AutoTut
Using GNU auto{conf,make,header}

Felipe P. G. Bergo
bergo@seul.org

May 2001

1 Introduction

Most programmers make first contact with **autoconf** and friends when compiling and installing free software obtained from the web. These programmers are told to run a **configure** script, then type **make**, and usually **make install** as root to install the new software.

The widespread use of these **configure** scripts made building software from the sources easier. This document discusses the software packages used to generate the **configure** script and other associated files.

This document is aimed at developers intending to develop Free Software or already maintaining Free Software packages[1]

We assume the reader is already familiar with *Makefiles*.

2 Who's who

The software packages we are dealing directly with here are: **GNU autoconf**, **GNU automake** and **GNU autoheader**. Indirectly, we are dealing with the **m4** macro processor [6], **make** (either GNU[8] or BSD) and **aclocal**. We talk about **GNU libtool** too.

Figure 1 gives an overview of what the programs do. The main input files are **configure.in** and **Makefile.am**.

[1] An HTML version of this document with links to the example files is available at `http://www.seul.org/docs/autotut`

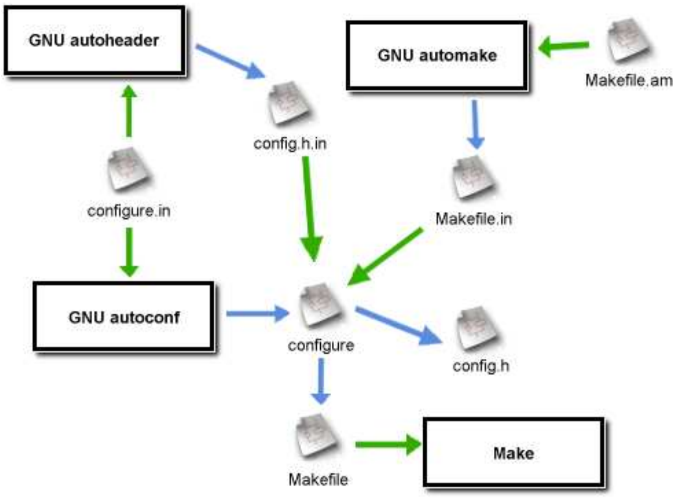

Figure 1: File Input/Ouput Graph

3 Meet Autoconf

In our first example[2] we'll ignore *automake* (by writing our own Makefile.in) and have a simple C program configure'd by **autoconf**.

■ Example 1: pi.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
        double value=0.0,denom=1.0,sig=1.0;
        unsigned long t,prec;

        printf("PI Approximator version %s\n",VERSION);

        prec=10;
        if (argc>1) prec=atol(argv[1]);

        for(t=prec;t;t--) {
                value+=sig/denom;
                sig=-sig;
                denom+=2.0;
        }
        value*=4.0;
```

[2]The full source code for all examples used in this text is available online at `http://www.seul.org/docs/autotut/examples`

```
        printf("pi ~= %.8f , with %lu iterations\n",value,prec);
        return 0;
}
```

This program will approximate the value of pi using a Taylor series.

■ Example 1: configure.in

```
AC_INIT(pi.c)

dnl find and test the C compiler
AC_PROG_CC
AC_LANG_C

AC_PROG_MAKE_SET

AC_HEADER_STDC
AC_CHECK_FUNC(atol,,AC_MSG_ERROR(oops! no atol ?!?))

VERSION="0.0.1"
AC_SUBST(VERSION)

dnl read Makefile.in and write Makefile
AC_OUTPUT(Makefile)
```

And finally the template for the Makefile (be careful, in Makefiles tabs and spaces are different. The 8-space indentations you see below must be tabs, not spaces):

■ Example 1: Makefile.in

```
CC = @CC@
VERSION = @VERSION@
CFLAGS = @CFLAGS@

all: pi-bin

pi-bin: pi.c
        $(CC) $(CFLAGS) -DVERSION=\"$(VERSION)\" pi.c -o pi

clean:
        rm -f pi

distclean:
        rm -f pi config.* Makefile
```

Having these 3 files in a directory, run **autoconf**:

```
autoconf
```

This will generate a file called **configure**, which happens to be a shell script. Now run

```
./configure
```

You should see an output like this

■ **Example 1: configure output**

```
creating cache ./config.cache
checking for gcc... gcc
checking whether the C compiler (gcc  ) works... yes
checking whether the C compiler (gcc  ) is a cross-compiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking whether make sets ${MAKE}... yes
checking how to run the C preprocessor... gcc -E
checking for ANSI C header files... yes
checking for atol... yes
updating cache ./config.cache
creating ./config.status
creating Makefile
```

As seen in the output, a file called **Makefile** was indeed created. It closely resembles the **Makefile.in** file. In fact, the only changes `configure` does in the translation from Makefile.in to Makefile is substituting variable values between at-signs (like @CC@, @VERSION@). And it does this kind of translation to every file listed in the AC_OUTPUT macro. We'll get to the macros soon.

Now you can run `make`. It will compile the program and generate a binary called `pi`. You might run it this way, for example:

```
galadriel:~/autotut-ex1$ ./pi 2000000
PI Approximator version 0.0.1
pi ~= 3.14159215 , with 2000000 iterations
galadriel:~/autotut-ex1$
```

4 Autoconf Macros

Now that the basic **autoconf** functionality has been presented, we discuss the format of the **configure.in** file.

The configure.in file is processed by the **m4** macro processor, but aside from the m4 macros, it is a Bourne shell script [2]. Thus you can use **if...then...fi**, and constructs from Bourne shells. You can have some reference on shell programming by reading the **bash** or **sh** man pages (depending on the operating system you are on).

■ **IMPORTANT!** ■ The square brackets have special meaning to the m4 processor, so if you want to use the **test** (see **test (1)**) command you must spell it out as `test` instead of using the Bourne shell [...] abbreviation.

Also, the shell variables defined and modified within configure.in are NOT substituted automatically in the AC_OUTPUT files. You must explicitly call AC_SUBST (as done with the VERSION variable in the first example) to request the substitution. However, when an internal autoconf macro says "sets variable NAME", it sets and AC_SUBSTs it, thus you must only worry about variables set on your own.

m4 macros are called with the following syntax: (**dnl** is the comment directive, it is equivalent to C++ // comments: may start at any position in the line, and the rest of the line is considered a comment)

