Open Source Development Tools

An Introduction to Make, Configure, Automake, Autoconf

Stefan Hundhammer <sh@suse.de>
10/2005

Novell

Content

- Manual compilation
- Introduction to the GNU tool chain and how the tools interact:
  - Make
  - Configure
  - Autoconf
  - Automake
  - Libtool (very short)

Intentions

- Give the audience a general idea what make, configure, autoconf, automake, and libtool are all about
- Provide a starting point to read the reference documentation
- Provide a basis to make decisions if those tools make sense for your own projects
- Get informed enough to participate in discussions about the subject

What it is Not

- An advanced expert lesson for make, configure, autoconf, automake, libtool
- A reference documentation for those tools
- A step-by-step guide how to integrate those tools into your own projects
- A complete listing of all involved utility programs

➜ If in doubt, RTFM

Required Knowledge

- A rough understanding of what a Linux or Unix system is all about
- Programming in any compiled language (C, C++, Pascal, ...)
- A rough understanding of C
- Basic Unix / Linux shell handling
- Explicitly no Unix / Linux guru knowledge required

The Beginner's Approach:

Manual Compilation

Compiling a Trivial C Program

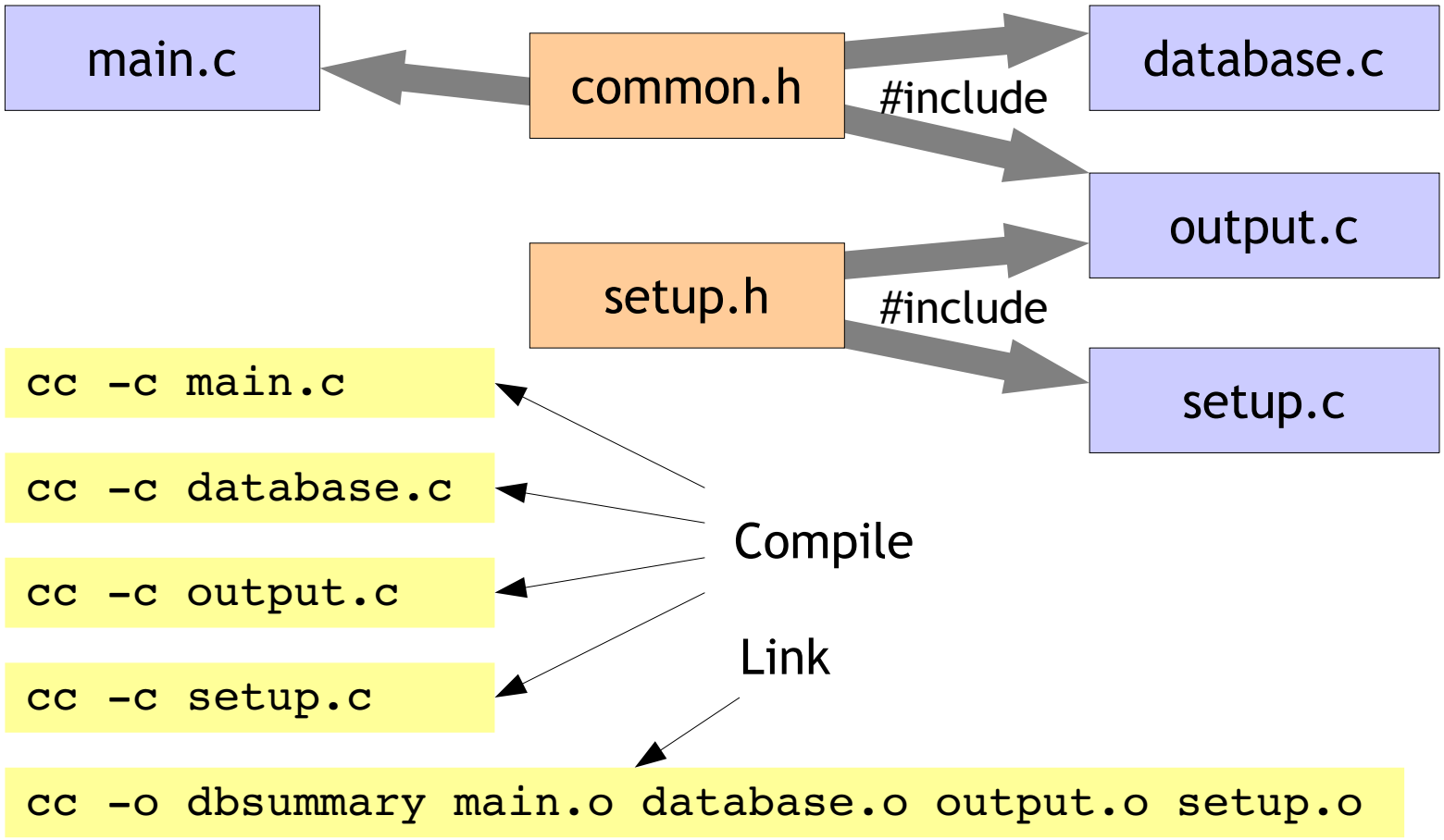
hello.c

```
include <stdio.h>

int main( int argc, char *argv[] )
{
    printf( "Hello, world!\n" );
}
```

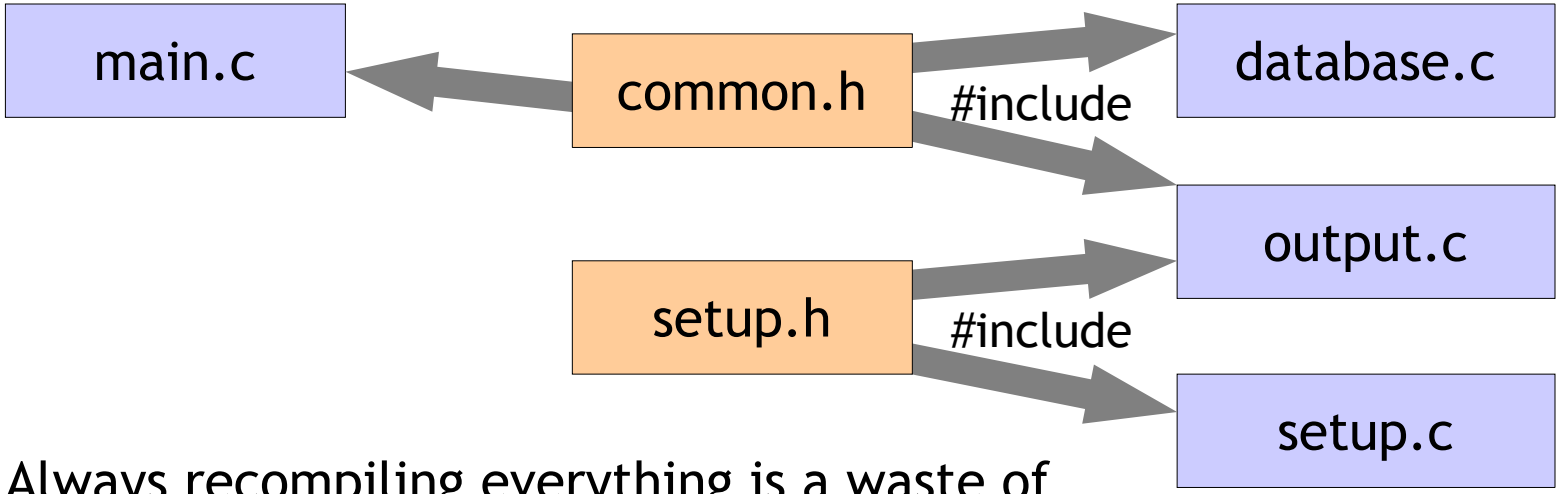
```
cc -o hello hello.c
```

- Compiles hello.c
- Links resulting object + C library (startup code, printf())
- Sets executable permission

Compiling a more Realistic C Program



Problem: When to Compile What?



