Herramientas de desarrollo bajo Linux

Antonio Luque Estepa
Escuela Superior de Ingenieros de Sevilla
aluque@zipi.us.es

12 de junio de 2000

Copyright © Antonio Luque Estepa <aluque@zipi.us.es>

Se concede permiso para realizar y distribuir copias de este documento, siempre proporcionando esta nota y la nota del copyright en todas las copias.

Se concede permiso para copiar y distribuir versiones modificadas de este documento, bajo las condiciones propuetas para copias completas. Las traducciones se incluyen en la categoría de "versiones modificadas".

Se permite y recomienda redistribuir comercialmente este documento; sin embargo, se sugiere que el distribuidor contacte antes con el autor, con el fin de disponer siempre de la versión más actualizada. Con este mismo propósito, se recomienda también a los traductores que contacten con el autor.

Índice General

1 Introducción **3**
- 1.1 Portabilidad . . . 3
- 1.2 Software libre . . . 4

2 Compilación y enlazado. GCC y compañía **4**
- 2.1 Introducción . . . 4
- 2.2 Herramientas disponibles . . . 4
- 2.3 Ejemplos . . . 5
- 2.4 Detalles . . . 6
- 2.5 Inconvenientes . . . 7

3 Dependencias. GNU Make **7**
- 3.1 Utilidad . . . 7
- 3.2 Ejemplos . . . 7
- 3.3 Detalles . . . 9
- 3.4 Inconvenientes . . . 10

4 Configuración automática. Autoconf **10**
- 4.1 Introducción . . . 10
- 4.2 Funcionamiento . . . 11
  - 4.2.1 `configure.in` . . . 12
- 4.3 Ejemplos . . . 12
- 4.4 Detalles . . . 14
  - 4.4.1 Autoconf . . . 14
  - 4.4.2 Autoscan . . . 15
  - 4.4.3 Imake . . . 15
- 4.5 Inconvenientes . . . 15

5 Automatización. Automake **15**
- 5.1 Introducción . . . 15
- 5.2 Funcionamiento . . . 16
- 5.3 Ejemplos . . . 16
- 5.4 Detalles . . . 18

6 Visión de conjunto **18**
- 6.1 El ejemplo . . . 19
- 6.2 Creación del proyecto . . . 19

1 Introducción

En este documento se pretende describir algunas de las herramientas de desarrollo de programas y aplicaciones más usadas en Linux. Las herramientas que se describen son: el compilador GCC, algunas utilidades GNU binarias, el GNU Make, Autoconf y Automake. Todas estas herramientas están también disponibles para otros sistemas GNU (no Linux), e incluso la mayoría se pueden usar en otros entornos (como MS Windows).

No se describen entornos integrados de desarrollo (IDEs), herramientas visuales ni herramientas de desarrollo rápido de aplicaciones (RADs). Todas las utilidades descritas aquí funcionan desde la línea de comandos, aunque para algunas de ellas existen entornos gráficos que facilitan la tarea para los casos más comunes.

Las diferentes herramientas no son alternativas, sino complementarias. Un proyecto simple probablemente necesitará sólo de un compilador y un enlazador. A medida que crezca la complejidad del proyecto se irán incluyendo sucesivamente otras herramientas que faciliten la tarea del desarrollo y mantenimiento del programa. Es importante tener desde el principio una idea de la magnitud del proyecto y de las herramientas que serán necesarias para plantear adecuadamente la estructura del mismo.

Se ha intentado seguir un mismo esquema a la hora de describir todas las herramientas. Se incluye una descripción de la utilidad, que muestra su objetivo y aplicaciones. A continuación se muestra se uso más común (que en la gran mayoría de casos es todo lo que hace falta saber), seguido de un ejemplo concreto. Después se enumeran algunos detalles y opciones útiles, para acabar mencionando algunos de los problemas y dificultades que el uso de la herramienta puede originar (y cómo solucionarlos).

En todos los casos, el ejemplo propuesto es el mismo. Se trata del ubicuo programa Hello, cuya principal utilidad es la impresión por pantalla del mensaje "Hello, world!". En nuestro caso, el programa dispone de un interfaz gráfico y soporta opciones en la línea de comandos, para hacerlo más interesante. Además, puesto que se puede querer reutilizar las funciones del programa en otros proyectos, se incluye la posibilidad de compilar una biblioteca Libhello.

Ha de notarse que este documento no es una introducción a la programación en general, sino al uso de herramientas concretas. Se presuponen al lector ciertos conocimientos básicos de desarrollo de programas, tales como compilación, enlazado, bibliotecas,... Los términos "librería" y "biblioteca" se usan indistintamente a lo largo de todo el documento.

1.1 Portabilidad

A lo largo del documento se hace referencia a la portabilidad de los programas. Un programa es portable si puede transportarse de un sistema informático a otro con pocos o ningún cambio. Aquí nos interesará sobre todo la portabilidad a nivel de compilación. El objetivo de esta portabilidad es que el programa pueda compilarse sin cambios en sistemas diferentes. La portabilidad no sólo tiene la ventaja de que el campo de sistemas en los que el programa puede potencialmente ejecutarse aumente (aumentando la audiencia esperada del programa), sino que además normalmente la portabilidad es una garantía de calidad.

No se van a tratar aquí temas específicos de portabilidad a nivel de código fuente, sino sólo a nivel de herramientas de desarrollo. No obstante, es muy recomendable que los autores de software se preocupen por ambos aspectos de la portabilidad, ya que esto redundará en la mejor calidad de sus programas. Además, el uso de algunas de estas herramientas (como por ejemplo Autoconf) puede ayudar a conseguir portabilidad a nivel de código fuente.

1.2 Software libre

Todas las herramientas tratadas en este documento son software libre. Esto quiere decir que el código fuente de las mismas está disponible para que cualquiera pueda observarlo o modificarlo, si se quiere. Además, la redistribución de estas herramientas (tanto en forma ejecutable como fuente) está permitida (y recomendada).