```
FOO(parameter1, parameter2, parameter3)

BAR dnl a macro without parameters

ZEN(parameter1,,parameter3) dnl optional parameter 2 omitted

TUT(parameter1,parameter2,[
this is parameter 3 which spans
along several lines. Now you
know why square brackets cannot
be used for the 'test' command.])
```

Conventionally , all macros provided by autoconf start with **AC_**. As we shall see later, automake's macros start with **AM_**.

Autoconf sets some variables by itself, the most important of them is **prefix** (referenced as `@prefix@` in a Makefile.in), which is the installation prefix chosen with *configure –prefix=path*. It defaults to /usr/local. This means binaries go in /usr/local/bin, libraries in /usr/local/lib, man pages in /usr/local/man and so on.

What we are most interested in are the macros provided by **autoconf**. This is not at all a complete list, just the most common macros.

Macro Prototype	**Comments**
AC_INIT(sourcefile)	Initializes autoconf, should be the first macro called in configure.in. sourcefile is the name (relative to current directory) of a source file from your code.
AC_PROG_CC	Determines a C compiler to use, sets the CC variable. If this is GCC, set the GCC variable to 'yes', otherwise 'no'. Initializes the CFLAGS variable if it hasn't been set already (to override CFLAGS, do it in configure.in BEFORE calling this macro)

AC_PROG_CXX	Determines a C++ compiler to use, sets the CXX variable. If this is the GNU C++ compiler, set GXX to 'yes', otherwise 'no'. Initializes the CXXFLAGS variable if it hasn't been set already (to override CXXFLAGS, do it in configure.in BEFORE calling this macro)
AC_LANG_C	Tests the C compiler
AC_LANG_CPLUSPLUS	Tests the C++ compiler
AC_PROG_INSTALL	Set variable INSTALL to the path of a BSD-compatible install program (see **install (1)**). If not found, set it to '*dir*/install-sh -c' , looking in the directories specified to AC_CONFIG_AUX_DIR. Also sets INSTALL_SCRIPT and INSTALL_PROGRAM to $(INSTALL), and INSTALL_DATA to '$(INSTALL) -m 644'. **You must provide a install-sh file in the current directory** (unless you use AC_CONFIG_AUX_DIR – the common practice is to provide a install-sh file) **else autoconf will refuse to run**.
AC_PATH_X	Try to locate the X window system's includes and libraries, and sets the variables x_includes and x_libraries to their locations.
AC_PATH_XTRA	Like the previous, but adds the required include flags to X_CFLAGS and required linking flags to X_LIBS.
AC_PATH_PROG(a,b[,c[,d]]) a=variable-name b=prog-to-check c=value-if-not-found d=path	Looks for prog-to-check in PATH, and sets variable-name to the full path if found, or to value-if-not-found if not.
AC_PROG_MAKE_SET	If make predefines the variable MAKE, define output variable SET_MAKE to be empty. Otherwise, define SET_MAKE to contain 'MAKE=make'.
AC_OUTPUT(files [,a[,b]])	Create output files. Perform substitutions on *files*, which contains a list of files separated by spaces. (is writes, say, Makefile, from a Makefile.in file, spec from spec.in, and so on. The name given here is *without* the .in suffix. The other 2 parameters are seldom used, consult the autoconf docs if needed. If AC_CONFIG_HEADER, AC_LINK_FILES or AC_CONFIG_SUBDIRS were called, the files named as their arguments are created too.
AC_CONFIG_HEADER(files)	Make AC_OUTPUT create the headers listed in the files list (space-separated). Replaces @DEFS@ in generated files with -DHAVE_CONFIG_H. The usual name for the header is config.h (created from config.h.in. The **autoheader** generates config.h.in files automatically for you (it is documented in the next sections).
AC_CONFIG_SUBDIRS(dirs)	run **configure** scripts in the subdirectories listed in *dirs* (space-separated). This is meant for when you nest child packages to your program (like including libraries as subdirs).
AC_CHECK_FUNC(a[,b[,c]]) a=function b=action if found c=action if not found	checks if the given C function is available in the standard library (i.e., the libraries that are linked by default to any C program).

AC_CHECK_FUNCS(a[,b[,c]]) a=list of functions (space-separated) b=action if found c=action if not found	similar to AC_CHECK_FUNC, but looks for many functions at once, setting HAVE_function for each function found (in the given set).
AC_CHECK_LIB(a,b[,c[,d[,e]]]) a=library name b=function name c=action if found d=action if not found e=see autoconf docs	Checks whether a function exists in the given library (library names without the leading lib, e.g., for libxml, use just xml here)
AC_HEADER_STDC	Checks for stdlib.h, stdarg.h , string.h and float.h, defines STDC_HEADERS on success.
AC_CHECK_HEADER(header[,a[,b]]) a=action if found b=action if not found	Checks whether a given header file exists.
AC_MSG_ERROR(message)	Notifies the user an error has occurred and exits configure with a non-zero status. This is what you should do when a required library or header is missing.
AC_MSG_WARNING(message)	Notifies the user with the given message. This is what you should do when an optional library is missing (thus the final result will be not as good as it could)
AC_ARG_ENABLE(feature,help[,a[,b]]) a=action if given b=action if not given	Checks whether the user gave –enable-*feature* in the configure command-line. The help string is shown in **configure –help**

The full documentation for these is found in [4].

There are many more autoconf macros. If these weren't already enough, you can supply a macro library with your own software distribution. To do this, place the macro definitions in a file called **aclocal.m4** in the same directory as configure.in, all macros defined in aclocal can now be used in configure.in. The common use for this is to have specific-library checking macros, e.g. AC_REQUIRE_GTK(1,2,8) to require the GTK library version 1.2.8 or later. You may also place your macros in a file called **acinclude.m4** if you are using the **aclocal** program. **aclocal** will be presented later.

And don't be tempted to have configure.in for each directory in your source tree. Multiple configure.in's are only desirable when you integrate different software packages in a common source tree (such as *my program and the 3000 libraries it requires*).

If you really need nested configure.in's you should read autoconf's documentation [4].

5 Autoheader

Often you will want to make your code portable and want to use values gathered in the *configure* step to select which code to compile (with #ifdefs in C code). However the autoconf macros that generate definitions can pile up quite quickly, and you may end up with gcc lines like:

