Make

1. Uso de Make

Al escribir un programa largo, intentaremos organizar el código, para facilitar el desarrollo y futuro mantenimiento del mismo. Organizar un programa implica dividir el código en procedimiento con nombres descriptivos; a veces esto implica también dividirlo en varios archivos. Crear un archivo ejecutable de varios archivos fuentes consiste en compilar cada archivo fuente en un archivo objeto de código de máquina. Los objetos se ligan en el binario, con una biblioteca estándar de código pre-compilado.

Make es un utilitario que permite expresar fácilmente la relación entre los archivos que componen un programa para guiar el desarrollo de un programa.

2. El Makefile

La guía de Make es un archivo de texto que escribe el programador llamado Makefile o makefile. Make busca este archivo en el directorio en el que se encuentra. El makefile se divide en dos partes, las variables y las dependencias.

2.1 Dependencias

En la sección de dependencias es donde se muestra la relación entre los archivos. Puede decirse que un makefile muestra una "estructura de árbol de dependencias", en el que expresa la relación entre los archivos que componen un programa.

Veamos un ejemplo genérico:

```
mi_programa: parte1.o parte2.o
  cc -o mi_programa parte1.o parte2.o

parte1.o: parte1.c
  cc -c parte1.c

parte2.o: parte2.c
  cc -c parte2.c
```

En la columna de la derecha hay una serie de labels (etiquetas). Los labels son, en general, nombres de archivos. Al lado de los labels hay una serie de dependencias, esto es, los archivos con los que se construye el archivo del label. Los archivos dependientes tienen sus propias etiquetas. Los líneas identadas bajo los labels son las acciones que se ejecutarán teniendo en cuenta las dependecias entre archivos. Es muy importante tener en cuenta que la identación debe ser una tabulación, y no la cantidad correspondiente de espacios.

Para concluir que archivos deben recompilarse, make se basa en el momento de creación del archivo dependiente. Si los archivos independientes son más recientes que el archivo que de ellos depende, se recompilará ya que han cambiado los archivos fuente o las bibliotecas.

2.2 Variables

Aquí tenemos el mismo makefile genérico con algunas variables agregadas:

```
# Makefile ejemplo

CC= cc
OBJETOS= parte1.o parte2.o
CFLAGS= -g
#CFLAGS= -O
```

```
CABECERAS= constantes.h globales.h

mi_programa: $ (OBJETOS)
  $ (CC) $ (CFLAGS) -o mi_programa $ (OBJETOS)

parte1.o: parte1.c $ (CABECERAS)
  $ (CC) $ (CFLAGS) -c parte1.c

parte2.o: parte2.c $ (CABECERAS)
   $ (CC) $ (CFLAGS) -c parte2.c
```

La primera línea comienza con el símbolo "#", que significa que es un comentario. Los comentarios pueden agregarse en cualquier parte del makefile. Cuando make utilice el makefile, ignorará las líneas con comentarios, como si fuesen líneas en blanco. Las variables se encuentran en las líneas 3 a 7, y deben ubicarse antes de comenzar con las dependencias.

Las variables pueden tener cualquier nombre. Para hacer referencia a una variable llamada VARIABLE se utiliza la expresión $ (VARIABLE). Cuando se encuentra una variable, se reemplaza en ese lugar por el texto de la variable.

Las variables son útiles ya que permiten evitar errores cuando un mismo texto se repite varias veces en el makefile ya que, de modificarse, se modifica una sola vez.

3. Uso de archivos cabecera

En un programa C hay dos formas de incluir un archivo cabecera. Utilizando comillas o entre los símbolos menor y mayor:

```
#include<stdio.h>
#include"globales.h"
```

Los símbolos <> indican que el archivo se encuentra en el directorio de búsqueda por defecto, en Linux, /usr/include. Para los archivos cabecera que se indican entre comillas debe indicar el path completo donde deben buscarse. En el ejemplo, el archivo cabecera "globales.h" se buscará en el directorio actual.

Para que el código sea más ordenado, conviene que todos los archivos cabecera creados por el programador se encuentren en un mismo directorio separado del directorio del programa. Esto implicará indicar la ruta completa dentro de la sentencia include en C. Pero en lugar de usar una línea como la siguiente:

```
#include"/trabajos_practicos/programa1/headers/mi_header.h"
```

Podemos usar:

```
#include <mi_header.h>
```

Esto es posible agregando una variable al makefile:

```
DIRINCL= -I/ trabajos_practicos/programa1/headers
```

Entonces, esta variable puede usarse en la acción para la compilación del programa:

```
$ (CC) $ (CFLAGS) -o parte1.c $ (DIRINCL)
```

4. Uso de Bibliotecas

Para el compilador, la opción -lnombrebiblioteca indica una biblioteca. En general, las bibliotecas tienen un formato de nombre estándar de forma lib[nombre].a. Por defecto, las bibliotecas se buscan en /usr/lib.