Las herramientas de las que hablamos también pueden ser utilizadas para realizar software propietario (no libre). Sin embargo, en los últimos años, el movimiento de software libre ha conseguido demostrar que la calidad del software aumenta considerablemente cuando el código fuente es abierto y todos pueden notificar posibles fallos en el mismo.

La elección acerca de si un programa concreto se distribuye libremente o no corresponde únicamente a sus autores. En cualquier caso, esperamos que los potenciales autores de software consideren la posibilidad de distribuir sus programas como software libre después de observar la gran cantidad de utilidades libres descritas aquí (y su calidad).

2 Compilación y enlazado. GCC y compañía

2.1 Introducción

La principal aplicación del compilador GCC es la creación de ficheros ejecutables a partir de los fuentes en C. También puede usarse para crear librerías o para compilar lenguajes distintos a C (como C++).

Figura 1: Compilación y enlazado con GCC

En su uso más común, GCC actúa como compilador, ensamblador y enlazador, produciendo el resultado final con una sola orden. Esta utilización de GCC puede verse en la figura 1.

Por otra parte, es posible utilizar GCC sólo para la compilación y dejar que sea el enlazador (`ld`) el que se encargue de enlazar los módulos del programa. Este uso se muestra en la figura 2.

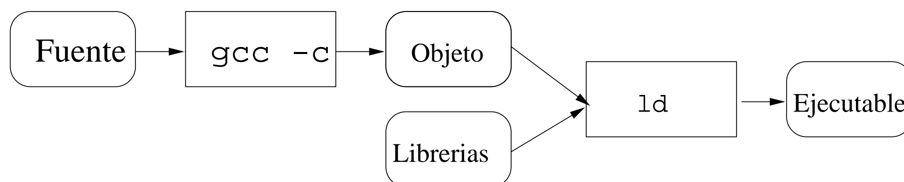

Figura 2: Compilación y enlazado separados con GCC

2.2 Herramientas disponibles

El paquete completo de herramientas GCC y binutils consta de las siguientes utilidades:

gcc El compilador. Genera fichero de código objeto a partir de los ficheros de código fuente. También puede realizar el proceso completo, llamando al ensamblador y el enlazador (éste es el comportamiento por defecto).

ld Enlazador. Crea el ejecutable o la librería a partir de los ficheros objeto y las librerías. Normalmente no se llama independientemente, sino a partir de GCC.

as Ensamblador. El ensamblador de GNU, usado por defecto por GCC, utiliza la sintaxis ATT, distinta de la sintaxis Intel, que es la más habitual fuera del mundo Unix. Existe otro ensamblador (**as86**), que usa la sintaxis Intel.

nm Lista los símbolos contenidos en un fichero objeto. Su principal utilidad es el listado de las funciones exportadas por una librería.

ar Archivador. Además de la función de archivo, aplicable a todo tipo de ficheros, **ar** sirve para la creación de bibliotecas estáticas. En Unix, una biblioteca estática no es más que un archivo que contiene sucesivamente todos los ficheros objeto que la forman. Estos archivos se pueden ver, extraer y modificar desde la línea de comandos usando **ar**.

strace Seguimiento de las llamadas el sistema. **strace** ejecuta el programa que se le pasa como argumento, mostrando todas las llamadas al sistema que realice, junto con los argumentos pasados y el valor devuelto. Es muy útil para conocer el funcionamiento de los programas y poder depurarlos.

gdb Depurador de binarios. Permite la ejecución paso a paso, el desensamblado, la monitorización de valores y muchas cosas más. Es conveniente usarlo a través de un interfaz gráfico.

2.3 Ejemplos

Nuestro programa de ejemplo consta de los ficheros fuente `hello.c` y `gui.c`, y el fichero de cabeceras `hello.h`. Puesto que usa funciones matemáticas[1], es necesario enlazarlo con la librería `libm.so`.

El interfaz gráfico utiliza funciones de XWindow, por lo que tendremos que incluir cabeceras `/usr/X11/include/X11/X11.h` y enlazar con la librería `/usr/X11/lib/libX11.so` (nótese que ambos están en localizaciones no estándar).

Compilación de programa

```
gcc -o hello hello.c gui.c -lm -lX11 -L/usr/X11/lib -I/usr/X11/include
```

`-o` Especifica el nombre del fichero de salida. Si no se indica, se pone por defecto el nombre `a.out`.

`-l` Enlazar con la librería indicada. Si se especifica `-l`*aaa* se enlaza con la librería dinámica `lib`*aaa*`.so`.

`-L` Directorio dónde buscar las librerías.

`-I` Directorio dónde buscar los ficheros de cabecera (*headers*).

[1] La impresión de la cadena de caracteres “Hello, world!” en esta implementación concreta necesita del cálculo de varios logaritmos decimales :-)

Compilación de librería

```
gcc -fPIC -c hello.c gui.c -I/usr/X11/include
gcc -shared -Wl,-soname,libhello.so.1 -o libhello.so.1.0.0 hello.o gui.o
ln -s libhello.so.1.0 libhello.so.1
ln -s libhello.so.1.  libhello.so
```

`-fPIC` Crea código independiente de la posición (ie, reposicionable), necesario en una librería dinámica.

`-shared` Produce un objeto compartido, capaz de ser enlazado con otros objetos para formar un ejecutable.

`-Wl,`***option*** Pasa la opción *option* al enlazador. Si la opción contiene comas, se divide en múltiples opciones al pasarla.

`-soname` Opción del enlazador. Asigna el campo `DT_SONAME` al valor especificado. Este campo se usa para asegurarse de que cada ejecutable se enlaza con la librería correcta, independientemente del nombre del fichero que la contenga. También asegura que la versión correcta de la librería es utilizada.

2.4 Detalles

Se comentan aquí algunas opciones normales en `gcc`, `ld` y demás programas.