```
gcc -g -O2 -Wall -fexpensive-optimizations -DSTDC_HEADERS=1
```

```
-DHAVE___clone=0 -DHAVE_localtime=1 -DVERSION=\"0.0.1beta-fb-rc4\"
-DSYSTEM_STRING=\"FreeBSD-5.0-CURRENT-i386\" -DSYSTEM=\"FreeBSD\"
-DPERL_VERSION_MAJOR=5 -DPERL_VERSION_MINOR=6
-DPERL_VERSION_MICRO=0 -I. -I.. -I/usr/X11R6/include
-I/opt/include -c gibberish.c -o gibberish.o
```

While it is no harm to your system, seeing these mutant odd-balls scroll as your program is compiled can be quite unpleasant. (as a side note, FreeBSD was chosen as an example in this document about GNU tools as the system name/version strings it provides are usually much longer and more polluted than those in GNU/Linux systems).

The solution is to put all the definitions in a header file. You can do that in autoconf with the AC_CONFIG_HEADER macro. The conventional name for this header is config.h. However, you must supply a .in header (config.h.in) with all the possible definitions spelled out, so that autoconf will just change #define to #undef or add a value to the #define. Here comes **autoheader**. Just run **autoheader** on a directory with a configure.in that contains a AC_CONFIG_HEADER macro call, and it will write the .in file.

Let's use the same pi.c program we did before to show the work of autoheader. I won't list all files again, just configure.in.

■ **Example 2: configure.in**

```
AC_INIT(pi.c)
AC_CONFIG_HEADER(config.h)

dnl find and test the C compiler
AC_PROG_CC
AC_LANG_C

AC_PROG_MAKE_SET

AC_HEADER_STDC
AC_CHECK_FUNCS(atol atoi strtod)

VERSION="0.0.1-rc2"
AC_SUBST(VERSION)

dnl read Makefile.in and config.h.in, write Makefile and config.h
AC_OUTPUT(Makefile)
```

We're not really using atoi() and strtod(), they're there just to make some volume.

After running **autoheader** we get this **config.h.in**:

■ **Example 2: config.h.in**

```
/* config.h.in.  Generated automatically from configure.in by autoheader.  */

/* Define if you have the ANSI C header files.  */
#undef STDC_HEADERS

/* Define if you have the atoi function.  */
#undef HAVE_ATOI

/* Define if you have the atol function.  */
#undef HAVE_ATOL

/* Define if you have the strtod function.  */
#undef HAVE_STRTOD
```

And now you run configure as usual. It will write a config.h based on this template. Let's see what it generates:

■ **Example 2: config.h**

```
/* config.h.  Generated automatically by configure.  */
/* config.h.in.  Generated automatically from configure.in by autoheader.  */

/* Define if you have the ANSI C header files.  */
#define STDC_HEADERS 1

/* Define if you have the atoi function.  */
#define HAVE_ATOI 1

/* Define if you have the atol function.  */
#define HAVE_ATOL 1

/* Define if you have the strtod function.  */
#define HAVE_STRTOD 1
```

Of course you must #include "config.h" in your source files to use the definitions.

Also, autoheader only copies definitions from the /usr/share/autoconf/acconfig.h file. If some macro you used needs a different entry in config.h.in, write it in a file called **acconfig.h** in the same directory as configure.in, and it will be included in config.h.in by autoheader.

6 Automake

You may have noticed that our example Makefile.in does not have a "install" target that installs the program to its permanent location on the system. Neither does it have a "uninstall" target. It also lacks other lesser-known, standard targets [7] [5], such as "dist". The tarball doesn't have a README file, nor an INSTALL file. Where's the license ?

Dozens of new versions of Free Software packages are released every day. The average user expects the task of getting a new software package (or a new release of a software package he already uses) to be: fast, easy, simple and uneventful.

The software archive should be standard, the user should already know how to unpack it; The building process should not require any programming skills from the user – did you know there are people out there who can't understand Perl ? – requiring the user to edit an include file that'll be parsed by a strict-syntax interpreter/compiler should be avoided at all costs; The building process should not take long. If it really must take long, all user-interaction should be concentrated in one single, as fast as possible, step. The simple user – not concerned with programming languages, compilers, etc. – should receive error messages in a user-friendly manner, and as soon as possible. This means

```
checking for main in -lstdc++... no
configure: error: libstdc++ missing. This software
requires libstdc++, please come back after installing it.
```

is much better than, after compiling 10 files, the user be brought to:

```
oops.cc: In function ‘int main(int, char **)’:
oops.cc:7: ‘list’ undeclared (first use this function)
oops.cc:7: (Each undeclared identifier is reported only once
oops.cc:7: for each function it appears in.)
oops.cc:7: parse error before ‘>’
oops.cc:10: ‘c’ undeclared (first use this function)
oops.cc:11: confused by earlier errors, bailing out
```

The former solution at least gave the user something to write in the Google or Freshmeat search boxes.

The build process should be consistent for many reasons. One is philosophical: when you make an end-user feel skilled, by teaching him how to build software from source, he may try other softwares distributed the same way (probably he will). He comes closer to the Free Software community. He notices he accomplished very much with little effort – he just had to type 3 commands the INSTALL file told him to type – and is looking forward to learn more. Another reason is practical: you want your users to use your software, you may want that other programmers check your software for bugs (it will happen, whether you want it or not ;-), but you don't want to have your mailbox filled with requests like 'Cannot compile foobar, please help me'. Worse, many people will just turn their back on your software if they fail to build it.

Some software packages, like the Linux kernel, for instance, require such a burden of options (which driver comes in, which comes not, what CPU you have, whether your network addresses are 4- or 16-byte long, or both, or neither, and the list goes on), that using autoconf to configure it would be a pain. But this is a quite special piece of software, it should be configured in its own way.

Most Free Software packages available have very few, if any, fancy requirements, and it is an annoyance for the user if s/he has to read a long document to discover how to compile and install (sometimes even finding the document is a challenge). This even leads to users preferring binary packages that come without source code (heresy!). It is a service to all Free Software users to ensure that your software configures/compiles/installs in a consistent way. The de facto standard is to use automake+autoconf. You may try to emulate the behavior of an automake-generated Makefile by hand, but there are many pitfalls you can be trapped in.

In the next sections we present **GNU automake**, which generates a **Makefile.in** file based on a **Makefile.am** file. Automake will also look for missing files that may cause your users to be confused if they are not present. Automake will either place a standard one in the package directory or complain about the missing file. (The rule is that universal files are copied from automake's distribution, and package-dependent files like AUTHORS, ChangeLog, etc. aren't).

Note: automake is not the only tool to help your users. See the bibliography for other documents discussing software packaging and distribution.

7 Constructing an Automake-aware Software Package

The Makefile.am file basically contains Makefile variable definitions, included in a templated Makefile built by automake. So the syntax in this file is the syntax of a Makefile: comments are placed in lines starting with '#', multi-line lists are made with backslashes.

In this section we present an example, in the next we explain the format of the Makefile.am file in detail.

The example for this section is a GTK+ front-end for the **du** utility, called **gdu**. First we show, step by step, what the source files are and what files must be created (as input for automake and autoconf), and how automake, autoconf and other utilities are run to generate a working package.

1. Create an empty directory for the project

   ```
   mkdir gdu
   ```

2. Place source files

   For gdu, the source is consists of 2 C++ source files (.cc), 1 C++ header (.h), 6 pixmap pictures for the toolbar (.xpm) and a man page for the program (gdu.1x).

