

Autotools Tutorial

Hendrik Rittich

7. November 2006

Inhaltsverzeichnis

1	Einleitung	2
2	Einfaches Kompilieren	2
2.1	Ein einfaches Programm	2
2.2	build.sh	3
3	GNU Make	4
3.1	Warum Make?	4
3.2	Aufbau eines Makefiles	4
3.3	Unser Makefile	5
3.4	Variablen	6
4	GNU Autoconf	7
4.1	Portabilität durch Autoconf	7
4.2	Funktionsweise	7
4.3	M4	8
4.4	Configure	8
4.5	The Power of Autoconf	10
5	GNU Automake	13
5.1	Makefiles aus dem Automaten	13
5.2	Makefile.am	14
5.3	Unterverzeichnisse	15
5.4	Anmerkung	16
6	Libraries	17
6.1	Einleitung	17
6.2	Libtool	17
6.3	Automake	18
6.4	Statische Libraries	19
7	Fazit	19

1 Einleitung

Der Begriff GNU Autotools fasst eine Gruppe an Programmen zusammen, mit denen sich unter verschiedenen Unix-Derivaten Programme kompilieren lassen. Dabei übernehmen sie die Konfiguration, die Überprüfung auf Abhängigkeiten, und das anschließende Build. Zu den GNU Autotools zählen normalerweise GNU Make¹, Autoconf², GNU libtool³ und GNU Automake⁴

Voraussetzungen, um den Artikel zu verstehen, sind einfache C-Kenntnisse und ein wenig Erfahrung mit der Shell. Natürlich sollten die oben genannten Programme auf einem lauffähigen System installiert sein, außerdem wäre es nicht schlecht, weil ich es in einem Beispiel verwende, GTK+⁵ installiert zu haben.

2 Einfaches Kompilieren

2.1 Ein einfaches Programm

Wir erstellen ein simples C Programm, das aus mehreren Dateien besteht.

```
1  /* main.c */
2  #include "greeting.h"
3
4  int main (int argc, char* argv[]) {
5      greeting_do ();
6
7      return 0;
8  }
9
1
1  /* greeting.c */
2  #include <stdio.h>
3
4  #include "greeting.h"
5
6  void greeting_do (void) {
7      printf("Hello,\nHow are you?\n");
8  }
9
1
1  /* greeting.h */
```

¹GNU Make: <http://www.gnu.org/software/autoconf/>

²Autoconf: <http://www.gnu.org/software/autoconf/>

³GNU libtool: <http://www.gnu.org/software/libtool/>

⁴GNU Automake: <http://www.gnu.org/software/automake/>

⁵GTK+: <http://www.gtk.org/>

```
2
3  #ifndef TOOLSPROG_GREETING_H
4  #define TOOLSPROG_GREETING_H
5
6  void greeting_do (void);
7
8  #endif
```

2.2 build.sh

Das Programm wollen wir nun übersetzen. Die einfachste Möglichkeit ist, sich eine Shell zu greifen und den Compiler von Hand aufzurufen. Da wir nicht immer alles neu eintippen wollen, schreiben wir alle Befehle in ein Shell-Script. Zum Kompilieren verwende ich hier in den Beispielen den GCC⁶. Den Compiler gibt es für eigentlich jede Plattform, darum ist er für unsere Demonstrationszwecke gut geeignet.

```
1  #!/bin/sh
2  # build.sh
3  gcc -o greeting.o -c greeting.c
4  gcc -o main.o -c main.c
5  gcc -o prog1 main.o greeting.o
```

Das Script machen wir ausführbar und rufen es auf. Das geht mit folgenden Shell Befehlen.

```
$ chmod a+x build.sh
$ ./build.sh
```

Nach kurzer Zeit ist unser Programm kompiliert und wir können es benutzen.

```
$ ./prog1
Hello,
How are you?
```

Unser simples Buildsystem besteht nun aus einem Shell script, das unser Programm erzeugt. Als erstes werden die Dateien *greeting.c* und *main.c* kompiliert. Danach werden die erstellten Object Dateien zum Programm *prog1* zusammen gelinkt.

