The Makefile utility

(Extract from the slides by
Terrance E. Boult
http://vast.uccs.edu/~tboult/)

Motivation

- Small programs ⟶ single file
- “Not so small” programs :
  - Many lines of code
  - Multiple components
  - More than one programmer

Motivation – continued

- Problems:
  - Long files are harder to manage (for both programmers and machines)
  - Every change requires long compilation
  - Many programmers can not modify the same file simultaneously
  - Division to components is desired

Motivation – continued

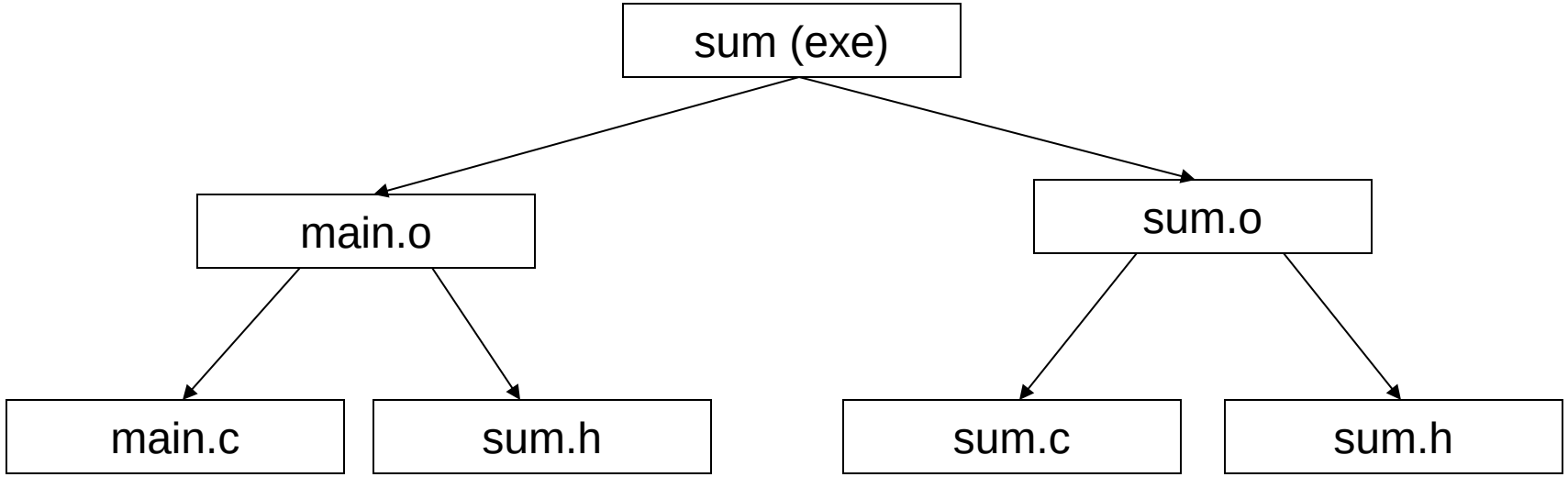
- Solution : divide project to multiple files
- Targets:
  - Good division to components
  - Minimum compilation when something is changed
  - Easy maintenance of project structure, dependencies and creation

Project maintenance

- Done in Unix by the Makefile mechanism
- A makefile is a file (script) containing :
  - Project structure (files, dependencies)
  - Instructions for files creation
- The make command reads a makefile, understands the project structure and makes up the executable
- Note that the Makefile mechanism is not limited to C programs

Project structure

- Project structure and dependencies can be represented as a DAG (= Directed Acyclic Graph)
- Example :
  - Program contains 3 files
  - main.c., sum.c, sum.h
  - sum.h included in both .c files
  - Executable should be the file sum