```
gdu.cc        gdu.h       tree.cc
about.xpm     back.xpm    home.xpm
reload.xpm    root.xpm    stop.xpm
gdu.1x
```

Of course, in a real-world situation, you won't copy these files, you'll write them :-).

3. Write configure.in

   Write a **configure.in** file for autoconf. Notice that some automake macros (AM_) come in, and some autoconf macros are replaced by automake macros.

   You must call `AM_INIT_AUTOMAKE(package,version)` right after `AC_INIT`, use `AM_CONFIG_HEADER` instead of `AC_CONFIG_HEADER`.

   This is our configure.in:

   ■ **Example 3 (GDU): configure.in**

```
AC_INIT(gdu.cc)
AM_INIT_AUTOMAKE(gdu,0.0.1)
AM_CONFIG_HEADER(config.h)

CXXFLAGS=

AC_PROG_CC dnl for AM_PROG_GTK only
AC_PROG_CXX
AC_PROG_INSTALL
AC_PROG_MAKE_SET

AC_HEADER_STDC

dnl it will define GTK_CFLAGS and GTK_LIBS for us,
dnl comes from gtk.m4 (aclocal will find it for us)
AM_PATH_GTK(1.2.6)

AC_LANG_CPLUSPLUS dnl else AM_PATH_GTK blows up

AC_CHECK_LIB(stdc++, main,,AC_MSG_ERROR(gdu requires libstdc++))
AC_CHECK_HEADERS(stack,,AC_MSG_WARN(STL classes missing ?))
AC_CHECK_HEADERS(string,,AC_MSG_WARN(STL classes missing ?))
AC_CHECK_HEADERS(list,,AC_MSG_WARN(STL classes missing ?))
AC_CHECK_HEADERS(vector,,AC_MSG_WARN(STL classes missing ?))

AC_OUTPUT(Makefile)
```

   If you use macros from aclocal files (usually in /usr/share/aclocal), as we did here (the AM_PROG_GTK macro is in /usr/share/aclocal/gtk.m4, and some automake macros must be imported too), you must run the **aclocal** program to create an

aclocal.m4 file with all needed macros. This file is overwritten every time aclocal is run, so if you want to have your own macros written locally, place them in **acinclude.m4**, aclocal will include these in the generated **aclocal.m4**.

4. Run aclocal, autoconf, autoheader

   Now you can already generate some of the files. You must run aclocal, autoheader and autoconf; aclocal must run first. In a Bourne shell you can run all three in a row with

   ```
   aclocal && autoheader && autoconf
   ```

   aclocal will generate an **aclocal.m4** file (getting information from configure.in [needed] and acinclude.m4 [optional]).

   autoheader will read configure.in and produce a **config.h.in** file (or whatever you named it in the AM_CONFIG_HEADER macro).

   autoconf will read configure.in and aclocal.m4 and produce a **configure** script.

5. Write a Makefile.am

   This is our Makefile.am:

   ■ **Example 3 (GDU): Makefile.am**

   ```
   AUTOMAKE_OPTIONS = gnu
   LDADD =    @GTK_LIBS@
   CPPFLAGS = @GTK_CFLAGS@
   bin_PROGRAMS = gdu
   gdu_SOURCES = gdu.cc tree.cc

   noinst_HEADERS = gdu.h

   # man page
   man_MANS = gdu.1x

   # we want these in the dist tarball
   EXTRA_DIST = back.xpm reload.xpm root.xpm \
                home.xpm stop.xpm about.xpm gdu.1x
   ```

   We'll dissect nomenclature later, but:

   AUTOMAKE_OPTIONS = gnu could be achieved by running automake with the –gnu command-line option. See 'automake –help' for all command-line options.

   LDADD and CPPFLAGS (CFLAGS for C programs) specify flags used for linking and compiling);

   bin_PROGRAMS is a (space-separated) list of all binaries that will be built and installed by your package. We'll build just **gdu** here.

gdu_SOURCES is the list of source files that are compiled and linked to generate **gdu**. For each item in the bin_PROGRAMS list you need a name_SOURCES list of sources.

man_MANS lists the man pages installed.

noinst_HEADERS is the list of headers that must go into a distribution (a distribution is the .tar.gz file you distribute you program) but are NOT installed to `@prefix@/include` in the **install** step.

EXTRA_DIST lists additional files that must be included in the distribution. The xpm files could be listed under noinst_HEADERS, it really makes no difference here.

6. Running automake

   Now we are ready to run automake (maybe). There some files missing, but automake will tell us about them, so relax. Run this (the –add-missing will add default files from the automake distribution when possible, complain about missing files when not)

```
automake --add-missing
```

   we obtain

   ■ **Example 3 (GDU): automake output**