Always recompiling everything is a waste of time and CPU power

-> recompile only what needs to be recompiled

- when it needs to be recompiled

Linking must always be done

-> A simple script is not an option

Automated Compilation:

Make

Make

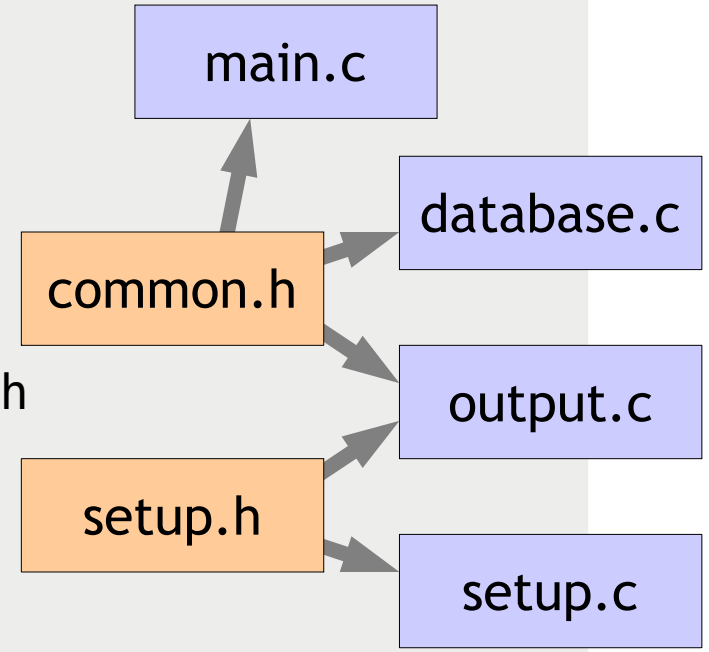
- Rule-based system to rebuild generated files when they are outdated
- Uses a file “Makefile” (by default) for rules
- Uses simple file system time stamps (the last change time of a file)
- Takes care of (specified) dependencies: Builds dependent files after the files they depend on are built (regardless in which order they are listed in the Makefile)
- Invoked simply with

```
make
```

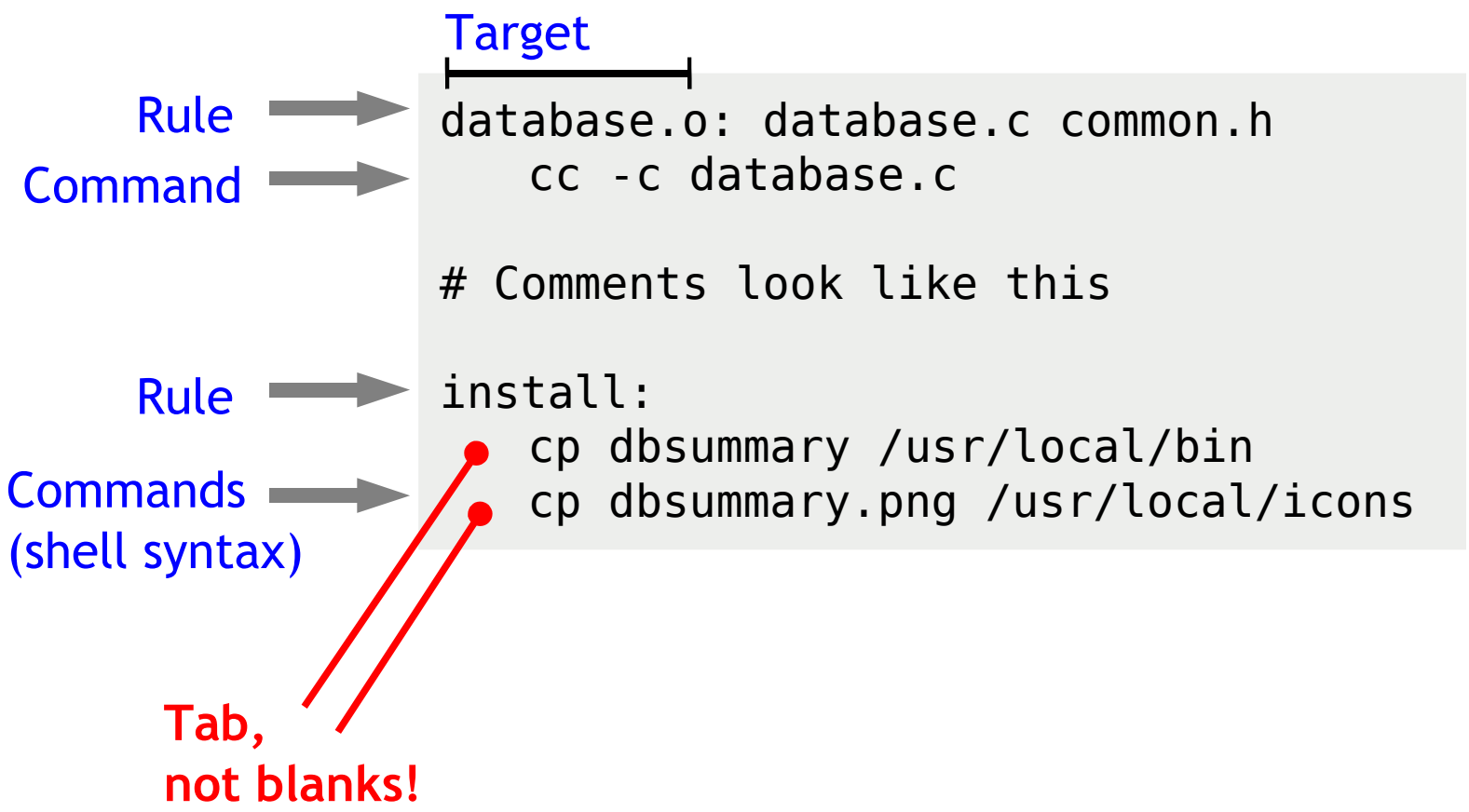
Makefile Example

Makefile

```
dbsummary: main.o database.o output.o setup.o
    cc -o dbsummary main.o database.o output.o setup.o

main.o: main.c common.h
    cc -c main.c

database.o: database.c common.h
    cc -c database.c

output.o: output.c common.h setup.h
    cc -c output.c

setup.o: setup.c setup.h
    cc -c setup.c
```



Makefile Syntax



Make Targets

- The first part of each rule is called a *target*
- *make* can be called with any target in the Makefile
- By default, *make* starts with the first target in the Makefile
- Common targets:
  - install
  - clean

```
make install
```

```
make clean
```

Makefile Variables

Makefile

```
OBJ = main.o database.o output.o setup.o

dbsummary: $(OBJ)
    cc -o dbsummary $(OBJ)

clean:
    rm -f $(OBJ)
```

- No explicit declaration
- Undefined variables expand to an empty string (no error)
- Variables are expanded within $( ) $(VAR)
- Variables are assigned without $( ) VAR = ...