⁶The GNU Compiler Collection: <http://gcc.gnu.org>

3 GNU Make

3.1 Warum Make?

Unser Programm wird anstandslos kompiliert und läuft wunderbar. Naja... es tut halt das, was es soll. Warum sollte man jetzt also noch mehr Zeit investieren, um ein komplexeres Build System zu benutzen, als unsere *build.sh*?

Stellen wir uns einfach mal vor, unser Programm würde nicht aus drei, sondern aus 300 Dateien bestehen. Wir haben einen Fehler in der Datei *greeting.c* gefunden und korrigiert. Jetzt wollen wir das Programm testen. Unser simples Script würde alle 300 Dateien neu kompilieren und wir müssten eine halbe Stunde warten, bevor wir das Programm testen könnten. Es würde vollkommen ausreichen, nur die Datei *greeting.c* zu kompilieren und dann Alles zusammenzulinken, aber unser Script ist einfach zu dumm dafür. Sicherlich könnten wir dem Script ein wenig mehr Intelligenz einhauchen, aber für unsere Zwecke gibt es schon eine Lösung, nämlich GNU Make...

3.2 Aufbau eines Makefiles

Make generiert die Dateien neu, deren Quellcode sich geändert hat. Dazu müssen wir eine Zielfeile und ihre Quelldateien angeben. Wir können so viele Zielfeilen in einem Makefile eintragen, wie wir möchten. Die Struktur dafür sieht so aus:

```
Zielfeile: Quelldatei1 Quelldatei2 ...
    Anweisung1
    Anweisung2
    ...
```

Beispiel:

```
hello: hello.c
    gcc -o hello hello.c
```

Die Zielfeile muss immer am Anfang einer Zeile stehen. Danach folgt der Doppelpunkt gefolgt von den Quelldateien. Hier müssen dann alle Dateien angegeben werden, die den Inhalt der Zielfeile beeinflussen. Zum Beispiel sollten, zusätzlich zu der eigentlichen Quellcode Datei, alle eingebundenen Headerdateien, die man selbst ändert angegeben werden.

In den folgenden Zeilen werden alle Befehle aufgeführt, die benötigt werden, um die Zielfeile zu erstellen. Alle Befehle müssen mit einem oder mehreren Tabs eingerückt werden. Wichtig dabei ist, dass es *echte Tabs* und keine Leerzeichen sind!!!

Bei der Zielfeile muss es sich übrigens nicht immer um eine wirkliche Datei handeln. Folgender Abschnitt in einem Makefile kann dazu genutzt werden, um das Verzeichnis nach dem Build wieder aufzuräumen.

```
clean:
    rm -f hello
```

3.3 Unser Makefile

Erstellen wir nun das Makefile für unser Programm und speichern es unter dem Namen *makefile* ab.

```
1  # makefile
2  prog2: main.o greeting.o
3      gcc -o prog2 main.o greeting.o
4
5  main.o: main.c greeting.h
6      gcc -o main.o -c main.c
7
8  greeting.o: greeting.c greeting.h
9      gcc -o greeting.o -c greeting.c
10
11 clean:
12     rm -f prog2 main.o greeting.o
13
```

Der Aufruf von Make lautet nun wie folgt

```
$ make -f makefile Zieldatei
```

Wenn die Datei, wie in unserem Fall, *Makefile* oder *makefile* heißt, kann man auf die Angabe des Dateinamens auch verzichten. Make benutzt dann die Datei aus dem aktuellen Verzeichnis. Lässt man die Zieldatei weg, baut Make die erste Zieldatei, die es finden kann. Praktisch bedeutet das, dass folgender Befehl unser Programm erzeugt.

```
$ make
gcc -o main.o -c main.c
gcc -o greeting.o -c greeting.c
gcc -o prog2 main.o greeting.o
```

Und folgender Befehl räumt alles wieder auf.

```
$ make clean
rm -f prog2 main.o greeting.o
```

3.4 Variablen

Das ist schonmal ein großer Fortschritt. Ändern wir die Datei *main.c* werden nur die Dateien *main.o* und *prog2* neu erzeugt. Ändern wir aber die Datei *greeting.h* wird auch noch die Datei *greeting.o* erzeugt. Make ist in der Lage, die Abhängigkeiten der einzelnen Dateien festzustellen und ruft die Befehle in der richtigen Reihenfolge auf.