```
automake: configure.in: installing `./install-sh'
automake: configure.in: installing `./mkinstalldirs'
automake: configure.in: installing `./missing'
automake: Makefile.am: installing `./INSTALL'
automake: Makefile.am: required file `./NEWS' not found
automake: Makefile.am: required file `./README' not found
automake: Makefile.am: installing `./COPYING'
automake: Makefile.am: required file `./AUTHORS' not found
automake: Makefile.am: required file `./ChangeLog' not found
```

   install-sh is the shell replacement for the BSD install program, triggered by AC_PROG_INSTALL. mkinstalldirs and missing are similar shell scripts triggered by the AM_ macros in configure.in.

   INSTALL is the automake standard installation instructions text. COPYING is the GNU General Public License. COPYING was "needed" because we specified AUTOMAKE_OPTIONS = gnu. To work with bare requirements use AUTOMAKE_OPTIONS = foreign. In our example we stick with **gnu**, this is GPL'd program anyway.

   However, the other required files couldn't be resolved. Because they are package-specific. To keep the **gnu** option, we have to write each of those files.

   5 minutes later...

Having written the missing files, we run **automake** again (no –add-missing required, all it could add has already been added), and it runs swiftly without errors.

You'll notice that the added files are symlinks to files in /usr/share/automake. This is fine for some people, since the **make dist** step that creates the distribution tarball packs as files, but if you will import you source tree into CVS (Concurrent Versioning System – not needed but widely used, don't worry if you don't use it) you probably want to run automake with **automake –add-missing –copy** to force file copying (if you want to do it now, you'll have to remove the symlinks first).

7. Phew! Test it.

   You can now test your package with:

   ```
   ./configure
   make
   make dist
   make clean
   make distclean
   ```

   The first one will compile the package. The **dist** target will create a distribution tarball – the one you'll distribute to your users – in our case it will create a tarball named **gdu-0.0.1.tar.gz**, name and version are taken from the AM_INIT_AUTOMAKE macro in configure.in. This tarball complies with what users expect: it unpacks into a fresh directory of the same name (without the .tar.gz suffix).

■ **IMPORTANT!** ■ You should always test the distribution tarball (by unpacking, compiling and installing), for it is usual to have files left out because you forgot to add graphic files, program headers or other needed files in the EXTRA_DIST or noinst_HEADERS variables in Makefile.am. Some files (README, INSTALL, etc.) are automatically included in the distribution if found, and running automake with 'automake –help' will show a list of all included-if-found filenames.

The **clean** target removes binaries and object files (.o).

The **distclean** target removes not only the binaries, but all the configuration results created by the configure script. If you think your changes to configure.in or Makefile.am are not taking effect after running autoconf and automake, do a **make distclean** to clean any cached results. Also, you should do **make distclean** before importing your source tree into a CVS repository.

Other targets are install, uninstall and test.

8 Makefile.am

SUBDIRS. Automake'd projects can vary in directory topology. **Flat** topologies include all source files in the project's top directory, like **gdu**. **Shallow** topologies includes the sources in some subdirectories. The most common situation is having the source go inside a **src** directory instead of the top directory. **Deep** topologies is what happens when you include your source in the top directory and libraries in subdirectories.

To have a multi-directory source tree, specify all child directories in the SUBDIRS variable. Example:

```
SUBDIRS = src library1 library2
```

Each subdirectory must have its own Makefile.am.

PROGRAMS. The `*_PROGRAMS` variables specify programs to be built. bin_PROGRAMS list programs to be built and installed to `@prefix@/bin`, sbin_PROGRAMS are built and installed to `@prefix@/sbin`; EXTRA_PROGRAMS are built but not installed.

Where Files Go ? For each *program* listed, program_SOURCES lists the source files. include_HEADERS lists headers that are installed to `@prefix@/include`. EXTRA_DIST and noinst_HEADERS files are included in the distribution but not installed.

To install files to `@prefix@/share/program`, list the files in pkgdata_DATA.

Man pages can be listed in man_MANS. Texinfo documentation can be listed in info_TEXINFOS, e.g.: info_TEXINFOS = hello.texi. See the automake documentation [3] if your texinfo documentation has dependencies.

Scripts that must not be compiled can be listed in bin_SCRIPTS (to install somewhere else, change bin to whatever you want appended to @prefix@ – e.g.: sbin_SCRIPTS).

This is what most programs will need from automake. The automake documentation [3] describes more formally what is installed, what goes in a distribution, more variable names, etc.

9 Building Libraries: libtool

In this section we show a very short example of a shared library package. Automake will write a Makefile.in file that uses libtool to build it in a portable way. Libtool is **the** way to build libraries. It will prevent you from dealing with arcane system idiosyncrasies by selecting compiler and linker flags and running system utilities as needed to integrate libraries seamlessly into the system.

The example for this section is a set of 4 programs, one of which is a shared library. We'll use this same set of programs as an example for the next section.

The four programs are: **dinnerd**, **libdinner**, **hungryhacker** and **vdinner**.

dinnerd is a daemon that acts as a server for the Dining Hackers protocol (this is an unofficial protocol, mostly useless, that I created for this example – for a deeper explanation of protocol check the documentation in the dinnerd package). It manages a round table with at most 256 places, and waits for hackers (clients) to sit around. There are always N Chinese food boxes and N chop-sticks on the table, where N is the current number of hackers. Hackers need 2 chop-sticks to eat, so they all cannot eat at the same time. Competition for chop-sticks occurs, and hackers get angry and may beat each other if they can't get a chop-stick, and even die of starvation if they can't eat for too long. dinnerd provides two sub-protocols: one for hackers to connect and interact (request chop-sticks, etc.) and another for observers to watch the whole scene.

libdinner is a library that provides a set of functions to access a *dinnerd* server without the need to code directly with the Unix network API – you don't need to know what bytes come and go in the connection, the library wraps the ugly things for you.

hungryhacker implements a client for the first sub-protocol. It implements a hacker that sleeps (in the sense of the Unix sleep() call, not human sleep) for a random amount of time, try to acquire chop-sticks, eat for a random amount of time, repeat.

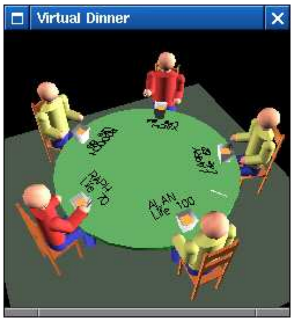

Figure 2: vdinner screen-shot

vdinner implements a client for the second sub-protocol, it grabbing information about the status of the table from *dinnerd* using the library functions of *libdinner* and renders everything animated in 3D with the aid of the Mesa graphics library.

In this section we'll focus on *libdinner*, which uses **libtool** to build portably across operating systems.

The library source code is contained on two files: libdinner.c and dinner.h. The later will be installed under @prefix@/include and applications that link with the library will need to include it.

The first step to build a library is including the **AM_PROG_LIBTOOL** macro in `configure.in`. This is *libdinner*'s `configure.in` file:

■ **Example 4 (libdinner): configure.in**