Makefile Variables Pitfall

Makefile

```
OBJ = helper_main.o helper.o

helper: $(OBJ)
    cc -o helper $(OBJ)

OBJ = main.o func.o db.o

myprog: $(OBJ)
    cc -o myprog $(OBJ)
```

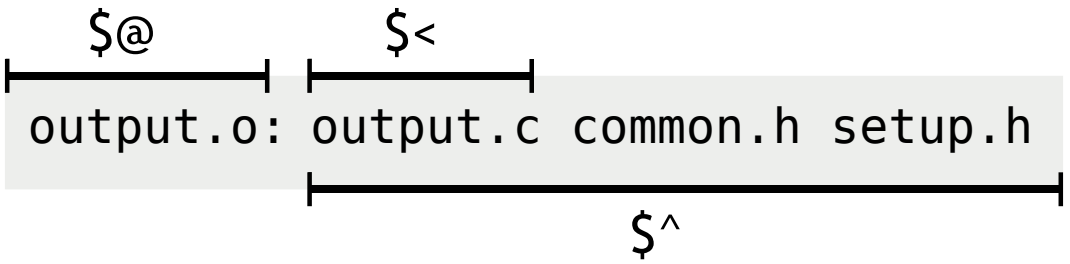
$(OBJ) will contain "main.o func.o db.o" for all (!) uses

This will not work: The last assignment to a variable is used for all (!) occurences of that variable

Automatic Variables

For every rule in a Makefile, a number of variables is automatically defined. The most important are:

- $@ the current target
- $< the first dependent file
- $^ all dependent files



Using Automatic Variables

```
dbsummary: main.o database.o output.o setup.o
    cc -o dbsummary main.o database.o output.o setup.o

output.o: output.c common.h setup.h
    cc -c output.c
```

Avoiding error-prone duplicating

```
dbsummary: main.o database.o output.o setup.o
    cc -o $@ $^

output.o: output.c common.h setup.h
    cc -c $<
```

Common Make Variables

- $(CC) C compiler (default: “cc”)
- $(CFLAGS) flags (command line options) for the C compiler
- $(CPP) C preprocessor
- $(CPPFLAGS) flags for the C preprocessor
- $(CXX) C++ compiler (default: “g++”)
- $(CXXFLAGS) flags for the C++ compiler
- $(LD) linker (obsolete, use $(CC) instead)
- $(LDFLAGS) linker flags (used for libs + library paths etc.)

Rule of thumb: All programs that are invoked in a Makefile should be specified as a variable.

If a program commonly needs command line flags, there should be a separate variable for the flags.

Example: Compiler options (warning level, include paths, ...)

Suffix Rules

Generic rule to make a .o file from a .c file:

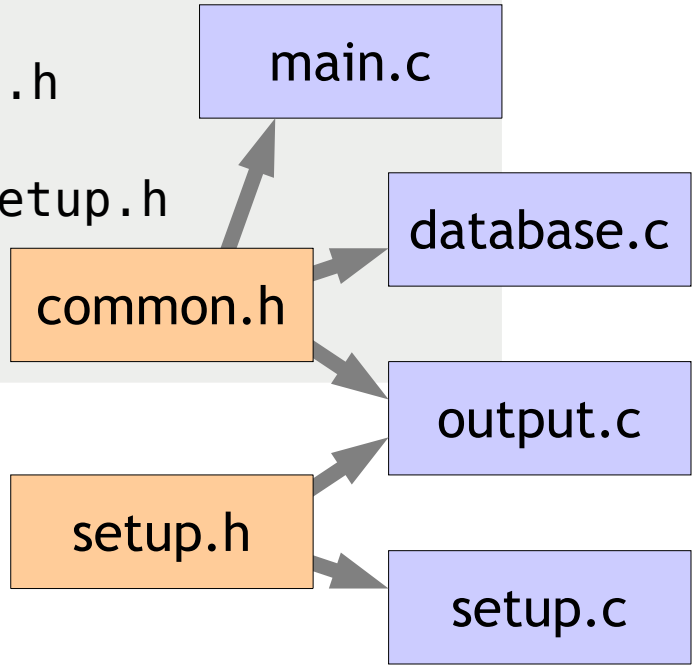
```
.c.o:
    $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

This rule (and many similar rules for common file suffixes) is predefined
-> no need to specify the compile commands in Makefiles

Makefile Example Revisited

Makefile

```
dbsummary: main.o database.o output.o setup.o
    $(CC) $(LDFLAGS) -o $@ $^

main.o: main.c common.h

database.o: database.c common.h

output.o: output.c common.h setup.h

setup.o: setup.c setup.h
```



Make Conclusion

- Pro
  - A lot more efficient than simply always recompiling everything
  - Powerful and flexible
  - Make can be used independently of any more advanced tools (autoconf, automake)
- Con
  - Makefiles can easily get very complex
  - Lots of small details to take care of (compiler flags, utility paths, ...) that make the result nonportable
  - Does not make dependencies any easier (header files, maybe included in multiple levels)

Portability among Open Systems:

Configure

Building Open Source Packages

- Download package sources
  ("tarball" - a (usually) compressed .tar archive)

```
wget ftp://ftp.any.org/pub/mypkg-1.2.tar.gz
```

- Unpack tarball

```
tar xzf mypkg-1.2.tar.gz
```

- Configure

```
cd mypkg-1.2.3
./configure
```

- Compile

```
make
```

- Install
  (with root privileges)

```
su
make install
```

A Configure Run

```
checking build system type... i686-pc-linux-gnu
checking for a BSD-compatible install... /usr/bin/instal
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking whether we are cross compiling... no
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
...
checking for egrep... grep -E
checking for ld used by gcc... /usr/i586-suse-linux/bin/ld
checking for BSD-compatible nm... /usr/bin/nm -B
checking whether ln -s works... yes
checking for ANSI C header files... yes
checking for sys/types.h... yes
checking for stdlib.h... yes
checking for string.h... yes
checking for unistd.h... yes
...
checking size of int... 4
checking size of short... 2
checking size of long... 4
...
checking for X... libraries /usr/X11R6/lib, headers /usr/X11R6/include
checking for Qt... libraries /usr/lib/qt3/lib, headers /usr/lib/qt3/include using -mt
...
creating Makefile
```

```
checking build system type...
i686-pc-linux-gnu
```

```
checking whether we are using
the GNU C compiler... yes
```

```
checking for string.h... yes
checking for unistd.h... yes
```

```
checking size of int... 4
```

```
checking for X...
libraries /usr/X11R6/lib
```

The Purpose of Configure

- System checks
- Checks for certain installed programs (compiler, linker, helper utilities, ... - a working development environment)
- Checks for system header files
- Checks for system libraries
- Checks for lengths of fundamental data types (int, short, long, pointers, ...)
- Results are stored in *make* variables and in a generated header file *config.h*

Historical Background of Configure

- In the history of Unix-type systems (since ~1970), many different variants were created:
  - AT&T: Unix
  - University of Berkeley: BSD-Unix
  - Sun: SunOS, Solaris
  - Hewlett-Packard: HP-UX
  - IBM: AIX
  - ...
  - Linux
- Most of those variants were slightly different:
  - System library calls + include files
  - Command line tools
  - Compiler (+ compiler options)

Historical config.h

- Software packages that should be portable among different Unix variants were distributed with an include file with all the system specifics

config.h