Unser Makefile ist allerdings noch reichlich unflexibel. Wenn wir zum Beispiel den Namen der Executable ändern möchten, oder den Aufruf des Compilers, müssen wir durch das ganze Makefile wandern und den entsprechenden Text ändern. In GNU Make gibt es dafür eine einfache Lösung: Variablen. Haben wir eine Variable deklariert, können wir sie einfach überall einfügen und müssen nur an einer Stelle das Makefile ändern. Die Deklaration sieht folgendermaßen aus.

```
VARIABLEN_NAME = Variablen Inhalt
```

Einfügen können wir den Variableninhalt an beliebiger Stelle im Makefile auf folgende Weise.

```
$(VARIABLEN_NAME)
```

Schreiben wir unser Makefile also auf folgende Weise um.

```
1  # makefile
2  BIN = prog3
3  OBJS = main.o greeting.o
4  CC = gcc
5  CFLAGS = -O2
6  LDFLAGS = -s
7
8  $(BIN): $(OBJS)
9          $(CC) $(LDFLAGS) -o $(BIN) $(OBJS)
10
11 main.o: main.c greeting.h
12         $(CC) $(CFLAGS) -o main.o -c main.c
13
14 greeting.o: greeting.c greeting.h
15         $(CC) $(CFLAGS) -o greeting.o -c greeting.c
16
17 clean:
18         rm -f $(BIN) $(OBJS)
19
```

Testen wir also nun unser neues Makefile.

```
$ make
gcc -O2 -o main.o -c main.c
gcc -O2 -o greeting.o -c greeting.c
gcc -s -o prog3 main.o greeting.o
```

Hinzugekommen sind noch die Compiler Argumente *-O2* und *-s*. Diese Argumente sorgen dafür, dass der Compiler das Programm optimiert und die Debugging Symbole entfernt.

GNU Make kann noch viel mehr. Zum Beispiel if-Bedingungen. Wer mehr über GNU Make wissen will, kann einfach mal in die Hilfe oder ins Internet⁷ schaun.

```
$ info make
```

4 GNU Autoconf

4.1 Portabilität durch Autoconf

Mit dem bis jetzt gesammelten Wissen lässt sich unser Programm schon unter den meisten Linux Distributionen übersetzen. Aber was würde zum Beispiel passieren, wenn auf unserem Zielsystem der GCC nicht installiert ist? Oder wie sieht es aus, wenn wir eine bestimmte Header-Datei einbinden, die auf unterschiedlichen Systemen in unterschiedlichen Verzeichnissen liegt? Wir müssten für jedes Betriebssystem ein eigenes Makefile erstellen. Früher oder später kommt man also zu dem Schluss, dass eine generelle Lösung her muss. Und hier kommt Autoconf ins Spiel.

4.2 Funktionsweise

Wie kann uns also Autoconf bei unserem Problem helfen? Zuallererst benennen wir die Datei *makefile* in *makefile.in* um. In diese Datei fügen wir spezielle Variablen ein, die Autoconf dann später ersetzt und uns eine gültige *makefile* generiert. Dafür erstellt Autoconf ein Shell Script, das auf dem Zielsystem aufgerufen wird und die Aufgabe erledigt.

Als weiteres Feature erstellt uns Autoconf eine Header Datei *config.h*, in der, je nach Systemkonfiguration, bestimmte Makros definiert sind oder nicht.

Die Aufgabe für uns ist jetzt also, eine *configure.in* Datei zu erstellen, mit deren Hilfe das *configure* Shell Script erstellt wird. Außerdem müssen wir die Datei *makefile.in* so anpassen, dass es mit Autoconf zusammen arbeitet.

⁷http://vertigo.hsrl.rutgers.edu/ug/make_help.html

4.3 M4

Noch eine kleine Information zur Funktionsweise von Autoconf. Autoconf basiert auf GNU M4⁸. M4 ist ein Makro Prozessor. Unsere *configure.in* ist also ein M4 Script. Praktisch bedeutet das, dass die *configure.in* schon unser Shell Script ist. Nur dass bestimmte Makros durch anderen Script Code noch ersetzt werden, bevor das Script zum Einsatz kommt. In unserer *configure.in* können wir also zusätzlich zu den Autoconf Makros noch alle Shell Befehle verwenden.