```
AC_INIT(libdinner.c)
AM_INIT_AUTOMAKE(libdinner,0.0.1)
AM_PROG_LIBTOOL

AC_PROG_INSTALL

AC_LANG_C
AC_PROG_CC
AC_PROG_MAKE_SET

AC_HEADER_STDC

AC_CHECK_HEADERS(unistd.h netdb.h netinet/in.h sys/types.h sys/socket.h,,AC_MSG_ERROR[
required header file missing])

AC_CHECK_FUNCS(gethostbyname socket htons connect shutdown,,AC_MSG_ERROR([
required standard library function missing]))

AC_OUTPUT(Makefile)
```

And this is *libdinner*'s `Makefile.am` file:

■ Example 4 (libdinner): Makefile.am

```
AUTOMAKE_OPTIONS = gnu
lib_LTLIBRARIES = libdinner.la

libdinner_la_SOURCES = libdinner.c

include_HEADERS = dinner.h

libdinner_la_LDFLAGS = -version-info 0:0:0
```

As usual, automake variable names have two parts. The *lib* in lib_LTLIBRARIES means the target will be installed to @prefix@/lib. And the LT in LTLIBRARIES is from **L**ib**T**ool. When target names have dashes (-) or dots (.), they're substituted for underscores, so we have `libdinner_la_SOURCES` instead of `libdinner.la_SOURCES`.

We're using for the first time the include_HEADERS variable. It will place `dinner.h` in @prefix@/include.

The last line deals with shared library versioning, which is NOT what you usually call a version number. Most software, both free and non-free, have a versioning of their own. Developers choose almost randomly when to increase major, minor and micro version numbers. With shared libraries it can't be done by guessing, since the changes in a library may break current binary code that was linked against an older version. So when you see a package called libdinner-0.0.1.tar.gz, 0.0.1 is the author designated version number. I could have called it `libdinner-2.A.Omega-Roglaur`. But the libtool version number for this package is 0:0:0.

Libtool calls each of those 3 numbers: CURRENT, REVISION, AGE.

Current is the version of the interface the library implements. Whenever a new function is added, or its name changed, or the number or type of its parameters (the prototype – in libraries we call this the **function signature**), this number goes up. And it goes up exactly by one.

Revision is the revision of the implementation of this interface, i.e., when you change the library by only modifying code inside the functions (fixing bugs, optimizing internal behavior, or adding/removing/changing signatures of functions that are private to the library – used only internally) you raise the revision number only.

Age is the difference between the newest and oldest interface the library currently implements. Let's say you had 8 versions of your library's interface, 0 through 7. You are now on the 4th revision of the 8th interface, that is, 7:3:X (remember we start counting on zero). And when you had to make choices for what old interfaces you would keep support – for backward compatibility purposes, you chose to keep support for interfaces 5, 6 and (obviously) the current, 7. The libtool version of your library would be 7:3:2 , because the Age is 7-5 = 2.

From the libtool documentation [1], verbatim:

```
   **Never** try to set the interface numbers so that they correspond
to the release number of your package.  This is an abuse that only
fosters misunderstanding of the purpose of library versions.  Instead,
use the `-release' flag (*note Release numbers::.), but be warned that
every release of your package will not be binary compatible with any
other release.
```

You have been warned. If you are planning to write a library that many people may want to use, you should read libtool's documentation [1], it'll give you good advice. You probably already are able to read it in your system with the command `info libtool`. How to navigate in the GNU info system is left as an exercise to the reader.

The last advice about libtool is about the need to run the `ldconfig` command on some systems after installing a library: even though the linker (ld) is able to find the new library, the dynamic linker – the one responsible for loading the library when a program that uses it comes up – may not have updated its cache yet. Again, the command to do it `ldconfig`.

10 Nesting

Until now all our packages were **flat**, with all source contained in a single directory. Now we examine the other two kinds of packages: **Shallow** and **Deep**.

The major difference between shallow and deep is that in *shallow* packages you have only one *configure.in*, located in the top directory, and it will check everything that everything in the subdirectories need.

Deep packages have a configure.in in each subdirectory, and it looks very much like an aggregate of programs which just happen to be distributed together.

10.1 Shallow Example

Our shallow example is **vdinner**

It has three important requirements: X, Mesa and libdinner. All the code is contained in the `src` subdirectory. For shallow topologies, we write a top-level *configure.in* just like in a *flat* topology, except that we must remember to AC_OUTPUT Makefiles for all subdirectories and references to files may need directory prefixes, such as `src/vdinner.c`. This is vdinner's *configure.in*:

■ Example 5 (vdinner): configure.in

```
AC_INIT(src/vdinner.c)
AM_INIT_AUTOMAKE(vdinner,0.0.1)
AM_CONFIG_HEADER(config.h)

AC_PROG_INSTALL

CFLAGS=

AC_PROG_CC
AC_LANG_C
AC_PROG_MAKE_SET

AC_HEADER_STDC

AC_CHECK_HEADERS(ctype.h sys/time.h unistd.h math.h,,AC_MSG_ERROR([
required header files missing]))

AC_PATH_XTRA

dnl check for Mesa

AC_CHECK_HEADERS(GL/glut.h,,AC_MSG_ERROR([
Mesa header not found. The Mesa graphics
library is required to compile and run
vdinner.
Check http://mesa3d.sourceforge.net]))

CFLAGS="$CFLAGS $X_CFLAGS"
LIBS="$X_PRE_LIBS $X_LIBS -lX11 -lXext -lXmu -lXt $X_EXTRA_LIBS -lGL -lGLU -lglut -lm"

AC_MSG_CHECKING(for the Mesa graphics library)
AC_TRY_LINK([
#include <GL/glut.h>
],[
glutInitWindowSize(400,400);
glutCreateWindow("test");
glEnable(GL_DEPTH_TEST);
glShadeModel(GL_FLAT);
],AC_MSG_RESULT(yes),AC_MSG_ERROR([
```

```
Unable to link to Mesa library. If you
just installed it: try running ldconfig
as root]))

dnl libdinner (our library)
AC_CHECK_HEADERS(dinner.h,,AC_MSG_ERROR([
dinner.h not found; maybe libdinner was not
installed ? The hungryhacker package uses the
libdinner library. Install libdinner first.]))