```
#define HAVE_STRING_H 1
#define HAVE_UNISTD_H 1
#define SIZEOF_INT 4
...
```

- The site administrator who wanted to build (compile) that package had to edit this include file for his system

  Problem: This requires in-depth knowledge of the target system

First Versions of Configure

- Simple shell script to find out the values for config.h automatically
- Restricted to the least common denominator of all possible target systems – i.e. no more advanced tools could be used that the most basic of the target systems provided:
  - Simple (Bourne) shell
  - Simplest versions of shell commands: sed, tr, grep
  - No Perl or other more advanced interpreters
  - Very basic compiler for tests for include files and libs

Tests for Header Files

- A small C program is generated on the fly that includes the header file
- If the program can be compiled without error, the header file is available

```
#include <unistd.h>

int main()
{
    return 0;
}
```

Will throw compile error if header file is not available

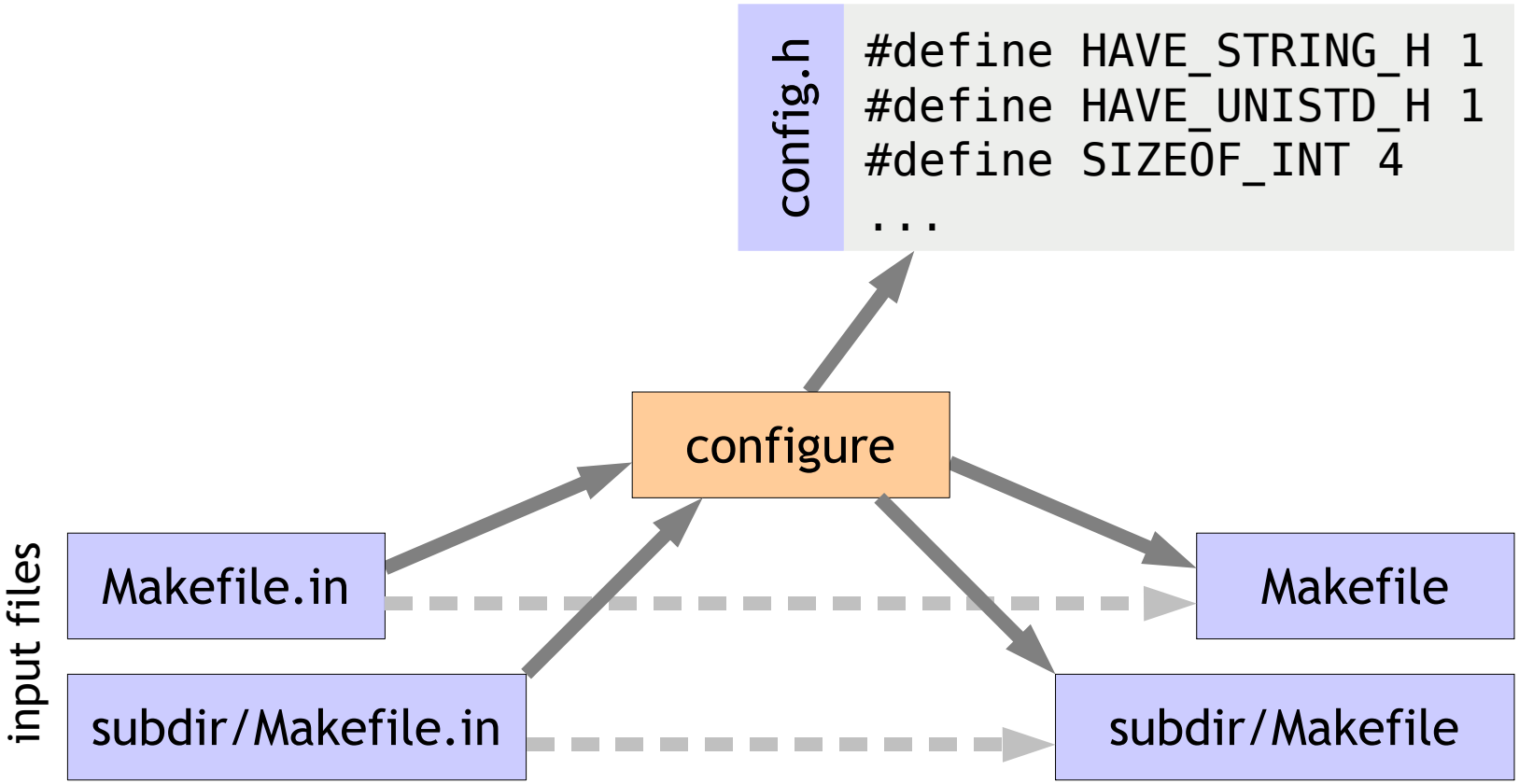
Tests for Libraries

- A small C program is generated on the fly that uses a typical function of that library
- If the program can be linked without error, the library is available

```
#include <X11/Xlib.h>

int main()
{
    XOpenDisplay("");
    return 0;
}
```

Will throw linker error if this function is not available

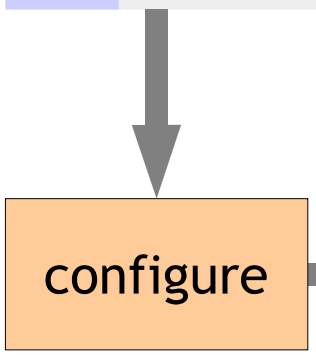
Files Generated by Configure



Makefile.in Content

Makefile.in

```
CC=@CC@
CFLAGS=@CFLAGS@
INSTALL=@INSTALL@
...
myprog: main.o output.o setup.o
    $(CC) $(LDFLAGS) -o $@ $^
```



Makefile