`-o` En todos los casos, especifica el nombre del fichero de salida.

`-c` Esta opción indica a GCC que sólo debe compilar los ficheros fuente, no enlazarlos.

`-S` No compila. Sólo genera código ensamblador para cada fichero fuente.

`-E` Sólo ejecuta la etapa de preprocesamiento. No ejecuta el compilador propiamente dicho.

`-D` Define el símbolo de preprocesador indicado a continuación. Incluir `-D`*simb*`=`*val* en la línea de órdenes es equivalente a tener la línea `#define simb val` en el código fuente.

`-Wall` Activa todos los avisos de GCC. Muy útil para prevenir posibles errores difíciles de notar.

`-g` Añade información de depuración al fichero resultante. Necesaria para poder usar `gdb` sobre el fichero compilado.

`-v` Muestra los comandos ejecutados para realizar cada paso de la compilación (preprocesamiento, compilacón,...). También muestra la versión del programa.

La opción `-v` puede usarse para ver qué símbolos son definidos automáticamente por el compilador:

```
$echo 'main() { }' |gcc -E -v -
Reading specs from /usr/lib/gcc-lib/i486-linux/2.7.2.3/specs
gcc version 2.7.2.3
/usr/lib/gcc-lib/i486-linux/2.7.2.3/cpp -lang-c -v -undef
-D__GNUC__=2 -D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386
```

```
-Dlinux -D__ELF__ -D__unix__ -D__i386__ -D__linux__ -D__unix
-D__i386 -D__linux -Asystem(unix) -Asystem(posix) -Acpu(i386)
-Amachine(i386) -
GNU CPP version 2.7.2.3 (i386 Linux/ELF)
#include "..." search starts here:
#include <...> search starts here:
  /usr/local/include
  /usr/i486-linux/include
  /usr/lib/gcc-lib/i486-linux/2.7.2.3/include
  /usr/include
End of search list.
# 1 ""
main() { }
```

Los símbolos que aparecen en la anterior línea de comandos después de cada `-D` son definidos automáticamente por este compilador, y se pueden usar en el código fuente para comprobar si el programa se está compilando en un entorno Linux, Unix, para una arquitectura concreta, o con el compilador GNU.

2.5 Inconvenientes

Cuando se tiene un programa de cierta envergadura que consta de varios ficheros, no es práctico recompilar todo cada vez que se modifica un fichero. GNU Make se encarga de tener en cuenta las dependencias entre ficheros y recompilar y/o enlazar sólo cuando sea necesario.

3 Dependencias. GNU Make

3.1 Utilidad

Make permite gestionar adecuadamente un proyecto que conste de varios ficheros, llamando a las herramientas adecuadas que regeneren las partes del proyecto que lo necesiten.

Las dependencias de un fichero constan en una lista de ficheros que especifican qué ficheros son necesarios para su creación. Por ejemplo, el fichero `hello.o` depende de `hello.c` y de `hello.h`. Cuando alguno de estos dos es modificado, es necesario recompilar `hello.o` para volver a ponerlo en sincronismo con sus dependencias.

El programador debe especificar a Make cuáles son las dependencias de los ficheros mediante un fichero de texto, normalmente llamado `Makefile`, y que suele residir en el mismo directorio que el código fucnte. En este fichero también es especifica la forma de regenerar cada fichero cuando está fuera de sincronismo con sus dependencias. Una vez creado el Makefile, sólo hay que invocar a Make, y éste se encarga de chequear qué ficheros necesitan ser regenerados y hacerlo.

Este flujo de ficheros se muestra en la figura 3.

3.2 Ejemplos

Compilación de programa

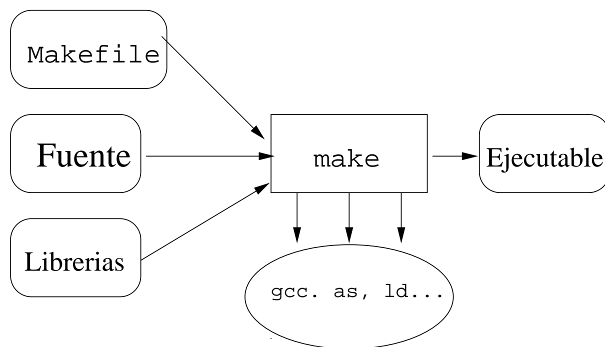

Figura 3: Uso de Make

```
1   # Makefile para hello

    OBJS= hello.o gui.o
    CFLAGS= -Wall -I/usr/X11/include
5   LIBS=-lm -lX11
    LDFLAGS= -L/usr/X11/lib
    CC= gcc

    all: hello
10  .c.o:
            $(CC) -c $(CFLAGS) $< -o $@
    hello: $(OBJS)
            $(CC) $(LIBS) $(LDFLAGS) -o $@ $(OBJS)
```

Un Makefile está formado principalmente por reglas, que son relaciones de dependencia e instrucciones para la regeneración de ficheros.

Las reglas se especifican en la forma:

```
objetivo: dependencias
        instrucciones
        ...
```

Es importante que las instrucciones estén precedidas por un tabulador, y no por espacios.

Si el objetivo es más nuevo que las dependencias especificadas, no se hace nada. En caso contrario, el fichero objetivo es regenerado usando las instrucciones. Naturalmente, las dependencias pueden tener a su vez otras dependencias. Make gestiona correctamente esto, construyendo los ficheros a partir de sus dependencias en el orden adecuado.

Las primeras líneas del Makefile de ejemplo son definiciones de macros. Una macro se define con `MACRO= sustitución`. Más tarde, se referencia en el fichero mediante `$(MACRO)`.

Dos macros importantes son `$<` y `$@`. La primera se sustituye por el nombre de las dependencias usadas en la regla actual. La segunda se sustituye por el nombre del fichero destino de la regla.