Makros haben die folgende Form.

```
MAKRO(Arg1, Arg2, ... , ArgN)
```

Sollten wir als Argument einen längeren Text benutzen wollen, sollten wir diesen in eckige Klammern setzen. Wenn wir nämlich in diesem Text ein Komma benutzen, dann würde M4 dieses Komma als Ende des Arguments interpretieren, was wir nicht wollen. Sollten wir zusätzlich Makronamen in unserem Text verwenden und nicht wollen, dass diese durch den Inhalt des Makros ersetzt werden, müssen wir doppelte eckige Klammern verwenden. Beispiel:

```
MAKRO(Argument, [Ein langes Argument], [[MAKRO ist ein Makro]])
```

4.4 Configure

So viel zur Theorie. Jetzt kommt die praktische Arbeit. Als erstes passen wir unser Makefile an. Dafür fügen wir die Autoconf Variablen der folgenden Form ein.

```
@VARIABLEN_NAME@
```

Nun sieht unsere *makefile.in* so aus:

```
1  # makefile
2  BIN = prog4
3  OBJS = main.o greeting.o
4  CC = @CC@
5  CFLAGS = @CFLAGS@ @DEFS@
6  LDFLAGS = @LDFLAGS@ @LIBS@
7
8  $(BIN): $(OBJS)
9          $(CC) $(LDFLAGS) -o $(BIN) $(OBJS)
10
11 main.o: main.c greeting.h
12          $(CC) $(CFLAGS) -o main.o -c main.c
```

⁸<http://www.gnu.org/software/m4/>

```

13
14  greeting.o: greeting.c greeting.h
15          $(CC) $(CFLAGS) -o greeting.o -c greeting.c
16
17  clean:
18          rm -f $(BIN) $(OBJS)
19

```

Als nächstes öffnen wir eine Shell, wechseln in unser Programmverzeichnis und starten das Programm *autoscan*.

```
$ autoscan
```

Jetzt erscheint in dem Verzeichnis die Datei *configure.scan*. Diese Datei enthält das Grundgerüst für unser Autoconf Script und sollte, wie folgt, aussehen.

```

1  #                                                    -*- Autoconf -*-
2  # Process this file with autoconf to produce a configure script.
3
4  AC_PREREQ(2.59)
5  AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
6  AC_CONFIG_SRCDIR([greeting.c])
7  AC_CONFIG_HEADER([config.h])
8
9  # Checks for programs.
10 AC_PROG_CC
11
12 # Checks for libraries.
13
14 # Checks for header files.
15
16 # Checks for typedefs, structures, and compiler characteristics.
17
18 # Checks for library functions.
19
20 AC_CONFIG_FILES([makefile])
21 AC_OUTPUT

```

Wir benennen die Datei in *configure.in* um und bearbeiten sie, wie folgt.

```

1  dnl configure.in
2  AC_PREREQ(2.59)
3  AC_INIT(Prog4, 1.0, noreply@mail.com)
4  AC_CONFIG_SRCDIR([main.c])

```

```

5  AC_CONFIG_HEADER([config.h])
6
7  # Checks for programs.
8  AC_PROG_CC
9
10 # Checks for libraries.
11
12 # Checks for header files.
13
14 # Checks for typedefs, structures, and compiler characteristics.
15
16 # Checks for library functions.
17
18 AC_CONFIG_FILES([makefile])
19 AC_OUTPUT

```

In Zeile (2) legen wir fest, welche Autoconf-Version mindestens benötigt wird, um aus der *configure.in* die Datei *configure* zu erstellen. Zeile (3) initialisiert Autoconf und legt ein paar generelle Informationen, wie Programmname und -version fest. Zeile (8) sucht nach dem auf dem System installierten C Compiler. In Zeile (18) werden die zu erstellenden Dateien festgelegt und in Zeile (19) werden die Dateien dann endlich erzeugt.

Erstellen wir nun endlich unser Script! Dazu sind folgende Befehle nötig.

```

$ aclocal
$ autoheader
$ autoconf

```

Als erstes werden die benutzten Autoconf Makros im aktuellen Verzeichnis zwischengespeichert. Dann erstellen wir die Datei *config.h.in*. Zu guter Letzt erstellen wir das Script *configure*.