```
CC=gcc
CFLAGS=-ansi -Wall -O2
INSTALL=/usr/bin/install -c -p
...
myprog: main.o output.o setup.o
    $(CC) $(LDFLAGS) -o $@ $^
```

Configure Conclusion

- Pro
  - Solves make's problems with compiler flags and utility paths
  - Can help to make programs a lot more portable among different Unix-type systems
- Con
  - A configure script is very hard to write manually
  - Hard to maintain, yet requires constant maintenance as target systems evolve
  - Requires in-depth knowledge of all possible target systems
  - Requires Unix guru level 12 (on a scale 1..10)

Creating Configure Scripts:

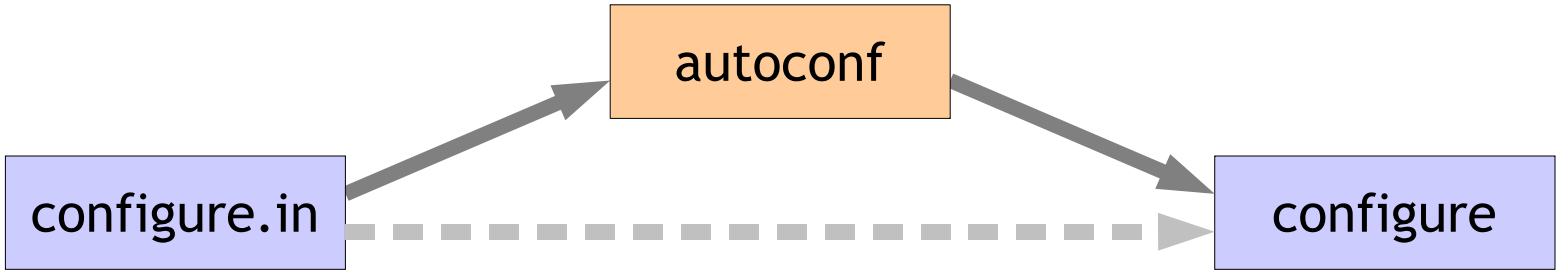
Autoconf

Autoconf

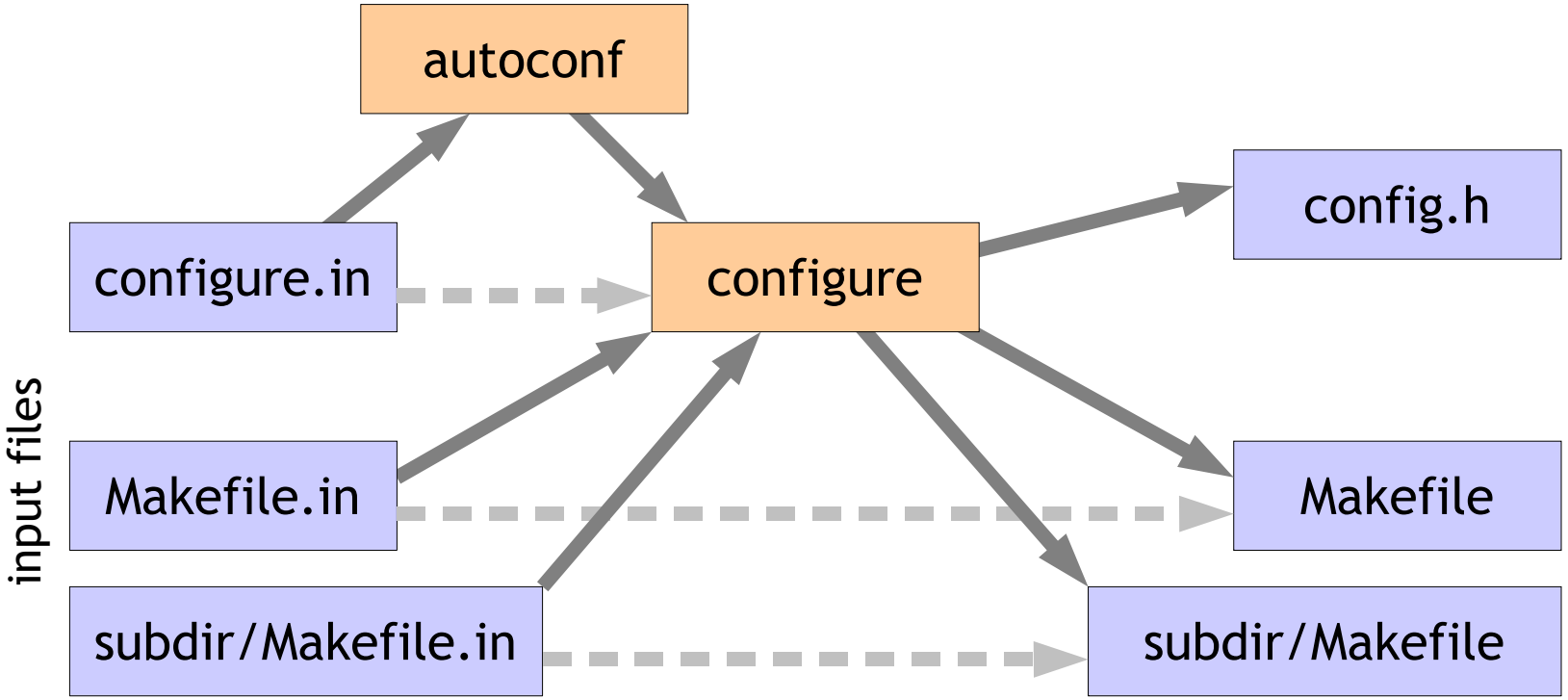
Configure scripts:

- Always do more or less the same
- Long and sometimes complex
  -> difficult to write
- Require changes as target systems change with each new version

-> Don't write them manually, generate them



Files Generated by Configure and Autoconf



configure.in Content

- Initializations
- Symbolic names for directory paths
- Tests for programs (Compilers, Lex, YaCC, install, ...)
- Tests for libraries
- Tests for header files
- Custom tests
- Which files are to be generated
  (list each Makefile in each subdirectory)
- Instruction to generate the file

configure.in Example

configure.in

```
AC_INIT(MyProjectName, 0.42, bugs@my.org)
AC_CONFIG_SRCDIR([Readme.MyProject])
AC_CONFIG_HEADER([config.h])

AC_PROG_CXX
AC_PROG_CPP
AC_PROG_INSTALL

AC_HEADER_STDC
AC_CHECK_HEADERS([string.h unistd.h limits.h])

AC_CONFIG_FILES([Makefile
                 doc/Makefile
                 src/Makefile])
AC_OUTPUT
```

configure.in Syntax

- Autoconf uses the M4 macro processor
- The M4 syntax is very bizarre
- It is simple text substitution, not an interpreter
- Lots of quoting is needed – quote with [ ]
- Everything that does or might contain whitespace or commas has to be quoted
- Autoconf uses include files for lots of predefined macros (aclocal.m4, acinclude.m4, system-wide)
- Writing custom autoconf M4 macros is not for the faint of heart – avoid if possible

Predefined Autoconf Tests: Programs
(Complete List)

- AC_PROG_CC — C compiler
- AC_PROG_CPP — C preprocessor
- AC_PROG_CXX — C++ compiler
- AC_PROG_LEX — lex (or flex, the GNU version)
- AC_PROG_YACC — yacc (or bison, the GNU version)
- AC_PROG_INSTALL — install
- AC_PROG_LN_S — ln -s (symbolic linking)
- AC_PROG_EGREP — egrep (extended regular expression search)
- AC_PROG_FGREP — fgrep (non-regexp text search)
- AC_PROG_AWK — awk interpreter
- AC_PROG_RANLIB — ranlib ((static) library helper utility)

(see also *info autoconf*)

Predefined Autoconf Tests: System Library Functions (Incomplete List)

• AC_FUNC_CHOWN	chown()	change owner of file
• AC_FUNC_FORK	fork()	start new process
• AC_FUNC_MALLOC	malloc()	memory allocation
• AC_FUNC_MEMCMP	memcmp()	compare memory blocks
• AC_FUNC_MKTIME	mktime()	convert time struct to time_t (seconds since 1.1.1970)
• AC_FUNC_REALLOC	realloc()	memory allocation
• AC_FUNC_STAT	stat()	file statistics (size, owner, permissions, mod. time)
• AC_FUNC_STRLEN	strlen()	length of zero-terminated C string

Other Predefined Autoconf Tests

- AC_HEADER_XXX Test for header file XXX