sum (exe)
main.o
sum.o
main.c
sum.h
sum.c
sum.h

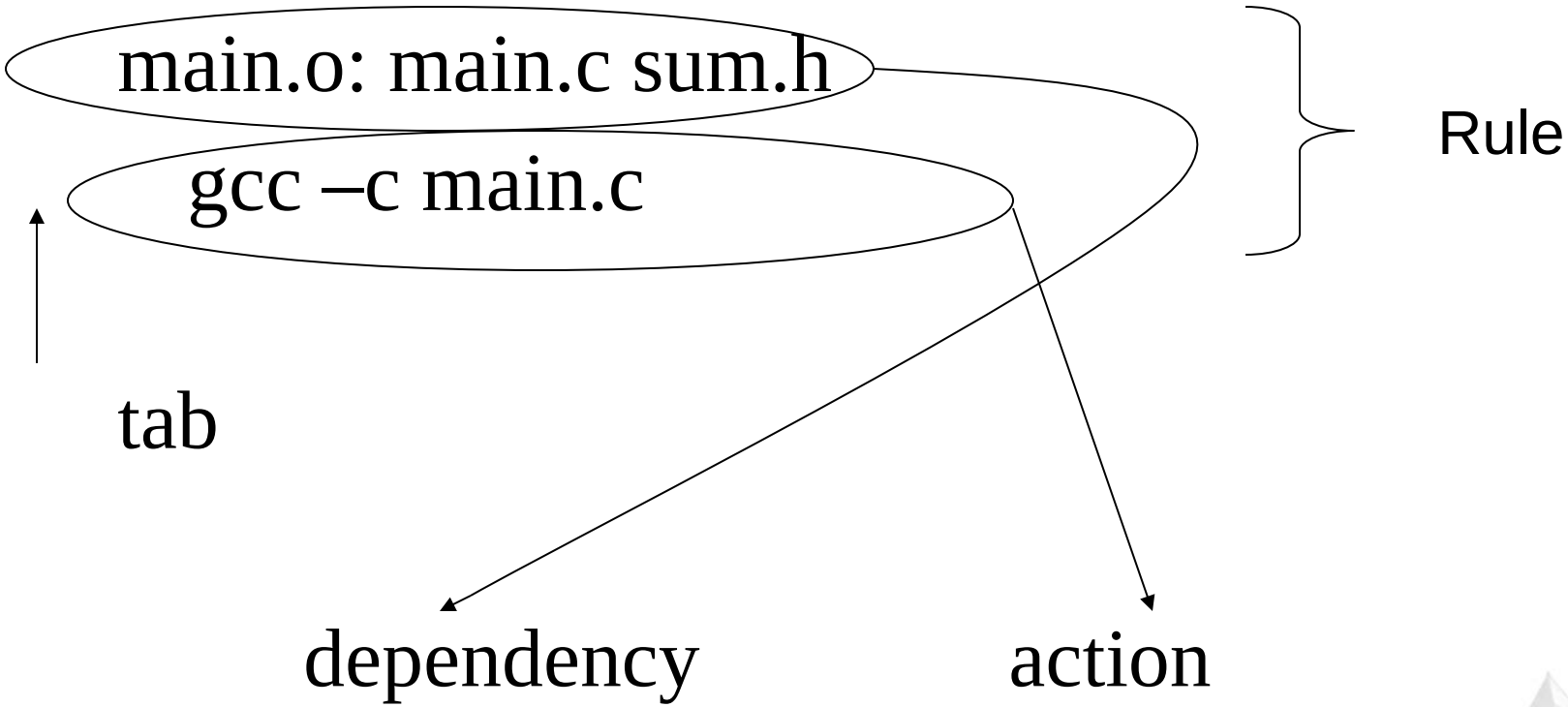
makefile

```
sum: main.o sum.o
    gcc –o sum main.o sum.o

main.o: main.c sum.h
    gcc –c main.c

sum.o: sum.c sum.h
    gcc –c sum.c
```

Rule syntax



Equivalent makefiles

- .o depends (by default) on corresponding .c file. Therefore, equivalent makefile is:

```
sum: main.o sum.o
    gcc –o sum main.o sum.o

main.o: sum.h
    gcc –c main.c

sum.o: sum.h
    gcc –c sum.c
```

Equivalent makefiles - continued

- We can compress identical dependencies and use built-in macros to get another (shorter) equivalent makefile :

```
sum: main.o sum.o
    gcc –o $@ main.o sum.o

main.o sum.o: sum.h
    gcc –c $*.c
```