A veces se utilizan objetivos que no corresponden a ficheros reales (destinos *phony*, postizos). Ejemplos comunes son `all`, `clean`, `install`,... que realizan acciones concretas diferentes de la simple generación de un fichero.

Normalmente `all` se refiere a la reconstrucción completa de todo los objetivos del programa, `clean` especifica que se deben limpiar los ficheros no necesarios (como ficheros intermedios, de código objeto. etc.), e `install` indica a Make que copie los destinos resultantes de la compilación (ejecutables, librerías, documentación, etc.) en las localizaciones adecuadas para su uso en el sistema. Naturalmente, el autor del `Makefile` debe escribir las reglas correspondientes a estos objetivos postizos para que cumplan su función.

El objetivo `.c.o` se usa para referirsa a que todos los ficheros acabados en `.o` se construyen a partir de los equivalentes acabados en `.c` mediante las instrucciones indicadas.

La compilación se ha hecho en dos etapas para mostrar el uso de `.c.o`, pero se podría realizar igualmente un `Makefile` que compile y enlace en una sola línea.

3.3 Detalles

Makefiles

- El carácter `#` al principio de una línea la marca como comentario.
- Una línea terminada con \ continúa en la línea siguiente.
- La directiva `include` permite incluir el contenido de un fichero dentro del Makefile.
- Al invocar a Make, éste busca, por orden, un fichero llamado `GNUmakefile`, `makefile` y `Makefile`. El nombre recomendado es éste último. Se puede especificar otro nombre con `make -f fichero`.

Reglas

- El orden de las reglas no tiene importancia, excepto para determinar el objetivo por defecto, que es el objetivo que Make intentará construir si no se especifica ninguno. Éste es el primer objetivo del fichero.
- Se puede especificar el objetivo en la línea de comandos, por ejemplo `make all` o `make install`.
- Se puede especificar que un objetivo es postizo incluyendo una línea de la forma `.PHONY : objetivo`.
- Si un objetivo no tiene dependencias ni instrucciones, y este objetivo es un fichero que no existe, entonces Make supone que este objetivo se actualiza al ejecutar sus instrucciones (inexistentes). Esto implica que todos los objetivos que dependan de éste se ejecutarán siempre. Por ejemplo:

  ```
  clean: FORCE
          rm $(objects)
  FORCE:
  ```

  En este Makefile, las instrucciones de `clean` se ejecutarán siempre. El nombre `FORCE` no es especial, pero es costumbre denominar así a este tipo de objetivos.

Instrucciones

- Las instrucciones se ejecutan a través de `/bin/sh`, a no ser que se especifique lo contrario. Para especificar otra shell, se usa la variable `SHELL`. Cada línea de un conjunto de instrucciones se ejecuta en una nueva instancia de la shell (esto implica que comandos como `cd` no afectan a las siguientes líneas).
- Cuando una instrucción comienza por `@`, no se muestra por pantalla.

Variables automáticas

`$@`	Nombre del fichero objetivo de la regla actual
`$<`	Nombre de la primera dependencia
`$?`	Nombres de todas las dependencias que son más recientes que el objetivo
`$^`	Nombres de todas las dependencias, separadas por espacios.

3.4 Inconvenientes

Aunque Make facilita en gran medida el trabajo de compilación, no hace nada por la portabilidad. La única manera en que Make soporta la diversidad de entornos y plataformas es mediante el uso inteligente de variables.

Por ejemplo, unos sistemas llaman al compilador de C `cc`, otros `gcc`, otros `pgcc`, etc. Es normal incluir una variable `CC` al principio del Makefile que contenga el nombre con que llamar al compilador. Lo mismo se hace con `CFLAGS`, `LIBS`, etc., que pueden ser diferentes de sistema a sistema.

Lo habitual es que el usuario mofifique a mano el Makefile para adaptarlo a su sistema particular. En el mejor de los casos se incluyen ya algunos Makefiles listos para su uso en un sistema concreto, como `Makefile.linux`, `Makefile.sun`, `Makefile.win32`...

En cualquier caso, esta solución sólo puede servir para proporcionar portabilidad en el momento de la compilación, pero no en el de escribir el código fuente. Es normal que diferentes variantes de Unix tengan implementaciones ligeramente diferentes de funciones comunes (como los Sockets, las funciones de identificación, de bases de datos, etc.). A la hora de escribir programas en C, esto se tiene en cuenta de una manera tediosa usando `#define`'s y `#ifdef`'s.

Existe una mejor solución: el uso de un fichero ejecutable que detecte la configuración concreta del sistema en el que se está ejecutando y sea capaz de generar un Makefile adecuado, así como un fichero de cabecera para incluir en el código fuente C. Entonces es este fichero el que se incluye en la distribución del programa en lugar del Makefile. Veremos más sobre esto en las secciones 4 y 5.

4 Configuración automática. Autoconf

4.1 Introducción

En la sección 3.4 se ha comentado la utilidad de un programa que fuese capaz de determinar la configuración concreta de un sistema entre todas las alternativas posibles y que escribiese un Makefile y un fichero de cabeceras adecuados al programa que queremos compilar.

Puesto que las comprobaciones necesarias dependen mucho del tipo de programa a compilar no es factible tener un ejecutable general válido para todos los programas. En su lugar,

cada distribución en código fuente debe contener un ejecutable que compruebe el sistema en que se encuentra y genere los ficheros necesarios para la compilación.

La dificultad de crear un programa así es evidente. Por eso existe la utilidad Autoconf. Esta utilidad es capaz de generar ese programa de configuración para cada distribución de código fuente.

La entrada al programa Autoconf es una lista de comprobaciones necesarias para que el programa que se quiere compilar funcione en sistemas distintos entre sí. Su salida es otro programa que realizará las comporbaciones indicadas y generará los ficheros que se usarán para la compilación.