```
#if HAVE_UNISTD_H
#   include <unistd.h>
#endif

#if HAVE_STRING_H
#   include <string.h>
#endif
```

This does not make the code any prettier, but maybe portable (unless the target system is very broken)

- AC_STRUCT_XXX Test for system structure XXX
- AC_TYPE_XXX Test for system type XXX
  usually with simple replacement,
  if not available (int, void *, ...)

Generic Autoconf Tests

AC_CHECK_TOOL(VAR, PROG, FALLBACK)

- Looks for a program called PROG in $PATH
- Assigns the name (without path) of the program found to VAR
- Uses FALLBACK if PROG could not be found

```
AC_CHECK_TOOL( XMLLINT, xmllint, false )
```

-> XMLLINT=”xmllint” (if found)
or XMLLINT=”false” (if not found)

Using *false* as fallback ensures there will be an error later during *make* if the content of the variable is actually used in a Makefile - and only then.
The underlying philosophy is “the show must go on”.

Improved Error Handling

```
AC_CHECK_TOOL( XMLLINT, xmllint, false )
test “x${XMLLINT}” = “xfalse” && \
AC_MSG_ERROR([xmllint not found])
```

xmllint not found -> $XMLLINT = false
-> test succeeds -> AC_MSG_ERROR() is called
-> configure exits with error

This does **not** work:

```
AC_CHECK_TOOL( XMLLINT, xmllint,
               [AC_MSG_ERROR([xmllint not found])])
```

Will throw error, but not stop configure

Using Custom Autoconf Variables

configure.in

```
myprojdir=${prefix}/usr/share/MyProj
stylesheetdir=${myprojdir}/stylesheets
AC_SUBST(myprojdir)
AC_SUBST(stylesheetdir)
```

AC_SUBST exports an autoconf symbol to the generated files:

Makefile.in

```
STYLESHEETDIR=@stylesheetdir@
STYLESHEETS=myproj.css
...
install:
   $(INSTALL) $(STYLESHEETS) $(STYLESHEETDIR)
```

Autoscan

- Generates template *configure.scan* for *configure.in*
- Simply invoke it on the project toplevel directory:

```
autoscan
```

- Edit the generated file:
  - Project name, version, bug mail address
  - Unique file in project toplevel directory in AC_CONFIG_SRCDIR
  - Check if the generated tests make sense in your project
  - Add custom tests or symbols for directories
- Rename