$@ name of the target

$* name of each target without extension

make operation

- Project dependencies tree is constructed
- Target of first rule should be created
- We go down the tree to see if there is a target that should be recreated. This is the case when the target file is older than one of its dependencies
- In this case we recreate the target file according to the action specified, on our way up the tree. Consequently, more files may need to be recreated
- If something is changed, linking is usually necessary

make operation - continued

- make operation ensures minimum compilation, when the project structure is written properly
- Do not write something like:

  prog: main.c sum1.c sum2.c

  gcc –o prog main.c sum1.c sum2.c

  which requires compilation of all project when something is changed

Another makefile example

```
# Makefile to compare sorting routines

BASE = /home/blufox/base
CC          =   gcc
CFLAGS  =   -O –Wall
EFILE      =   $(BASE)/bin/compare_sorts
INCLS     =   -I$(LOC)/include
LIBS        =   $(LOC)/lib/g_lib.a \
                  $(LOC)/lib/h_lib.a
LOC        =   /usr/local

OBJS = main.o    another_qsort.o    chk_order.o \
           compare.o    quicksort.o

$(EFILE): $(OBJS)
       @echo “linking …”
       @$(CC) $(CFLAGS) –o $@ $(OBJS) $(LIBS)
# @command suppresses the echoing of the command

$(OBJS): compare_sorts.h
       $(CC) $(CFLAGS) $(INCLS) –c $*.c

# Clean intermediate files
clean:
       rm $(OBJS)
```

Example - continued

- We can define multiple targets in a makefile
- Target clean – has an empty set of dependencies. Used to clean intermediate files.
- make
  - Will create the compare_sorts executable
- make clean
  - Will remove intermediate files

Make: Advanced Options

- Pattern rules
  - Uses a pattern in the target with % as wildcard
  - Matched % can be used in dependencies as well
  - Simple Example:

```
%.o : %.cc
<tab>command …
```

- Pattern rules with automatic variables
  - $< first dependency
  - Advanced Example:

```
%.o : %.cc
<tab>$(CC) $(CCFLAGS) –c $< $(INCPATHS)
```

Make: A Simple Example

```
CC=g++                              # Compiler to use
FLAGS=-g                            # Compile flags
MASLAB_ROOT=maslab-software         # Maslab software root directory
LIB_DIR=$(MASLAB_ROOT)/liborc       # orc-related library directory
INC_DIR=$(MASLAB_ROOT)/liborc       # orc-related include directory
LIBS=-lm –lpthread -lorc            # Library files
                                        # note –l gcc option

all : helloworld

helloworld.o : helloworld.cc
    $(CC) $(FLAGS) –c $*.cc –o $@ -I$(INC_DIR)
# note –I gcc option


helloworld: helloworld.o
    $(CC) -o helloworld helloworld.o $(LIBS) –L$(LIB_DIR)
# note –L gcc option

clean:
    rm -f *.o helloworld
```

The real Problem

- How do we handle platform specific issues?
  - Providing a different Makefile for each architecture
  - Using Autoconf, Automake and Libtool
- The installer needs only
  - Bourne shell
  - C compilers
  - Make program