Mit folgendem Befehl erstellen wir nun unser Makefile.

```

$ ./configure

```

Jetzt können wir, wie gewohnt Make aufrufen.

```

$ make
gcc -g -O2 -o main.o -c main.c
gcc -g -O2 -o greeting.o -c greeting.c
gcc -o prog4 main.o greeting.o

```

4.5 The Power of Autoconf

Ziehen wir doch ein wenig Nutzen aus unserem bisherigen Wissen. Wir wollen ein Programm schreiben, das uns den Sinus einer bestimmten Zahl anzeigt. Des Weiteren wollen wir die Information mit GTK+ anzeigen, wenn es installiert ist. Beginnen wir mit der *configure.in*.

```

1  dnl configure.in
2  AC_PREREQ(2.59)
3  AC_INIT(Prog5, 1.0, noreply@mail.com)
4  AC_CONFIG_SRCDIR([main.c])
5  AC_CONFIG_HEADER([config.h])
6
7  # Checks for programs.
8  AC_PROG_CC
9  PKG_PROG_PKG_CONFIG([0.14.0])
10
11 # Checks for libraries.
12 AC_CHECK_LIB(m, sinf, , AC_MSG_ERROR([Math lib not found]))
13 PKG_CHECK_MODULES(GTK,
14                    [gtk+-2.0 >= 2.4.0],
15                    [AC_DEFINE([HAVE_GTK], , [GTK+ is supported])],
16                    [AC_MSG_RESULT(no)])
17 LIBS="$LIBS $GTK_LIBS"
18 CFLAGS="$CFLAGS $GTK_CFLAGS"
19
20 # Checks for header files.
21 AC_CHECK_HEADER([math.h] , , [AC_MSG_ERROR([math.h not found])])
22
23 # Checks for typedefs, structures, and compiler characteristics.
24
25 # Checks for library functions.
26
27 AC_CONFIG_FILES([makefile])
28 AC_OUTPUT

```

Schauen wir uns an, was da im Detail passiert.

```
AC_CHECK_LIB(m, sinf, , AC_MSG_ERROR([Math lib not found]))
```

Wir überprüfen ob sich die Funktion *sinf* in der Library *m* befindet.

Wenn die Library gefunden wird, wird der Variable *LIBS* die Bibliothek *m* hinzugefügt.

Sollte die Library nicht gefunden werden, geben wir mit dem Makro *AC_MSG_ERROR* eine Fehlermeldung aus und beenden das Script.

```
AC_CHECK_HEADER([math.h] , , [AC_MSG_ERROR([math.h not found])])
```

Wir überprüfen, ob auf dem System die Datei *math.h* vorhanden ist. Wenn wir die Datei nicht finden, beenden wir das Script mit einer Fehlermeldung.

```
PKG_PROG_PKG_CONFIG([0.14.0])
```

Nun müssen wir GTK+ konfigurieren. Dazu benutzen wir das Programm pkg-config. Als erstes testen wir, dass pkg-config in einer Version größer als 0.14 installiert ist.

```
PKG_CHECK_MODULES(GTK,  
                  [gtk+-2.0 >= 2.4.0],  
                  [AC_DEFINE([HAVE_GTK],,[GTK+ is supported])],  
                  [AC_MSG_RESULT(no)])
```

Hier testen wir mit Hilfe von pkg-config, ob wir GTK+ verwenden können. Als erstes legen wir fest, dass allen Konfigurationsvariablen das Prefix „GTK“ vorangestellt wird.

Dann überprüfen wir, ob GTK+ in einer Version größer als 2.4 vorhanden ist. Ist dies der Fall, definieren wir in der *config.h*, den Makro *HAVE_GTK*, andernfalls geben wir den Text „no“ aus.

```
LIBS="$LIBS $GTK_LIBS"  
CFLAGS="$CFLAGS $GTK_CFLAGS"
```

Hier werden den normalen Konfigurations-Variablen die gerade ermittelten Konfigurations-Daten hinzugefügt. Beachtet das vorher festgelegte Prefix „GTK“.