AC_CHECK_LIB(dinner,dinner_open,,AC_MSG_ERROR([
could not link to libdinner. The hungryhacker
package uses the libdinner library. Install
libdinner first.]))

AC_OUTPUT(Makefile src/Makefile)
```

The only difference from previous examples is src/vdinner.c in AC_INIT and src/Makefile in AC_OUTPUT (it does use some macros we haven't used before, like AC_PATH_XTRA and AC_TRY_LINK, but due to this package' idiosyncrasies, not for its topology).

We need a *Makefile.am* file in every directory. This is the top-level directory *Makefile.am*:

■ Example 5 (vdinner): Makefile.am

```
AUTOMAKE_OPTIONS = gnu
SUBDIRS = src
```

Quite straightforward. The directories should be listed in SUBDIRS in the order you wish them to be built. This is important when one subdirectory depends on other subdirectories, which is not the case here. SUBDIRS must not list sub-subdirectories. If you want to have source code in `src/another` , you must add `src` to the SUBDIRS variable in the top-level *Makefile.am* and add `another` to the SUBDIRS variable in the *src/Makefile.am* file.

And this is the *Makefile.am* file in `src`:

■ Example 5 (vdinner): src/Makefile.am

```
bin_PROGRAMS = vdinner

vdinner_SOURCES = vdinner.c models.c

noinst_HEADERS = models.h

man_MANS = vdinner.1x

EXTRA_DIST = vdinner.1x
```

It also has nothing different from what we discussed in the previous sections.

Automake must be run **from the top-level directory only**. It will automatically descend into all SUBDIRS and create the associated Makefile.in files. If you try running automake from an inner directory it will complain about the lack of a configure.in there, but you don't need one.

10.2 Deep Example

As an example of a deep topology, we'll put all four programs (dinnerd, libdinner, hungryhacker and vdinner) in one package. We'll call this all-in-one package the *dinnersuite*. First we create a `dinnersuite` directory and unpack the distribution tarballs into it. I'll also rename the unpacked directories to remove the version string from the name, but it's not needed. Now we have this directory tree:

```
+ dinnersuite
      |
      +--- dinnerd
      |
      +--- hungryhacker
      |
      +--- libdinner
      |
      +--- vdinner
              |
              +--- src
```

The first thing to do is writing the top-level *configure.in* and *Makefile.am* files. *Makefile.am* becomes:

■ Example 6 (dinnersuite): Makefile.am

```
AUTOMAKE_OPTIONS = foreign
SUBDIRS = dinnerd libdinner hungryhacker vdinner
```

hungryhacker and vdinner depend on libdinner, so they must be built only after libdinner. *configure.in* becomes:

■ Example 6 (dinnersuite): configure.in

```
AC_INIT(dinnerd/dinnerd.c)
AM_INIT_AUTOMAKE(dinnersuite,0.0.1)
AC_CONFIG_SUBDIRS(dinnerd libdinner hungryhacker vdinner)
AC_OUTPUT(Makefile)
```

Notice that we don't need to AC_OUTPUT anything below the top-level directory: this task is delegated to the configure scripts inside each subdirectory by the **AC_CONFIG_SUBDIRS** macro.

If we run autoconf and automake now, we'll have everything generated, but in this particular package we must do more, to deal with dependencies: *hungryhacker* and *vdinner* will check for *libdinner* in their configure scripts, but in configure time the library is not yet built or installed. And we don't need to check it: we're providing it in our package.

Also, we want the user to configure all packages at once, compile all packages at once, then install all packages at once. But the packages that depend on *libdinner* will fail to compile before the library is correctly installed. We will add libtool support to these packages and change their makefile.am files to link the library from the distribution directory. On *hungryhacker*, we change the *configure.in* file to:

■ Example 6 (dinnersuite): hungryhacker/configure.in

```
AUTOMAKE_OPTS = gnu
bin_PROGRAMS = hungryhacker

hungryhacker_SOURCES = hacker.c
hungryhacker_LDADD = ../libdinner/libdinner.la
INCLUDES = -I../libdinner

man_MANS = hungryhacker.1

EXTRA_DIST = sample.sh hungryhacker.1
```

The differences from the stand-alone package are the hungryhacker_LDADD and INCLUDES lines. The INCLUDES line tells the compiler where to find `dinner.h` before it is installed to @prefix@/include. The LDADD line asks to link against the library file in the *libdinner* directory. The .la format depends on libtool for linking, so we must also change *configure.in*, where we add libtool support (AM_PROG_LIBTOOL) and remove the checks for *libdinner* (because we know we have it):

■ Example 6 (dinnersuite): hungryhacker/Makefile.am

```
AC_INIT(hacker.c)
AM_INIT_AUTOMAKE(hungryhacker,0.0.1)
AM_CONFIG_HEADER(config.h)
AM_PROG_LIBTOOL

AC_PROG_INSTALL
AC_PROG_MAKE_SET

AC_PROG_CC
AC_LANG_C

AC_HEADER_STDC

AC_CHECK_HEADERS(assert.h errno.h unistd.h time.h,,AC_MSG_ERROR([
```

```
required header missing]))