```
mv configure.scan configure.in
```

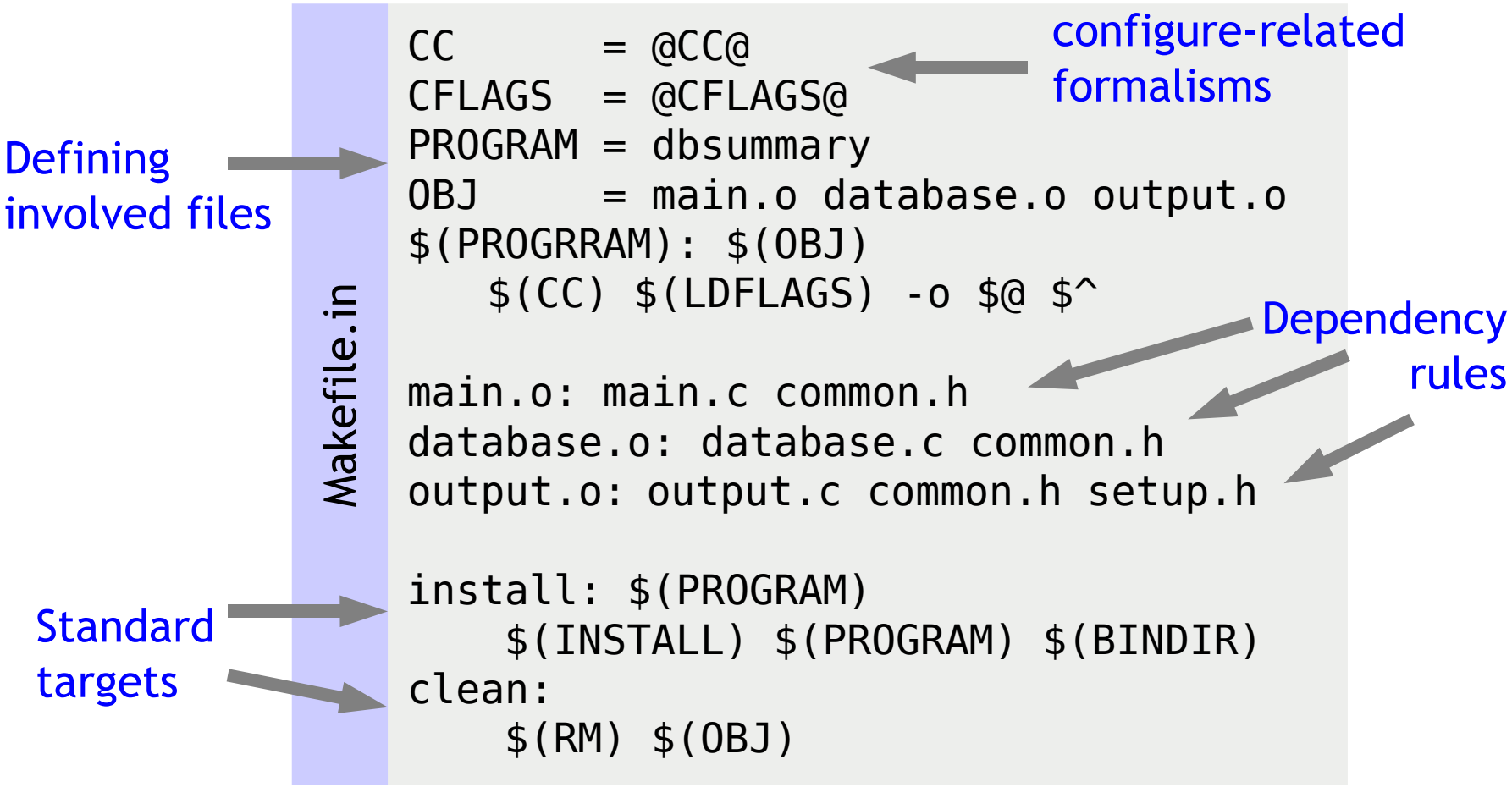
Autoconf Conclusion

- Pro
  - Autoconf takes the horror out of writing configure scripts
  - Realistically, only with a tool like autoconf an average programmer can write a configure script at all
  - Easy bootstrapping with autoscan and a little editing
- Con
  - Yet another level of indirection
  - Uses the M4 macro processor with its weird syntax
  - Makefile.in files tend to be even more complex (albeit more portable) than simple Makefiles
  - Does not address the problem of complexity of make rules at all

Tackling Makefile Complexity:

Automake

What Makes a Makefile(.in) so complex?



What Makes a Makefile(.in) so complex?

- Configure-related formalisms
  - Follow strict rules - could easily be generated
- Standard targets
  - Can simply be copied from a fixed set of rules
- Dependency rules
  - Hard to keep track manually – requires a tool anyway
  - Modern compilers (gcc) can generate direct and indirect dependencies to header files
- Defining involved files
  - Must be done manually

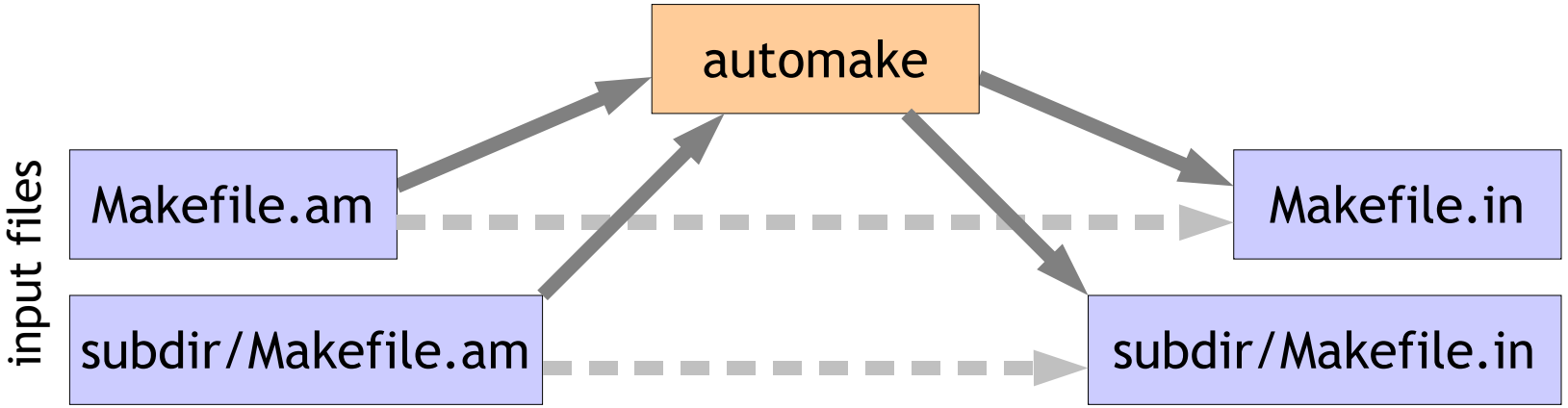
This is all that remains,
everything else
can be done automatically

RFC 1925

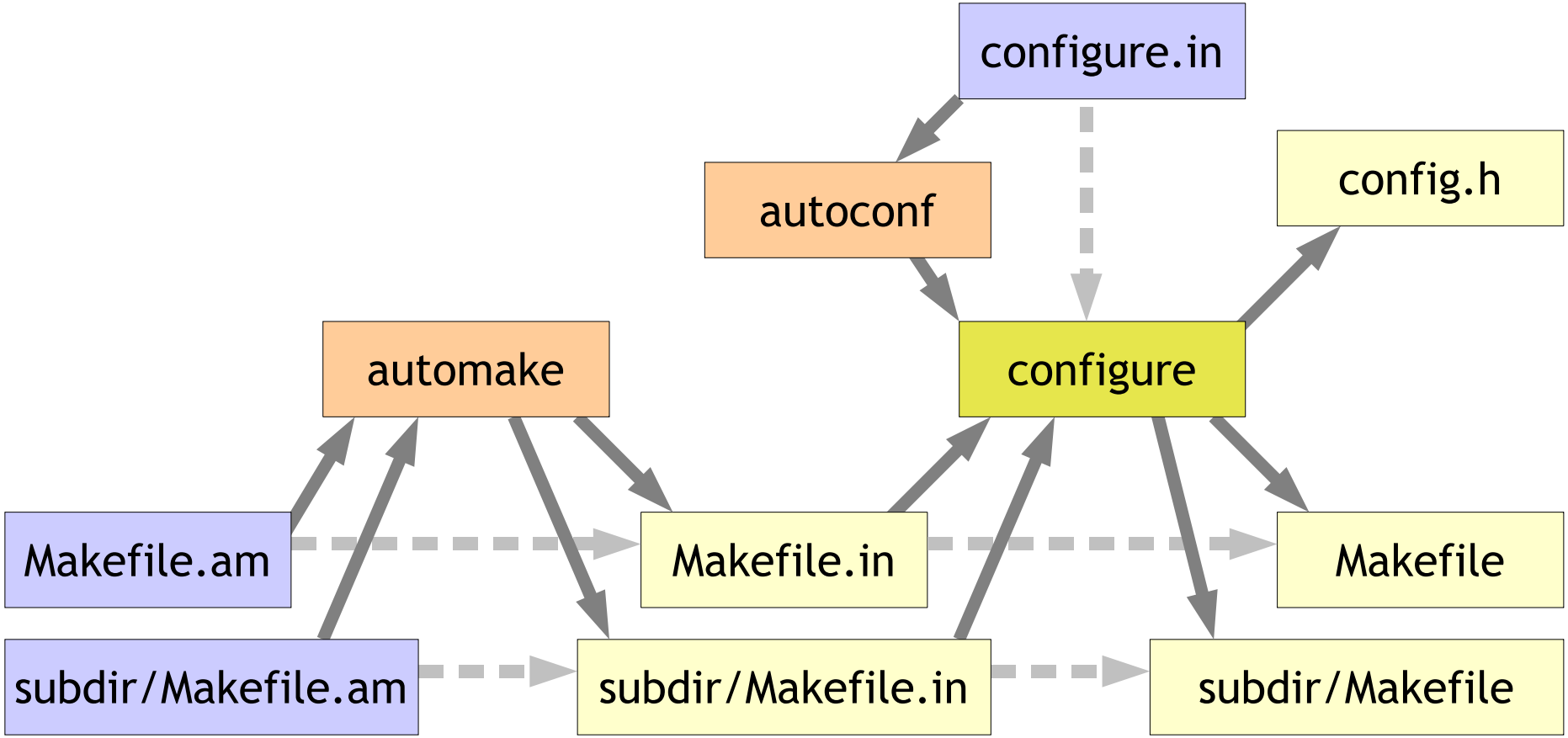
http://www.faqs.org/rfcs/rfc1925.html

```
...
(6a) It is always possible to add another level of
indirection.
...
(8) It is more complicated than you think.
...
With sufficient thrust, pigs fly just fine.

