**Dealing with Libraries**

One of the most useful features of the automake package is its support for creating libraries. In truth, most of this support is provided by another GNU software package, called libtool, distributed separately from automake. However, automake and libtool live quite comfortably together and have grown up in the same backyard, so to speak.

Because of this close integration, you don't have to do much to build software with complex library needs. Libtool sets things up so that you don't even need to decide between static and shared libraries until compile time.

**Libtool Support**

The first thing you need to do to add libtool support to a project is call the AM\_PROG\_LIBTOOL macro in configure.in, with no parameters. Among other things, this macro adds the --enable-shared and --enable-static flags and their --disable-shared and --disable-static counterparts to the configure script and engages libtool integration with automake. By default, the libtool script creates both shared and static libraries. The system administrator can force the building of only static libraries with the --disable-shared flag, or the building of only shared libraries with the --disable-static flag.

The changes to Makefile.am are a little more complex, but by no means tricky. libtool makes use of a special automake primary, \_LTLIBRARIES. Any libraries built with the \_LTLIBRARIES primary will use the libtool wrapper instead of directly calling the standard UNIX library tools, such as ar and ranlib.1

Since libtool creates an abstraction wrapper around the library files it builds, it has to be careful what it calls those files. If it used the common .o or .a file extension for files that aren't really linkable object files, it would run the risk of confusing later stages of the build process. To protect you from this danger, the libtool script tags its libraries with the .la file extension. You should follow this convention when referring to libtool libraries in your Makefile.am file. Although the .la files aren't normal binary library files, you should treat them in your makefiles as if they were-but this means you must be very consistent about your use of libtool. If you try to link a .la library into an application without using libtool, your linker will be unable to figure out what to do with the strangely formatted file. The build will grind to a halt.

Without libtool, your Makefile.am file might contain something like this:

|  |
| --- |
| lib\_LIBRARIES=libgrump.a  libgrump\_a\_SOURCES=grump.c |

If you decided to switch over to libtool, you would have to change these lines to use the .la file extension and the \_LTLIBRARIES primary:

|  |
| --- |
| lib\_LTLIBRARIES=libgrump.la  libgrump\_la\_SOURCES=grump.c |

An interesting characteristic of shared libraries is revealed when you use the noinst\_LTLIBRARIES variable: Shared libraries must always be installed. The shared library is a runtime dependency of the executable. If you install the executable without it, you are installing a broken package. automake realizes this and will always force static libraries when it sees noinst\_LTLIBRARIES, to avoid this potential crisis.

Other useful primaries for dealing with libraries are \_LDFLAGS, \_LDADD, and \_LIBADD. These three variables have very similar names, with closely related functionality. We'll try to unravel the differences here.

The \_LDADD primary is good for adding extra object files and libraries to the link line for a specific binary target. If you define the \_LDADD variable for a target, it will override the global LDADD variable that is normally used. You can't pass linker flags, other than -l and -L, with the \_LDADD primary; if you try to do so, automake will bail out with a friendly warning message. Furthermore, you can use \_LDADD only with executables. If you need to pass extra objects or libraries to a library target, you can use \_LIBADD instead.

The \_LDFLAGS primary is used to add miscellaneous linker parameters to libraries or executables, beyond what \_LDADD and \_LIBADD allow you to pass. You can use it to pass flags directly to the linker, such as the -version-info flag (see Section 3.4.5).

**libtoolize**

Libtool relies on four shell scripts to query the system and set certain things up: config.guess, config.sub, ltconfig, and ltmain.sh. The first two scripts poll the target system and attempt to distill it down to a single canonical name-for example, "i586-pc-linux-gnu." config.guess makes a guess at the target system's canonical name, and config.sub validates that name and expands it to its fully qualified form. Libtool uses the canonical name to decide which set of rules it should use to create libraries for the target system. This is a very important step because different flavors of UNIX can have radically different ways of carrying out this task. If libtool guesses the wrong target operating system, the shared libraries it creates will not work.

The ltconfig script creates a special, customized version of the libtool shell script that gives the libtool package its name. The configure script invokes ltconfig for you, as part of the AM\_PROG\_LIBTOOL macro. The ltconfig script runs several autoconf-like checks on the target system (using config.guess and config.sub), depending on the command line parameters with which configure was invoked. It then writes the results into a newly created libtool script, along with the contents of the ltmain.sh script. Later, during the build process, libtool generates the proper commands to create libraries for that operating system.