4.2 Funcionamiento

Autoconf genera como salida un programa ejecutable llamado `configure`. La función de este programa depende de lo que le hayamos indicado a Autoconf, pero en general chequeará el sistema en que se esté ejecutando, haciendo varias comprobaciones y al final escribirá los resultados de esas comprobaciones en uno o más ficheros predeterminados.

El funcionamiento de `configure` se especifica en un fichero llamado `configure.in`. Este fichero se procesa con Autoconf y da como resultado un `configure`. Cuando éste se ejecute leerá (entre otros) un `Makefile.in` y un `config.h.in`, a partir de los cuales creará un `Makefile` y un `config.h`.

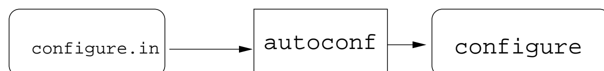

Figura 4: Preparación de un paquete para su distribución

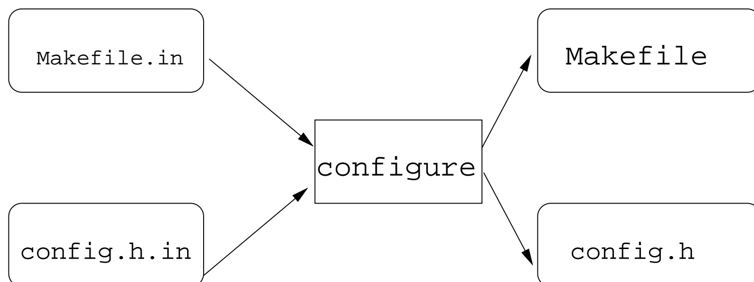

Figura 5: Configuración de un paquete

Estos flujos de ficheros se ven en la figura 4 desde el punto de vista del programador, y en la figura 5 desde el punto de vista del usuario.

En el paquete preparado para la distribución (el paquete que llegará al usuario final) se incluyen los ficheros `configure`, `Makefile.in` y `config.h.in`, pero no es necesario incluir `configure.in`.

Tampoco es necesario que el usuario final tenga Autoconf instalado en su sistema. Autoconf sólo es usado por el desarrollador, y para la compilación final del programa sólo hacen falta las herramientas vistas antes (GCC y Make).

4.2.1 `configure.in`

El fichero `configure.in` indica qué comprobaciones deben ir en el `configure`.

Existe un programa llamado `autoscan` que es capaz de crear ficheros `configure.in` simples a partir del código fuente. Ver la sección 4.4.2 para más información.

Un `configure.in` está formado por llamadas a diferentes macros. Cada uno debe contener una llamada a `AC_INIT` antes de las comprobaciones y otra a `AC_OUTPUT` después de ellas.

La estructura recomendada para un `configure.in` es la siguiente:

`AC_INIT`{fichero}
chequeo de programas
chequeo de librerías
chequeo de ficheros de cabecera
chequeo de `typedef`'s
chequeo de estructuras
chequeo de particularidades del compilador
chequeo de funciones de biblioteca
chequeo de servicios del sistema
`AC_OUTPUT`{fichero,...}

Cada macro debe ir en una línea. Las líneas de comentario empiezan por `dnl` (`configure.in` es preprocesado por m4).

Además, cada directorio o subsirectorio en una distribución debe contener un fichero `Makefile.in`, a partir del cual `configure` creará un Makefile válido. Para lograr esto, `configure` realiza una sustitución de cada variable `@VARIABLE@` que se encuentre en `Makefile.in` con el valor que se ha determinado para esa variable.

4.3 Ejemplos

Los ficheros necesarios para la construccíón del programa Hello son:

Fichero `configure.in`

```
 1  dnl Process this file with autoconf to produce a configure script.
    AC_INIT(hello.c)

    dnl Checks for programs.
 5  AC_PROG_CC
    AC_SUBST(CC)
    AC_PROG_INSTALL

    dnl Checks for libraries.
10  AC_CHECK_LIB(m, log)
    AC_CHECK_LIB(X11,XOpenDisplay)
    AC_SUBST(LIBS)

    dnl Checks for header files.
15  AC_HEADER_STDC
    AC_CHECK_HEADERS(X11/X11.h)
```

```
dnl Checks for typedefs, structures, and compiler characteristics.
AC_C_CONST
20
dnl Checks for library functions.
AC_CHECK_FUNCS(XOpenDisplay,,break)

dnl Checks for programs
25
AC_OUTPUT(Makefile)
```

Las líneas 5-6 indican que se debe buscar el compilador de C y guardar su nombre en la variable `CC`. Si no se especifica `AC_SUBST`, la variable no se sustituye en los ficheros pasados a `AC_OUTPUT`.

Las líneas 10-12 se aseguran de que están disponibles las librerías `libm` y `libX11` y de que contienen las funciones `log()` y `XOpenDisplay()`. Normalmente no es necesario chequear cada función que se usa en el programa; basta con una para asegurarse de que la librería es correcta.

La macro `AC_HEADER_STDC` comprueba que el sistema contiene las cabeceras definidas en el estándar ANSI C.

La línea 16 busca el fichero de cabecera `<X11/X11.h>` y si lo encuentra define la variable `HAVE_X11`.

La macro `AC_C_CONST` determina si el compilador de C soporta completamente la palabra clave `const`.

En la línea 22 se busca la función `XOpenDisplay()`. Si se encuentra, se define la variable `HAVE_XOPENDISPLAY` y se ejecuta la acción detrás de la primera coma (en este caso, nada). Si no se encuentra se ejecuta la ultima acción.

Por último, la línea `AC_OUTPUT(Makefile)` indica que se debe crear un fichero `Makefile` a partir de otro llamado `Makefile.in` sustituyendo las apariciones de las variables antes especificadas.

Se podría incluir una macro `AC_CONFIG_HEADER(`header`)` para crear un fichero de cabecera que contuviese los `#define`'s.