Erstellen wir nun unser Programm.

```
1  /* main.c */  
2  #include <stdio.h>  
3  #include <math.h>  
4  
5  #ifdef HAVE_CONFIG_H  
6  #include "config.h"  
7  #endif  
8  
9  #ifdef HAVE_GTK  
10 #include <gtk/gtk.h>  
11 #endif  
12  
13 void sin_info (float rad) {  
14     float val;  
15     val = sinf (rad); /* berechne den sinus von rad */  
16  
17 #ifdef HAVE_GTK  
18     {  
19         GtkWidget* dlg;  
20         dlg = gtk_message_dialog_new (NULL,  
21                                     GTK_DIALOG_MODAL,
```

```

22             GTK_MESSAGE_INFO,
23             GTK_BUTTONS_OK,
24             "sin(%f) = %f", rad, val);
25     gtk_dialog_run (GTK_DIALOG (dlg));
26     gtk_widget_destroy (dlg);
27 }
28 #else
29     printf("sin(%f) = %f\n", rad, val);
30 #endif
31 }
32
33 int main (int argc, char* argv[]) {
34 #ifdef HAVE_GTK
35     gtk_init (&argc, &argv);
36 #endif
37
38     sin_info (0.5 * 3.141593);
39
40     return 0;
41 }
42

```

Ich schätze, das Programm ist selbsterklärend. Jeder sollte nun auch in der Lage sein, das Makefile dafür selber zu schreiben. Also Kompilieren...

```

$ aclocal
$ autoheader
$ autoconf
$ ./configure
[...]
$ make
[...]

```

...und testen...

```
./prog5
```

5 GNU Automake

5.1 Makefiles aus dem Automaten

Eigentlich haben wir ja nun alles, was wir brauchen. Wir können unsere Programm ohne viel Aufwand auf vielen Betriebssystemen zum Laufen bringen. Warum sollte man jetzt noch mehr lernen? Stellen wir uns nochmal vor, wir hätten ein Programm mit 300 Dateien. Hierfür ein Makefile zu schreiben und

auch noch dafür zu sorgen, dass alle Abhängigkeiten richtig eingehalten werden, ist ganz schön lästig. Wer möchte bitte alle Dateien eines Programms öffnen, um zu schauen, welche Headerdateien diese benutzen?

Wir kommen also zu dem Schluss, dass es nicht schlecht wäre, wenn uns jemand diese Arbeit abnehmen würde. Unser Programm der Wahl ist Automake.

5.2 Makefile.am

Nehmen wir also wieder unserer einfaches Beispiel vom Anfang. Wir erstellen nun eine Datei *Makefile.am*, aus der unsere Datei *Makefile.in* generiert wird.

```
1 # Makefile.am
2 bin_PROGRAMS = prog6
3 prog6_SOURCES = main.c greeting.c greeting.h
```

Das sieht doch ziemlich einfach aus. In Zeile (1) legen wir fest, wie die Programme heißen, die wir erstellen wollen. In Zeile (2) Teilen wir Automake mit, welche Quelldateien zu unserem Programm gehören.

Damit wir Automake nutzen können, müssen wir die Datei *configure.in*, wie folgt, anpassen.

```
1 dnl configure.in
2 AC_PREREQ(2.59)
3 AC_INIT(Prog6, 1.0, noreply@mail.com)
4 AM_INIT_AUTOMAKE
5 AC_CONFIG_SRCDIR([main.c])
6 AC_CONFIG_HEADER([config.h])
7
8 # Checks for programs.
9 AC_PROG_CC
10
11 # Checks for libraries.
12
13 # Checks for header files.
14
15 # Checks for typedefs, structures, and compiler characteristics.
16
17 # Checks for library functions.
18
19 AC_CONFIG_FILES([Makefile])
20 AC_OUTPUT
```

Automake benötigt noch zusätzliche Scripte und Dateien, um arbeiten zu können. Legen wir diese also an.

```

$ aclocal
$ autoheader
$ touch NEWS README AUTHORS ChangeLog
$ automake --add-missing
configure.in: installing './install-sh'
configure.in: installing './missing'
Makefile.am: installing './INSTALL'
Makefile.am: installing './COPYING'
Makefile.am: installing './depcomp'
$ autoconf

```

Jetzt können wir unser Programm einfach kompilieren und testen.

```

$ ./configure
[...]
$ make
[...]
$ ./prog6
Hello,
How are you?