These helper scripts are the backbone of the libtool system. Libtool is shipped with a clever utility, called libtoolize, to add this support to your project. Simply call libtoolize in your top-level source directory, and these four files will be copied (or symbolically linked) into your application directory and properly set up. libtoolize cuts down on some potentially hairy maintenance headaches. You can tweak its behavior quite a bit with command line parameters. See the libtool documents for more information.

**A Grumpy Example**

It's time to take a look at some source code. Let's throw together a little shared library, called libgrump, with a couple of small functions. We'll also build an executable that calls into that shared library. If you've done this the hard way before, creating makefiles by hand, you'll be surprised at how easy it is with automake and libtool. See Listings 3.2 through 3.6 for the source code.

|  |
| --- |
| Listing 3.2 Shared Library Header File: grump.h  # include <stdlib.h>  void grump\_some( );  void grump\_a\_lot\_more( ); |

|  |
| --- |
| Listing 3.3 Shared Library Implementation File: grump.c  # include "grump.h"  void grump\_some( )  {  printf("Oh, bother!...\n\n");  }  void grump\_a\_lot\_more( )  {  int i, index;  char \*grumps[5] = { "Aargh!", "Be gone!", "Sigh...",  "Not again!", "Go away!" };  for (i = 0; i < 5; i++)  {  index = (5.0 \* rand( ) / (RAND\_MAX + 1.0));  printf("%s\n", grumps[index]);  }  } |

|  |
| --- |
| Listing 3.4 Main-Module Source Code: main.c  #include "grump.h"  int main(int argc, char \*argv[])  {  grump\_some( );  grump\_a\_lot\_more( );  } |

|  |
| --- |
| Listing 3.5 configure.in for grumpalot  AC\_INIT(grump.c)  AM\_INIT\_AUTOMAKE(grump\_test, 0.0.1)  AC\_PROG\_CC  AM\_PROG\_LIBTOOL  AC\_OUTPUT(Makefile) |

|  |
| --- |
| Listing 3.6 Makefile.am for grumpalot  bin\_PROGRAMS=grumpalot  grumpalot\_SOURCES=main.c  grumpalot\_LDADD=libgrump.la  grumpalot\_LDFLAGS=  include\_HEADERS=grump.h  lib\_LTLIBRARIES=libgrump.la  libgrump\_la\_SOURCES=grump.c  libgrump\_la\_LDFLAGS=-version-info 0:0:0 |

We'll have to run several commands to set things up and then compile it all into the target grumpalot executable. All of these commands should look familiar by now. If not, you should probably go back and read this chapter again. It may seem like a lot of typing, but think of all the wonderful magic that goes on behind the scenes. It would take you weeks of extra work to duplicate all that. To make things even easier, GNOME projects often wrap all these commands up into a single shell script for us, called autogen.sh. We'll learn more about that in Section 3.5.4. Here are the commands you need to compile the example, with the output of the build tools snipped for clarity:

|  |
| --- |
| $ libtoolize  $ aclocal  $ touch NEWS README AUTHORS ChangeLog  $ automake --add-missing --gnu  $ autoconf  $ ./configure  $ make  $ ./grumpalot  Oh, bother!...  Go away!  Be gone!  Not again!  Not again!  Go away! |

**Exploring the Results**

Let's see what libtool has done for us. First, it looks as if libtool has created a .libs subdirectory, filled with every possible incarnation of our libgrump library, including libgrump.a, libgrump.la, and libgrump.so. Another curious fact surfaces when we look at the .libs directory: It also contains a grumpalot file! We have two executables-one in the main directory, and one hidden away with the library files. Let's snoop around and see if we can figure out what's going on. We'll start with the file command, a handy little utility that cracks open a file, examines it, and prints out what it finds.

|  |
| --- |
| $ file grumpalot  grumpalot: Bourne shell script text  $ file .libs/grumpalot  .libs/grumpalot: ELF 32-bit LSB executable, Intel 80386, version 1,  dynamically linked,not stripped |

It appears that libtool has generated some sort of wrapper script around the real executable, which is in the .libs directory. It does this to make sure the executable can properly find and load the shared libraries, even though the shared libraries haven't been installed yet. The wrapper script performs a little fancy juggling of paths that wouldn't normally be necessary with installed libraries; it then invokes the executable in .libs for us. In most cases,2 we can simply invoke the wrapper script as if it were the real executable, passing all the normal command line parameters to it.