Fichero `Makefile.in`

```
1   # Makefile para hello

    VPATH = @srcdir@
    srcdir = @srcdir@
5
    OBJS= hello.o gui.o
    CFLAGS= -Wall
    LIBS= @LIBS@
    LDFLAGS= @LDFLAGS@
10  CC= @CC@
    DEF= @DEFS@

    all: hello
    .c.o:
```

```
15          $(CC) -c $(CFLAGS) $< -o $@
    hello: $(OBJS)
            $(CC) $(LIBS) $(LDFLAGS) -o $@ $(OBJS)
```

Como puede observarse, este Makefile es prácticamente igual al mostrado en la sección 3.2, pero se han cambiado las definiciones de variables de Make para incluir las sustituciones de Autoconf.

4.4 Detalles

4.4.1 Autoconf

Comprobaciones

- `AC_PROG_CPP` Preprocesador de C. Variable `CPP`.
- `AC_PROG_CXX` Compilador de C++. Variable `CXX`.
- `AC_PROG_INSTALL` Programa de instalación. Variable `INSTALL`.
- `AC_PROG_LN` Funcionamiento de enlaces simbólicos. Variable `LN_S`.
- `AC_FUNC_MEMCMP` Si la función `memcmp()` no está disponible, aade `memcmp.o` a la variable `LIBOBJS`.
- `AC_FUNC_MMAP` Si la función `mmap` existe, define `HAVE_MMAP`.
- `AC_HEADER_DIRENT` Busca un fichero que defina `DIR` y pone la variable `HAVE_fichero_H` adecuada.
- `AC_MEMORY_H` Define `NEED_MEMORY_H` si `memcpy`, `memcmp`, etc no están declaradas en `<string.h>` y existe `memory.h`.
- `AC_UNISTD_H` Define `HAVE_UNISTD_H` si el sistema tiene este fichero.
- `AC_INT_16_BITS` Define `INT_16_BITS` si el tipo `int` tiene 16 bits.
- `AC_SYS_INTERPRETERS` Comprueba si el sistema admite ficheros ejecutables interpretados al estilo "`#!`".
- `AC_PATH_X` Trata de localizar el sistema XWindow, buscando su directorio raíz.
- `AC_SYS_LONG_FILE_NAMES` Define `HAVE_LONG_FILE_NAMES` si el sistema soporta nombres de ficheros mayores de 14 caracteres.

También es posible escribir nuevas comprobaciones, diferentes de las proporcionadas por Autoconf. De todas formas, las proporcionadas por defecto suelen ser lo bastante generales como para no tener que escribir ninguna nueva en la mayoría de los casos.

Si se escribe una función nueva, es necesario distribuirla junto con el resto del paquete, ya que los usuarios que quieran instalar el programa no dispondrán de ella en su sistema. Esto se hace incluyendo un fichero llamado `aclocal.m4` que contiene todas las macros no estándar que va a necesitar `configure`.

La escritura de nuevas macros de Autoconf es un tema que se sale del alcance de este documento. No es demasiado complicado hacerlo, y el lector interesado puede encontrar información útil en [4].

4.4.2 Autoscan

El programa Autoscan puede ser de ayuda para crear un fichero `configure.in`. Autoscan examina los ficheros de código fuente en el árbol de directorios y detecta las posibles incompatibilidades que podrían hacer al fichero no portable.

Entonces crea un fichero preliminar `configure.scan`. Este fichero debe ser revisado a mano para pulir algunos detalles, y entonces ser renombrado a `configure.in`.

Autoscan necesita tener Perl instalado.

4.4.3 Imake

Autoconf realiza un trabajo bastante similar a Imake. Sin embargo, el uso de Autoconf es el recomendado. Imake se basa en una base de datos de configuraciones estándar de sistemas para determinar cómo compilar una distribución. En cambio, Autoconf es capaz de hacer un buen trabajo en un sistema desconocido o no estándar.

4.5 Inconvenientes

Con el uso de Autoconf, el desarrollo portable de una aplicación es mucho más asequible. Es por este que Autoconf se ha convertido en una herramienta casi estándar, y prácticamente todas las distribuciones de programas en código fuente vienen con un fichero `configure`.

Uno de los problemas de Autoconf es su excesiva orientación hacia el mundo Unix [2], haciendo difícil la portabilidad a otros sistemas, como VMS o Windows. No obstante, en un sistema Windows dotado con herramientas Cygwin, Autonconf produce unos resultados más que aceptables, pudiéndose llegar en muchos casos a la portabilidad total.

Otro inconveniente es la dificultad de generar los `Makefile.in`'s en caso de proyectos grandes con muchos ficheros de código fuente. Para solucionar este problema existe la utilidad Automake, que trataremos en la sección 5.

5 Automatización. Automake

5.1 Introducción

Automake es capaz de generar automáticamente ficheros `Makefile.in` a partir de ficheros llamados `Makefile.am`. Cada `Makefile.am` es básicamente una serie de macros de Make (y algunas reglas). Los ficheros generados cumplen con los estándares GNU sobre Makefiles.

Este cumplimiento con el estándar es la principal ventaja de Automake. Los estándares GNU son largos y complicados, y sin el uso de una herramienta automática es tremendamente tedioso cumplirlos en su totalidad.

Automake necesita Perl para generar los `Makefile.in`, pero una vez creados, Perl no es necesario para la distribución del paquete. Por supuesto, Automake es una herramienta para el programador, y no para el usuario. No es necesario distribuir el `Makefile.am` ni que el usuario final tenga instalado Automake.

El diagrama de flujo del proceso de Automake se muestra en la figura 6.

[2] Mucha gente opinará que esto no es un problema sino una ventaja.