```

Zum aufräumen können wir einfach folgendes aufrufen.

```
$ make clean
```

Das ist allerdings nicht das einzige spezial Target, dass uns automake generiert. Hier sind die wichtigsten:

```

clean      Verzeichnis aufräumen
distclean  wie clean, nur gründlicher
install    Programm installieren
dist       Tarball mit dem Quelltext erstellen

```

5.3 Unterverzeichnisse

So langsam wird's unübersichtlich in unserem Verzeichnis. Autoconf und Automake haben so viele Dateien angelegt, dass wir schon Mühe haben, unsere Quellcode Dateien zu finden. Darum wollen wir diese Dateien jetzt in das Unterverzeichnis *src* verschieben.

Wir erstellen also das Verzeichnis *src* und verschieben die Dateien *Makefile.am*, *main.c*, *greeting.c* und *greeting.h* hinein.

Jetzt erstellen wir im Hauptverzeichnis eine neue *Makefile.am*.

```

1 # Makefile.am
2 SUBDIRS = src
3

```

In der Datei geben wir einfach alle Unterverzeichnisse an, in denen weitere Makefiles liegen.

Da unser configure Script auch die Datei *Makefile* im Verzeichnis *src* erzeugen soll, müssen wir unsere Autoconf Datei, wie folgt, anpassen.

```
1  dnl configure.in
2  AC_PREREQ(2.59)
3  AC_INIT(Prog7, 1.0, noreply@mail.com)
4  AM_INIT_AUTOMAKE
5  AC_CONFIG_SRCDIR([src/main.c])
6  AC_CONFIG_HEADER([config.h])
7
8  # Checks for programs.
9  AC_PROG_CC
10
11 # Checks for libraries.
12
13 # Checks for header files.
14
15 # Checks for typedefs, structures, and compiler characteristics.
16
17 # Checks for library functions.
18
19 AC_CONFIG_FILES([Makefile
20                  src/Makefile])
21 AC_OUTPUT
```

Das Programm erstellen wir mit den üblichen Kommandos.

```
$ autoconf
$ automake
$ ./configure
[...]
$ make
```

5.4 Anmerkung

Noch eine kleine Ergänzung. Automake erstellt die Makefiles so, dass es in der Regel reicht, nur die Makefiles aufzurufen, wenn sie einmal erstellt wurden.

```
$ make
```

Das bedeutet praktisch, dass das Makefile sich selber neu generiert, wenn die Dateien *configure.in* oder *Makefile.am* geändert wurden.

Außerdem ist es nicht nötig, dass auf dem System, auf dem das Programm kompiliert werden soll, Autoconf und Automake vorhanden sind. Sind erstmal die Dateien *configure* und *Makefile.in* erstellt, wird nur noch Make benötigt.

6 Libraries

6.1 Einleitung

Wir haben schon so einiges geschafft. Nachdem es im letzten Kapitel nochmal einfacher wurde, schauen wir uns jetzt noch ein etwas komplexeres Thema an. Nämlich Bibliotheken oder Libraries. Unser Ziel ist es, alle Funktionen aus der Datei *greeting.c* in eine Library auszulagern.

6.2 Libtool

Zum Erstellen der Libraries werden wir Libtool verwenden. Aber warum zum Teufel brauchen wir schon wieder ein neues Tool? Naja, wenn wir statische Libraries erstellen, ist das meistens kein Problem. Da hängt es lediglich vom Compiler ab, wie die Dateien übersetzt und gelinkt werden müssen. Aber wenn wir dynamische Libraries⁹ verwenden, wird's schwierig, denn nun hängt es vom Compiler und vom Betriebssystem ab, wie die Dateien kompiliert werden müssen. Damit wir nicht für jedes Betriebssystem ein eigenes Makefile schreiben müssen, lassen wir uns von Libtool helfen.

Würden wir unser Programm von Hand übersetzen, würde das mit der Hilfe von Libtool, wie folgt, aussehen.

```
1  #!/bin/sh
2  # build.sh
3  libtool --mode=compile gcc -o greeting.o -c greeting.c
4  libtool --mode=link gcc -o libgreeting.la greeting.lo \
5      -rpath /usr/local/lib -lm
6
7  gcc -o main.o -c main.c
8  libtool --mode=link gcc -o prog8 libgreeting.la main.o
```