However, this is not necessarily a good idea. It
is hard to be sure where they are going to land,
and it could be dangerous sitting under them as
they fly overhead.
...
```

Automake



The Complete Process



Makefile.am Example

Makefile.am

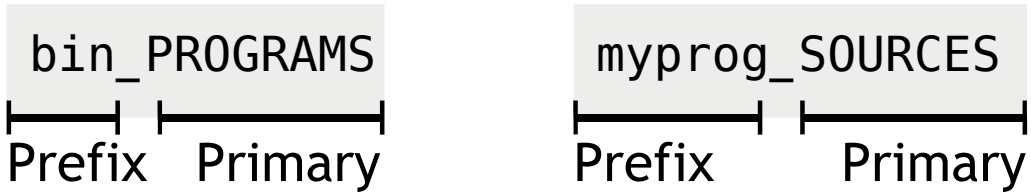
```
bin_PROGRAMS = dbsummary

dbsummary_SOURCES = main.c database.c \
                    output.c setup.c

noinst_HEADERS = common.h setup.h
```

This is all that is left of that lengthy example!

Standard targets are also automatically created:

- install
- clean
- dist
- distclean
- ...

Automake Variables



Common Primaries:

- _PROGRAMS
- _LIBRARIES
- _SCRIPTS
- _SOURCES
- _HEADERS
- _OBJECTS
- _DATA
- _LDADD

Generic Prefixes:

- bin_ will be installed to *bindir*
- sbin_ will be installed to *sbindir*
- lib_ will be installed to *libdir*
- noinst_ will not be installed
- EXTRA_ will be packaged upon *make dist*
- check_ used only for *make check* (for testsuites)

Building Programs with Automake

- Define a PROGRAMS primary
- Determine the prefix (where to install it):
  - bin ${prefix}/bin (-> /usr/bin)
  - sbin ${prefix}/sbin (-> /usr/sbin)
  - libexec ${prefix}/libexec (-> /usr/libexec)
  - noinst not at all

  ${prefix} is by default /usr/local or /usr
  can be set in configure.in with AC_PREFIX_DEFAULT
  can be overridden with

```
configure --prefix=/somewhere/else
```

- List all programs that are to be built
  (and installed to the same directory)

Adding Source Dependencies

Makefile.am

```
bin_PROGRAMS = foo bar

foo_SOURCES = fmain.cc fcalc.cc fcommon.h

bar_SOURCES = bmain.c butil.c
```

- Define a primary SOURCES for each program
- The SOURCES prefix is the program name
- If the program name contains special characters, it will be *canonicalized*: Special characters are transformed to underscores
  - my-prog -> my_prog_SOURCES
  - myc++comp -> myc__comp

Header Files

- Header files can be added to SOURCES, if the headers are project internal anyway and should not be installed
- Alternatively, they can be added to noinst_HEADERS
- Header files that are to be installed to the target system should be added to include_HEADERS
- include_HEADERS will be installed to ${prefix}/include (-> /usr/include) by default
- You can override that directory with

```
includedir=/somewhere/else/include

include_HEADERS = foocommon.h fooutil.h
```

Linking against Add-on Libraries

Use _LDADD:

Makefile.am

```
bin_PROGRAMS    = qhello

qhello_SOURCES = qhello.cc qutil.cc

qhello_LDADD    = -L$(QTLIBDIR) -lqt-mt \
                  -lX11 -lXext -lSM -lICE
```

- Portability might become a problem when adding very system-specific libraries
- Use a (custom) autoconf check for such libraries and their paths

Building Libraries

Makefile.am

```
lib_LIBRARIES = libfoo.a

libfoo_a_SOURCES = futil.c fcalc.c fxy.c

include_HEADERS = foo.h
```

- Static libraries are just as easy to build as programs
- Watch out for canonical names:
  libfoo.a -> libfoo_a_SOURCES
- Automake will take care of running “ar” and “ranlib”
- Use libfoo_a_LIBADD for binary only object files
- For shared libraries, use libtool and LTLIBRARIES

Custom Files

Makefile.am

```
legalesedir = $(prefix)/share/doc/myproj

legalese_DATA = LICENSE COPYING

EXTRA_DIST = $(legalese_DATA)
```

- Define as DATA
- Add a (user defined) name as prefix to install the files or noinst_ to not install them
- Define the directory name as variable with “dir” (xxxdir, not xxx_dir !)
- Add the files to EXTRA_DIST to package them upon *make dist*

Managing Directory Structures

```
SUBDIRS = src doc icons

...more rules...
```

- Add subdirectories to SUBDIRS in Makefile.am
- Don't forget to add a new subdirectory to the AC_CONFIG_FILES section of configure.in, too!
- By default, the current directory is processed after all subdirectories are finished
- The current directory can be added as “.” to SUBDIRS if a different processing order is desired

Distribution Tarballs

- Upon make dist, a tarball of the project is made (with the current version number from *configure.in*):

  myproj-0.42.tar.gz
- All SOURCES and HEADERS are automatically included
- Everything else needs to be added to EXTRA_DIST
- Alternatively, an additional prefix *dist_* can be used (dist_DATA)
- The tarball also includes the *configure* script and all *Makefile.in* files

  Rationale: Complex tools like automake and autoconf are for developers and maintainers, users should not need to know about them; *configure* OTOH cannot be avoided

Miscellaneous

- Add generated files to CLEANFILES so they can be included in the *make clean* rule:

```
CLEANFILES = foo.moc bar.moc
```

- Custom make targets can be added as desired:

```
bin_PROGRAMS  = hello
hello_SOURCES = hello.c

debug:
    echo “bindir: $(bindir)”
```

- Automake has to be initialized in *configure.in*:

```
AC_INIT( ... )
AM_INIT_AUTOMAKE
```

Making Shared Libraries:

Libtool

Introducing Libtool

- Creating a shared library is more complex than creating a simple binary program or a static library:
  - The code has to be relocatable, i.e., compiled in a special way (-fpic for GNU C)
  - Shared libs are handled radically different in different variants of Unix-type systems
  - You might need more (and pretty arcane) compiler and linker options on some systems
- Libtool is a tool to make all that just as easy as creating a static lib
- Automake has extensive libtool support

Building Shared Libraries

Use LTLIBRARY (for LibTool Library):

Makefile.am

```
lib_LTLIBRARIES = libfoo.so

libfoo_so_SOURCES = futil.c fcalc.c fxy.c

include_HEADERS = foo.h
```

Yet Another Abstraction Layer

- Libtool wraps all shared-lib dependent actions
- The user no longer deals directly with a .so file, but with a .la file
- You need to call the libtool binary for some actions (starting, debugging)

```
libtool ––mode=exec gdb mylibtest
```

- Libtools takes care of inter-library dependencies (lib a requires lib b, ...) when linking
- Libtool handles LD_LIBRARY_PATH for program execution

Useful Utilities

config.status

- Generated by *configure* (in the project toplevel directory)
- Recreates the files configure generates without all the checks *configure* performs
- Useful if the system configuration has not changed since the last *configure* run
- Is called by make by *automake*'s rules if *Makefile.am* changes
- Manual invocation:

```
automake
./config.status
```

autoreconf

Calls all the involved tools in the correct order:

- aclocal (copies system M4 macros to project directory)
- libtoolize (adds libtool support to project)
- autoheader (generates config.h template)
- automake
- autoconf

Getting it all Started

The Easy Way

- Use KDevelop
- You can start lots of different kinds of projects with KDevelop:
  - KDE
  - Qt
  - GTK+
  - Non-GUI
- Downside: The resulting generated files (configure.in, Makefile.am) will be pretty complex right from the start
- You may need to edit those files at a later time

For Real Men

- Follow the instructions in the online documentation to automake and autoconf

```
info automake
info autoconf
```

- Alternatively: Follow the instructions in

  Vaughan, Elliston, Tromey, Taylor:
  GNU Autoconf, Automake, and Libtool
  New Riders Press

Questions?

Reference

- This presentation is available at:

  http://www.suse.de/~sh/automake
- Linux online documentation (info pages):
  - info make
  - info automake
  - info autoconf
- Vaughan, Elliston, Tromey, Taylor:
  GNU Autoconf, Automake, and Libtool
  New Riders Press

  also available online at:

  http://sources.redhat.com/autobook

Novell®

Novell