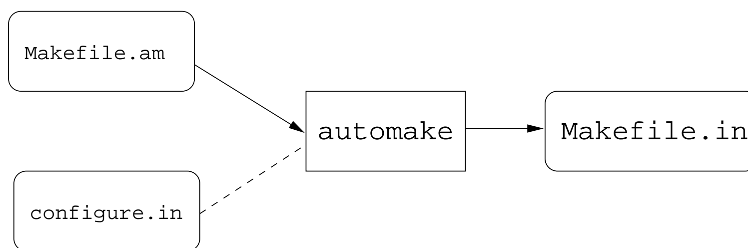

Figura 6: Funcionamiento de Automake

5.2 Funcionamiento

Automake trabaja con tres clases de estructuras de código fuente: planas, poco profundas y profundas.

Una estructura plana (*flat*) es aquella en la que todos os ficheros residen en un solo directorio. Es la estructura más común para programas sencillos.

Una estructura profunda (*deep*) tiene todo el código fuente en subdirectorios por debajo del directorio principal. Puede haber un subdirectorio para los programas, otro para las librerías, otro para la documentación, etc. El directorio principal contiene la información de la configuración (como el fichero `Makefile.in` y los demás).

Un paquete con estructura poco profunda (*shallow*) tiene el código fuente principal en el directorio inicial, y algunas partes (típicamente librerías) en subdirectorios.

Automake trata diferentemente estas clases de paquetes, y es conveniente organizar el código de manera que se ajuste a alguna de ellas.

Las variables en Automake siguen una convención de nombres que hace sencillo determinar cómo se construyen los programas.

Las variables primarias indican qué objetos (por ejemplo programas) van a ser compilados o construidos. Las variables adicionales indican cómo construir y dónde instalar esos objetos. Estas variables adicionales se nombran añadiendo un prefijo a las variables primarias.

Por ejemplo, la variable `PROGRAMS` contiene una lista de programas que se van a construir (es primaria). A partir de ella se derivan variables que indican dónde instalar esos programas. La variable `bin_PROGRAMS` contiene la lista de los programas que se instalarán en el directorio de binarios, y `lib_PROGRAMS` la lista de los que se instalarán en el directorio de librerías.

Por cada programa contenido en estas variables hay un conjunto de variables que especifican cómo construir esos programas. Por ejemplo, para el programa `hello` existen las variables `hello_SOURCES`, que contiene la lista de los ficheros fuente de Hello, `hello_MANS`, que incluye la lista de páginas de manual, etc.

5.3 Ejemplos

El fichero `configure.in` de Hello necesita ser modificado para dar cabida a Automake:

```
1   dnl Process this file with autoconf to produce a configure script.
    AC_INIT(hello.c)
    AM_INIT_AUTOMAKE(hello, 1.0.0)
    AM_CONFIG_HEADER(config.h)
5
```

```
    dnl Checks for programs.
    AC_PROG_CC
    AC_SUBST(CC)
    AC_PROG_INSTALL
10
    dnl Checks for libraries.
    AC_CHECK_LIB(m, log)
    AC_CHECK_LIB(X11,XOpenDisplay)
    AC_SUBST(LIBS)
15
    dnl Checks for header files.
    AC_HEADER_STDC
    AC_CHECK_HEADERS(X11/X11.h)

20  dnl Checks for typedefs, structures, and compiler characteristics.
    AC_C_CONST

    dnl Checks for library functions.
    AC_CHECK_FUNCS(XOpenDisplay,,break)
25
    dnl Checks for programs

    AC_OUTPUT(Makefile)
```

Las macros AM_ se proporcionan con Automake.

La línea 3 inicializa Automake. Esta línea es necesaria en cada `configure.in` que vaya a ser usado con Automake, y contiene el nombre del paquete y su versión.

La línea 4 indica a Automake cuál es el fichero de cabecera dónde `configure` escribirá la información de configuración.

El resto de líneas son de Autoconf.

Por otro lado, tenemos un `Makefile.am` como el siguiente:

```
1   bin_PROGRAMS = hello
    hello_SOURCES = hello.c gui.c
```

Lo único que hay que especificar en un `Makefile.am` como este (plano) es la lista de los programas a compilar, y los ficheros de código fuente de cada uno.

A continuación se incluye un ejemplo de ficheros `Makefile.am` para un paquete con una estructura profunda.

El `Makefile.am` del directorio principal sólo contiene:

```
1   EXTRA_DIST = LEEME
    SUBDIRS = doc src lib
```

La primera línea es opcional y consiste en una lista de ficheros que serán incluidos en la distribución a pesar de no ser código fuente ni parte de una distribución estándar (como este fichero `LEEME`). Los ficheros `README`, `INSTALL`, `ChangeLog`, etc. sí forman parte de una distribución estándar.

La segunda línea lista los subdirectorios de los que consta el paquete. Cada uno de estos subdirectorios debe contener su propio fichero `Makefile.am` indicando cómo construir el

programa.

Por ejemplo, el Makefile.am del subdirectorio src podría contener:

```
1  bin_PROGRAMS = hello
   hello_SOURCES = hello.c gui.c
   hello_LDADD = @LIBS@ @LDADD@
```

Este fichero especifica cuáles son los ficheros aprtir de los que hay que construir el ejecutable, y algunas de las opciones a añadir en la línea de órdenes.

5.4 Detalles

Al ejecutar Automake se puede especificar la opcion --add-missing (o su equivalente -a), y Automake se encargará de generar todos los ficheros estándar en una distribución y que no estén ya presentes (como README, INSTALL, . . .).

Macros Automake proporciona las siguientes macros para el fichero configure.in (entre otras):

- AM_CONFIG_HEADER Hace que Automake genere reglas para regenerar el fichero de cabecera de configuración (config.h).
- AM_CYGWIN32 Chequea la ejecución en un entorno Cygwin, bajo Win32. Si lo detecta, genera Makefile.in's que funcionen correctamente en este entorno.
- AM_INIT_AUTOMAKE Ejecuta varias macros necesarias para la inicialización. Requiere como argumentos el nombre del paquete y el número de versión del mismo.
- AM_PROG_CC_STDC Añade a la línea de órdenes del compilador de C las opciones necesarias para que se ejecute en modo ANSI C.
- AM_GNU_GETTEXT Macro requerida para paquetes que usen el sistema de internacionalización Gettext.
- AM_PROG_LIBTOOL Habilita el procesamiento por parte de la herramienta Libtool [3].