Let's find out where the object code for our various grump\_\* functions ended up. We can make use of another common UNIX tool, nm, a utility for dumping the symbol tables of an object file or executable into a legible ASCII format. The output of nm can be voluminous, especially on larger binary files, so we'll pipe the results through grep to filter out the symbols we don't care about.

|  |
| --- |
| $ nm .libs/grumpalot | grep grump  U grump\_a\_lot\_more  U grump\_some  $ nm .libs/libgrump.so | grep grump  000008a4 T grump\_a\_lot\_more  00000880 T grump\_some |

The nm utility uses a handful of single-letter codes to characterize each symbol. The documentation for nm contains a pretty good description of what they all mean. In our case, the libgrump symbols in the binary executable grumpalot are undefined (U). This is normal for any symbols imported from a library into an executable or another library, where the symbols are referenced but not implemented. In libgrump itself, the symbols are marked with a T, which means that nm found their implementations inside the text (or code) section of the library file. The addresses indicate exactly where in the code section the symbols reside. You can find out more about nm by typing info binutils at the command line of a GNU-equipped system and then going into the nm documentation.

Next we'll see how things change with static libraries. We can do this with the --disable-shared flag. We'll have to clean out the old object files to make sure everything recompiles correctly.

|  |
| --- |
| $ ./configure --disable-shared  $ make clean && make all  $ file grumpalot  grumpalot: ELF 32-bit LSB executable, Intel 80386, version 1,  dynamically linked, not stripped  $ nm grumpalot | grep grump  08048634 T grump\_a\_lot\_more  08048620 T grump\_some |

Things are a lot simpler this time. The .libs directory contains only static libraries. The .so files are gone, as is the .libs/grumpalot executable. As we see by the file command, the top-level grumpalot is now the real executable. libtool puts the executable in the .libs directory only when it's creating shared libraries.

Finally, to ease our minds we verify that the libgrump functions are linked directly into the executable. Notice how much larger the symbol addresses are when they are statically linked into the executable. The reason is that the symbols have absolute addresses when they reside in the executable but relative addresses when inside the shared library. The relative addresses make it possible to load shared libraries into different parts of an executable's memory. If a shared library tries to load itself into an area of memory that's already taken, the library loader can dynamically relocate it to another area.

**A Note about Version Numbers**

The path to creating new versions of shared libraries can be a twisted, precarious maze. You have to juggle distribution versions with library versions. The main software package might go through major alterations without a single change to the supporting libraries, and vice versa. It's dangerous to mix package version numbers with library versions because the two are not synony- mous. Package versions are fairly arbitrary, and mostly for the end user's benefit. Shared-library versions, however, refer to very specific changes in functionality. The dynamic library loader uses this version number to determine which implementation of the library to load. If you have only one libgrump.so file, the choice is easy. On UNIX, however, it's possible to have many versions of the same library in the same directory at the same time. The library version number is the loader's only hope for finding the correct library for a given application.

You can set the version for a library with an \_LDFLAGS primary in your Makefile.am file, with the -version-info flag:

|  |
| --- |
| libgrump\_la\_LDFLAGS=-version-info 5:1:2 |

The three numbers stand for CURRENT:REVISION:AGE, or C:R:A for short. The libtool script typically tacks these three numbers onto the end of the name of the .so file it creates. The formula for calculating the file numbers on Linux and Solaris is (C - A).(A).(R), so the example given here would create the file libgrump.so.3.2.1. Other operating systems might use a different library file name convention; libtool takes care of the details.

As you release new versions of your library, you will update the library's C:R:A. Although the rules for changing these version numbers can quickly become confusing, a few simple tips should help keep you on track. The libtool documentation goes into greater depth.

In essence, every time you make a change to the library and release it, the C:R:A should change. A new library should start with 0:0:0. Each time you change the public interface (i.e., your installed header files), you should increment the CURRENT number. This is called your interface number. The main use of this interface number is to tag successive revisions of your API.

The AGE number is how many consecutive versions of the API the current implementation supports. Thus if the CURRENT library API is the sixth published version of the interface and it is also binary compatible with the fourth and fifth versions (i.e., the last two), the C:R:A might be 6:0:2. When you break binary compatibility, you need to set AGE to 0 and of course increment CURRENT.

The REVISION marks a change in the source code of the library that doesn't affect the interface-for example, a minor bug fix. Anytime you increment CURRENT, you should set REVISION back to 0.