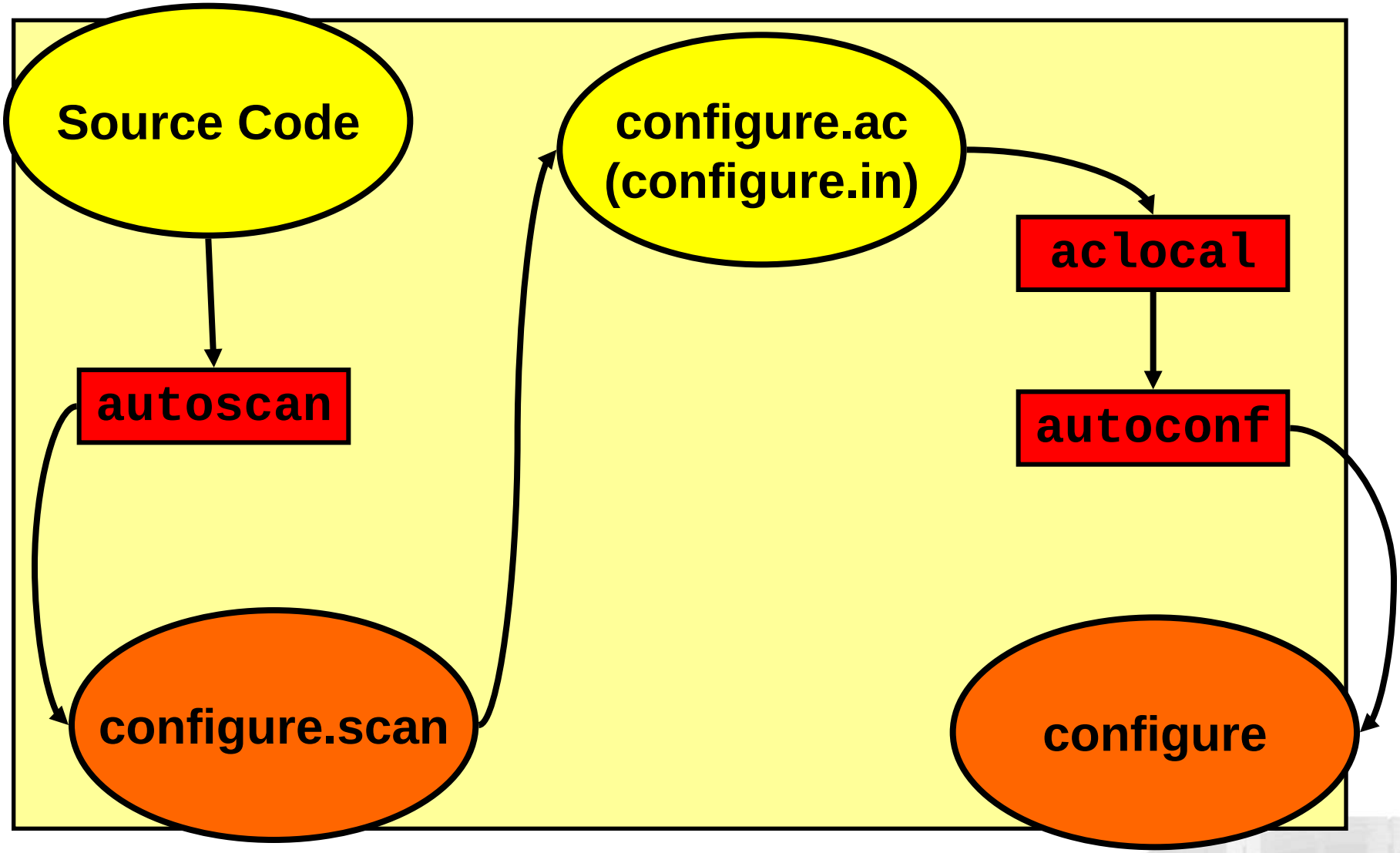
Some advantages when using GNU autotools

- The installation of a program is straightforward:

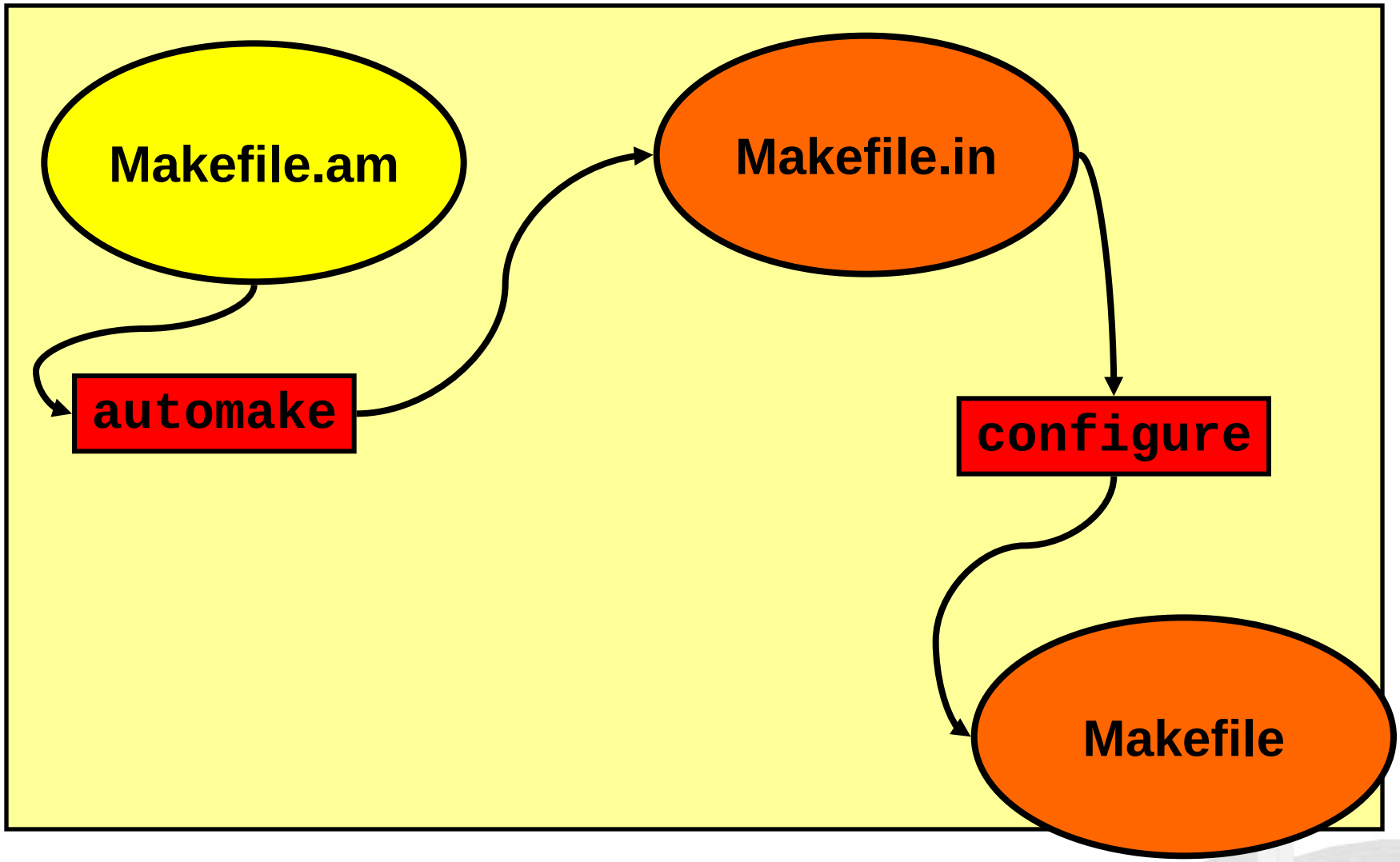
```
./configure; make; make install
```

- This procedure checks for system parameters, libraries, location of programs, availability of functions and writes a **Makefile**
- `./configure` supports many options to overwrite defaults settings
  - `/configure --prefix=…` (default `/usr/local`)

GNU autoconf



GNU automake



configure.ac

- dnl Comment … …
- AC_INIT(project_name, 1.2.8)
- AM_INIT_AUTOMAKE
- AC_PROG_CC — it locates the C (C++) compiler
- AC_HEADER_STDC — it checks for standard headers
- AC_CHECK_HEADERS(sys/time.h /header.h) — it checks for headers availability
- AC_CHECK_LIB(crypto SSLeay_version) — it checks for libraries availability
- AC_CHECK_FUNCS(ctime) — it checks for functions availability
- AC_PROG_INSTALL — it checks for BSD compatible install utility
- AC_OUTPUT

Makefile.am

- bin_PROGRAMS = foo
- foo_SOURCES=foo.c foo.h
- noist_PROGRAMS=test

(`make` compiles, `make install` does nothing)

- EXTRA_DIST=disclaimer.txt

Example

- foo.c :

```
#include <stdio.h>
main()
{
 printf(“Cum grano salis\n");
}
```

- Makefile.am :

```
bin_PROGRAMS = foo
foo_SOURCES = foo.c
```

- configure.ac :

```
AC_INIT(foo.c)
AM_INIT_AUTOMAKE(latin_words, 0.9)
AC_PROG_CC
AC_HEADER_STDC
AC_PROG_INSTALL
AC_OUTPUT([Makefile])
```

Summary

- Source Code, configure.ac, Makefile.am
- `autoscan; aclocal; autoconf`
- Create NEWS README AUTHORS ChangeLog
- `automake –add-missing`
- `./configure; make; make dist`
- Result: **project_name-2.10.tar.gz**

aclocal.m4 autom4te-2.53.cache ChangeLog config.status configure.in COPYING install-sh Makefile.am missing NEWS README AUTHORS autoscan.log config.log configure configure.scan INSTALL Makefile.in mkinstalldirs code.c

References

- GNU Autoconf, Automake, and Libtool
  http://sources.redhat.com/autobook/autobook/autobook_toc.html
- GNU Autoconf Manual
  http://www.gnu.org/manual/autoconf
- GNU Automake Manual
  http://www.gnu.org/manual/automake
- GNU Libtool Manual
  http://www.gnu.org/manual/libtool
- Learning the GNU development tools
  http://autotoolset.sourceforge.net/tutorial.html
- The GNU configure and build system
  http://www.airs.com/ian/configure/configure_toc.html
- GNU macro processor (GNU m4)
  http://www.gnu.org/manual/m4-1.4/m4.html