En general, las opciones del compilador pueden ir en cualquier orden, mientras se encuentren en pares que concuerden entre sí (par opción nombre, ejemplo -o mi_programa). Sin embargo, algunos sistemas requieren que la opción de biblioteca se encuentre al final de la línea de compilación. Es recomendable colocar esta opción siempre al final, ya que no se indica error en los sistemas que así lo requieren.

Además, debe tenerse en cuenta que las bibliotecas deben linkearse una sola vez en el programa, así que no deben linkearse a los archivos objeto.

4.1 Uso de las bibliotecas en un makefile

Generalmente, se utilizan dos variables para utilizar bibliotecas en un makefile, por ejemplo:

```
BIBLIOTECAS = -lm -lX11 -lmi_biblio
DIRBIBLIO = -L.
```

Listar las bibliotecas en una variable limpia la línea de compilación en el makefile y además permite agregar fácilmente nuevas bibliotecas. Tener una variable como DIRBIBLIO con la opción -L permite colocar como argumento otro directorio para el path de búsqueda de bibliotecas. El directorio "." representa en directorio actual. Si se usa junto a la opción "-lmi_biblio", el linker va a buscar el archivo libmi_biblio.a en el directorio actual. Así se vería el comando completo dentro del makefile:

```
$ (CC) $ (CFLAGS) -o mi_programa $ (OBJETOS) $ (DIRBIBLIO) $ (BIBLIOTECA)
```

5. Uso de makefile para otras acciones

Por defecto, al usar make se toma como programa ejecutable al primer label. Pero este primer label no tiene por qué ser el ejecutable, puede ser cualquier otro label dentro del makefile. De hecho, se puede crear más de un ejecutable con el mismo makefile. Esto se arregla especificando el label del ejecutable cuando se ejecuta make.

Esto es sumamente útil cuando se tiene un programa que incluye códigos de otros programas, por ejemplo:

```
final: programa1 programa2

programa1: $ (OBJETOSP1)
  $ (CC) $ (CFLAGS) -o programa1 $ (OBJETOSP1)

programa2: $ (OBJETOSP2)
  $ (CC) $ (CFLAGS) -o programa2 $ (OBJETOSP2)
```

Este makefile permite compilar tanto un programa individual utilizando make programa1 o todo el programa utilizando make o make final.

Los makefiles también suelen incluir acciones para mantener el programa. Una de las más comunes es incluir un label clean que permite borrar aquellos archivos que ya no van a utilizarse, por ejemplo:

```
clean:

  cp mi_programa mi_programa.backup
  rm -f *.o *.a mi_programa core
```

En este ejemplo, al ejecutar make clean, se crea una backup del ejecutable, y luego se borran todos los archivos que no sean fuente, que resulta una operación útil antes de recompilar el programa.

6. Características útiles

Continuación de línea: para adecuarse al estándar UNIX para los archivos de configuración, cuando se cambia de línea puede indicarse con una barra invertida.

Variables Anidadas: las variables pueden utilizarse dentro de otras variables:

```
CABPATH = ..
INCLUIDOS = -I$ (CABPATH)/otros/h \
  -I$ (CABPATH)/otros/h/red \
  -I$ (CABPATH)/otros/h/hdr
CFLAGS = -O $ (INCLUDES)
```

7. Makefiles eficientes

Dado que make deduce algunos comandos por sí mismo, no es necesario escribir un makefile tan explícito como los que se muestran en este apunte. Por ejemplo, el siguiente makefile va a ejecutar las mismas acciones que el ejemplo que se estuvo utilizando:

```
OBJETOS = parte1.o parte2.o

mi_programa: $ (OBJETOS)
  cc -o mi_programa $ (OBJETOS)
```

Si se utiliza make -n (para ver lo que hace make durante su ejecución) se obtendrá la siguiente salida:

```
cc -c parte1.c
cc -c parte2.c
cc -o mi_programa parte1.o parte2.o
```

La clave es que make, por defecto, espera un archivo objeto "archivo.o" creado al compilar un archivo "archivo.c". Para cada archivo .o listado como una dependencia, make genera una acción cc -c para su archivo fuente. Para mantener este makefile, sumando archivos fuente, basta agregar el archivo objeto a la dependencia. Además, make reconoce nombres estándar de variables tales como CC:

```
OBJETOS = parte1.o parte2.o

mi_programa: $ (OBJETOS)
  $ (CC) -o mi_programa $ (OBJETOS)
```

Este makefile se ejecutará de la misma forma que el anterior. El inconveniente de reducir un makefile al mínimo es que se torna inflexible. Si algún archivo necesita alguna condición en especial, ésta deberá indicarse explícitamente en el mismo.

Para profundizar el tema, conviene leer el manual 'The GNU Make Manual', disponible en www.gnu.org, utilizar las páginas de ayuda disponibles en Linux y el comando make -p -f /dev/null para imprimir todas las reglas internas y variables.