Al igual que sucedía con Autoconf, Automake también puede extender su funcionalidad mediante la escritura de nuevas macros. Por ejemplo, el programa Gettext incorpora la macro AM_GNU_GETTEXT para su uso conjunto con Automake.

6 Visión de conjunto

Se ha ehcho una descripción de cada una de las herramientas por separado, comentando cuál es su función y dónde encajan en el proceso general de creación de un programa.

A continuación se va a describir el funcionamiento conjunto de todas ellas, que será el más habitual.

El diseño de las herramientas propicia que prácticamente sólo sea necesario utilizar las de más alto nivel, y que éstas se encarguen de gestionar el funcionamiento de las de nivel

[3] Libtool es una herramienta para facilitar la generación de librerías dinámicas y estáticas que no se describe en este manual. Para más información, ver [7].

más bajo. De todas formas, es bueno conocer todas las herramientas que forman parte del proceso para tener un dominio completo sobre las operaciones que se realizan.

6.1 El ejemplo

Para describir el funcionamiento conjunto de las herramientas se van a describir los pasos que habría que seguir para crear la estructura de directorios, los ficheros de configuración, etc. de un proyecto de software.

Con el objeto de ser lo más general posible, se ha elegido un proyecto que contenga casi todos los elementos que se esperan en un programa moderno y estándar.

La estructura de directorios del proyecto será profunda, por ser la más flexible. Uno de los directorios contendrá la documentación del programa.

Se va a incluir soporte para la internacionalización del programa usando GNU Gettext. Gettext es una herramienta que se usa para añadir a los programas la capacidad de adaptarse al entorno geográfico en el que se ejecutan y mostrar su salida en el idioma local. Gettext requiere en el código fuente de dos directorios: el directorio `intl` contiene el código fuente de Gettext en sí mismo, y es necesario por si el usuario final no tiene Gettext instalado en su sistema; y el directorio `po` contiene los catálogos de mensajes, que son las traducciones a diferentes idiomas de los mensajes del programa. Para más información sobre Gettext, se puede ver la referencia [8].

La distribución del programa incluirá todos los ficheros que se esperan en una distribución de código fuente, como `README`, `AUTHORS`, `INSTALL`, `NEWS`, ...

6.2 Creación del proyecto

Para crear el proyecto es necesario seguir los siguientes pasos:

1. Crear un directorio que será el directorio base del programa. El estándar especifica que este directorio debe llamarse *programa-versión*, por ejemplo `hello-1.2.1`.
2. Crear los subdirectorios que incluirán el código y la documentación: `src` y `doc`.
3. Escribir el código y la documentación.
4. Crear un `configure.in` en el directorio raíz. Se puede hacer siguiendo las indicaciones dadas en la sección 4, o bien generarlo con Autoscan. Puesto que en este caso este fichero contendrá macros no estándar (como `AM_GNU_GETTEXT`, necesaria para Gettext) habrá que ejecutar `aclocal` para que cree el fichero `aclocal.m4` con las definiciones de estas macros, y louego renombrar `aclocal.m4` como `acinclude.m4`.
5. Crear un `Makefile.am` en el directorio raíz. En él incluir algo como

   ```
   SUBDIRS = src intl po doc
   ```

6. Crear un `Makefile.am` en los directorios `src` y `doc` (y en otros si los hay), especificando los objetivos y la forma de generarlos.

   Por ejemplo, el `Makefile.am` del directorio `doc` puede contener:

   ```
   EXTRA_DIST = hello_documentos.txt
   ```

7. Ejecutar `gettexttize` para que cree la estructura de directorios necesaria para Gettext (subdirectorios `intl` y `po`). Si el programa no usase Gettext este paso no sería necesario.

8. Crear un fichero `config.h.in` mediante la ejecución de `autoheader`. Si en `configure.in` no existe la macro `AC_CONFIG_HEADER` no hay que realizar este paso.

9. Ejecutar Automake para que cree el `Makefile.in` (añadir la opción `-a` para que genere los ficheros que falten).

10. Ejecutar Autoconf para que se cree el `configure`.

En este punto el programa está listo para su distribución. Si se hace

```
configure
make dist
```

se creará un fichero tar con la distribución del programa.

De entre todos los ficheros tratados aquí, los que deben ir en la distribución para el usuario final son:

- Todo el código fuente, documentación, y cualquier otro fichero que constituya el programa en sí mismo.
- `Makefile.in`
- `configure`
- `ABOUT-NLS`, `AUTHORS`, `COPYING`, `ChangeLog`, `INSTALL`, `NEWS` y `README`.
- `aclocal.m4`
- Ficheros necesarios para la instalación: `install-sh`, `missing`, y `mkinstalldirs`.

Los demás ficheros no son necesarios para la instalación por parte de un usuario final, pero se pueden incluir en la distribución para posibles desarrolladores que quieran modificar o contribuir al programa.

Referencias

[1] Eric S. Raymond, *Software Release Practice HOWTO*, Noviembre 1998.

[2] Daniel Barlow, *The Linux GCC HOWTO*, Febrero 1996.

[3] Richard M. Stallman, Roland McGrath, *GNU Make Manual*, Agosto 1997.

[4] Ben Pfaff, *Autoconf Manual*, Noviembre 1996.

[5] David McKenzie, Tom Tromey, *Automake Manual*, Abril 1998.

[6] Richard M. Stallman, *Using and Porting GNU CC*.

[7] Free Software Foundation, *The Libtool Manual*.

[8] Ulrich Drepper, *GNU Gettext*, Abril 1995.