AC_OUTPUT(Makefile)
```

The changes to the *vdinner* package are similar. Since the affected directory is `src`, the INCLUDES and LDADD lines need an extra .. each to reach the libdinner directory

You'll notice, if you try to compile the dinnersuite tarball, that the `configure` step takes much long and is redundant, because it calls configure scripts recursively in each directory.

Our dinnersuite package is also space-inefficient, it includes libtool in three of its subdirectories (libdinner, hungryhackers and vdinner). Deep topologies may integrate packages fast, but you should avoid this kind of package. Spending some time on writing an integrated top-level *configure.in* (shallow topology) results in more efficient configuration and building processes.

11 Breeding

Until now we presented a technical tutorial teaching how to use several programs – autoconf, automake, autoheader, aclocal, libtool – but our argument on why using them at all is very short-sighted. It concerns only the direct end-users of your software package: that you should make your software package fast, easy, simple and uneventful to get, build and install. We presented another argument, also short-sighted, and it sounds more like a threat: if you don't make it simple, users will give up on trying your software.

Here we provide advice on using your time and your community's in an efficient way. There will always be work to do in the Free Software camp (new hardware, new algorithms, new standards, new protocols and new habits in an ever-changing society are just some of the driving forces).

Detectability. If your software provides an useful service, it happens quite often that people want to build other software that depends on yours. This software will have its own building and installation process, and if it is written correctly, it won't just assume your software (on which it depends) is there, it'll check for it in some way. When you write a software others will probably depend on, **document** the way(s) to detect it. We depict the most usual, expected behavior:

(a) Install your software in standard places. If you're using **automake** you're probably doing so already. Placing your binary under `/var/coolstuff/lib/etc/boing/bin` is much like not placing it anywhere at all. Doing a `find / -name "foo"` isn't practical and can take hours or even days to complete on systems that mount directories from a network (NFS is sooo transparent...)

(b) Provide version information. The standard is to print a version string and exit without further action when the `-v` or `--version` options are given in the command line. If you want to provide the version information in a different way, it's ok, but you must document it somewhere. A man page is probably the best place.

(c) For libraries, the most common way to check for presence is trying to link a small program. Document in the library changelogs the API changes. This way, if you introduced a float foo_bar(int,int) call in version X.Y.Z, users can detect versions ¿= X.Y.Z by trying to link a program that calls that function.

(d) Many software packages (usually large ones) provide a `programname-config` script that is installed together with the program, that will provide information on version, include file paths, library paths, features, etc. Examples are GTK+ (gtk-config), Gimp (gimp-config), XMMS (xmms-config) and Gnome (gnome-config). Below we show an example of output for the gtk-config script.

```
galadriel:~$ gtk-config
Usage: gtk-config [OPTIONS] [LIBRARIES]
Options:
        [--prefix[=DIR]]
        [--exec-prefix[=DIR]]
        [--version]
        [--libs]
        [--cflags]
Libraries:
        gtk
        gthread
galadriel:~$ gtk-config --version
1.2.8
galadriel:~$ gtk-config --cflags
-I/usr/lib/glib/include
galadriel:~$ gtk-config --libs
-L/usr/lib -L/usr/X11/lib -lgtk -lgdk -rdynamic -lgmodule -lglib
-ldl -lXext -lX11 -lm
```

Writing such a script is trivial. The substitutions for @prefix@ paths and included features can be scripted with autoconf in a reasonably easy way.

(e) Provide m4 macros to perform checks and tests on/for your software. The best behavior is to install these to the `@prefix@/share/aclocal` directory with the name `yourpackage.m4`. This way other developers may use the macros in their *configure.in* input files and **aclocal** will automatically look for the missing macros and place them in the aclocal.m4 of the distribution of the package that depends on yours.

Compatibility. Make your software usable by the widest range of people possible. For example: when writing a library, it is quite easy to turn down C++ programmers by not enclosing the function prototypes in the public header files in `extern "C"` blocks. A way to do this in a portable manner is shown in the `dinner.h` header of *libdinner*, the example library of the libtool section.

Extensibility. Releasing the code under a copyleft license like the GNU GPL does 99You can't extend when you don't have the code (or you have it but are legally forbidden to use/distribute the modified code). But you can go further for the last 1write comments on important parts of your code, emphasizing what have not been implemented in the best possible way (or you didn't spend time finding out what the best possible way was), so that people willing to extend and enhance your code go straight to the spot.

Cooperability. When a group of programmers is working concurrently on a project, it is a good idea to split the functionality of the software across separate source files. Most versioning systems (like CVS) use file granularity: while it will try to merge changes from 2 programmers to a same file into one working frankenstein creature, it is not always successful. Splitting the source across several files reduces the risk of conflict. Some programmers don't like to do it because they needed to write new Makefile rules to track source file dependency. But now you know how to use autoconf and automake, adding another source file means adding one more .c filename to *Makefile.am* and rerunning *automake* and *configure*.

Add, don't divide. Is an operating system with 562 graphical IRC clients better than the same operating system with 67 graphical IRC clients ? Focus on enhancing currently existing software instead of duplicating efforts. If the authors of the current software are unwilling to take your patches or accept your help, don't start from scratch, and make sure not to fall in the same trap your predecessors fell: keep the door open for new developers and people willing to help.

References

[1] Free Software Foundation. *Libtool Manual*. http://www.gnu.org/software/libtool/manual.html.

[2] Mike G. BASH Programming Introduction HOWTO. Technical report, Linux Documentation Project, 1999. http://www.linuxdoc.org/HOWTO/Bash-Prog-Intro-HOWTO.html.

[3] David Mackenzie and Tom Tromey. *Automake Manual*. Free Software Foundation. http://www.gnu.org/manual/automake/index.html.

[4] David Makenzie and Ben Elliston. *Autoconf Manual*. Free Software Foundation. http://www.gnu.org/manual/autoconf/index.html.

[5] Eric S. Raymond. Software Release Practice HOWTO. Technical report, Linux Documentation Project, 1999. http://www.linuxdoc.org/HOWTO/Software-Release-Practice-HOWTO/index.html.

[6] Renè Seindal. *GNU m4 Manual*. Free Software Foundation. http://www.gnu.org/manual/m4/index.html.

[7] Richard M. Stallman. GNU Coding Standards. Technical report, Free Software Foundation, 1990. http://www.gnu.org/prep/standards.html.

[8] Richard M. Stallman and Roland McGrath. *GNU Make Manual*. Free Software Foundation. http://www.gnu.org/manual/make/index.html.

12 Copyright Notice