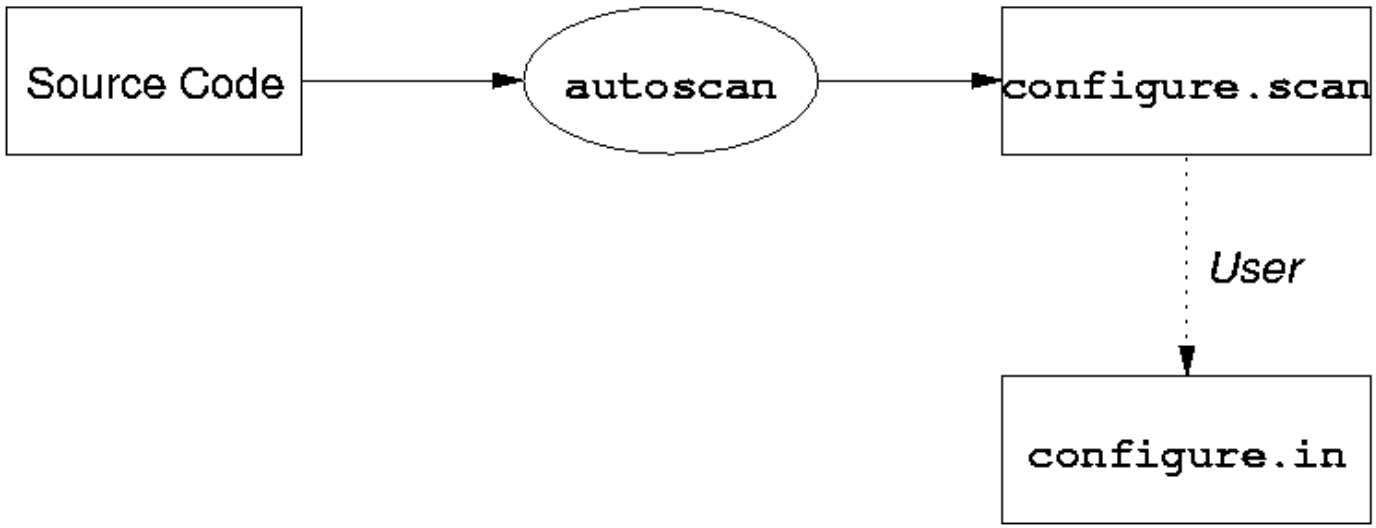
Configure/Autoconf

Creating a `configure.in` file ...

- requires developer to enter details about code
- much of this can be auto-extracted from code

Hence, the `autoscan` command ...

- extract config information from source code
- produce a "first draft" `configure.in` file
- developer manually completes `configure.in`



Configure/Autoconf

The `configure/autoconf` system is a nice example of automating, "routine" aspects of software development.
For more details about `configure/autoconf`, google or download software using it and experiment.