Wir haben vor fast alle Compileraufrufe, den Befehl *libtool* vorangestellt. Libtool filtert unseren Compiler aufruf und fügt, je nach Betriebssystem, noch entsprechende Compilerflags ein. Es ist schon verwunderlich, dass wir nun, statt Dateien mit der Endung *.so* und *.o*, Dateien mit den Endungen *.la* und *.lo* erstellen. Diese Dateien sind in Wirklichkeit keine Libraries beziehungsweise Objektdateien, sondern Wrapperscripts, die von Libtool erstellt werden, um uns den Umgang mit den Libraries zu vereinfachen. Die

⁹Dynamische Libraries haben meistens die Endung *.so*

eigentlichen Dateien liegen im versteckten Verzeichnis *.libs*. Weil Libtool diese ganzen Scripts verwendet, müssten wir auch Libtool benutzen, um das Programm und die Libraries auf dem System zu installieren. Da wir aber Automake verwenden wollen, kümmern wir uns nicht weiter darum, sondern schauen uns stattdessen an, wie wir Libtool in Automake verwenden.

6.3 Automake

Also passen wir unsere *Makefile.am* an.

```
1 # src/Makefile.am
2 lib_LTLIBRARIES = libgreeting.la
3 libgreeting_la_SOURCES = greeting.c greeting.h
4
5 bin_PROGRAMS = prog9
6 prog9_SOURCES = main.c
7 prog9_LDADD = libgreeting.la
8
```

Ich denke, das Script erklärt sich von selbst. Wir erstellen die Library und linken sie in unser Programm. Mit dem Makro *prog9_LDADD* linken wir die neue Library in unser Programm.

In unserem configure Script müssen wir außerdem Libtool initialisieren.

```
1 dnl configure.in
2 AC_PREREQ(2.59)
3 AC_INIT(Prog7, 1.0, noreply@mail.com)
4 AM_INIT_AUTOMAKE
5 AC_CONFIG_SRCDIR([src/main.c])
6 AC_CONFIG_HEADER([config.h])
7
8 # Checks for programs.
9 AC_PROG_CC
10 AC_PROG_LIBTOOL
11
12 # Checks for libraries.
13
14 # Checks for header files.
15
16 # Checks for typedefs, structures, and compiler characteristics.
17
18 # Checks for library functions.
19
20 AC_CONFIG_FILES([Makefile
21                 src/Makefile])
22 AC_OUTPUT
```

War ja nun nicht so schwer. Also kompilieren wir...

```
$ alocal
$ autoconf
$ libtoolize
$ ./configure
[...]
$ make
```

Wie wir sehen, müssen wir einmalig das Programm *libtoolize* aufrufen, um ein paar Scripts zu erstellen, die Automake benötigt.

Auch wenn es sich hier um ein wirklich komplexes Thema handelt, ist es mit Automake ziemlich einfach, das Programm zum Laufen zu bekommen.

6.4 Statische Libraries

Wollen wir jetzt statt dynamischer Libraries, statische, können wir dies ohne Probleme mit unserem bisher erstellten Script machen.

```
$ ./configure --enable-shared=no
$ make clean
$ make
```

Wir schalten einfach die Erstellung der dynamischen Libraries beim Konfigurieren aus.

Wollen wir komplett auf dynamische Libraries verzichten, können wir auch komplett auf Libtool verzichten. Wer also keine dynamischen Libraries braucht, entfehrt alle libtool-spezifischen Aufrufe und erstellt in der Datei *Makefile.am* die Libraries auf folgende Weise.

```
lib_LIBRARIES = libgreeting.a
libgreeting_a_SOURCES = greeting.c greeting.h
```

7 Fazit

Autotools sind cool ;-)

Zumindest sind sie ein mächtiges Werkzeug, um plattformunabhängige Programme zu erstellen. Wenn man ein Programm nur für ein oder zwei Plattformen erstellt kann man natürlich auch einfach nur Make benutzen, um die Programme zu erstellen. Es kann aber nie schaden, auf zukünftige Portierungen vorbereitet zu sein.

Ich hoffe ich konnte euch einen kleinen Einblick in die Autotools vermitteln und es hat euch Spaß gemacht, den Artikel zu lesen. Es gibt noch etliche Dinge, die mit den Autotools möglich sind, die ich hier nicht vorstellen kann. Wer sich also weiter mit dem Thema beschäftigen möchte findet in den Info-Documents zu den einzelnen Tools jede Menge weiterer Informationen.