Grado en Ingeniería Informática. Metodología de la Programación.

Práctica 2. El programa `make` y los ficheros *makefile*

Francisco J. Cortijo Bon

Curso 2012-2013

Objetivos

1. Ser conscientes de la dificultad de mantener proyectos complejos (con múltiples ficheros y dependencias) si no se utilizan herramientas específicas.
2. Conocer el funcionamiento de la orden `make`.
3. Conocer la sintaxis de los ficheros *makefile* y cómo son interpretados por `make`.

Índice

La gestión y mantenimiento del software durante el proceso de desarrollo puede ser una tarea ardua si éste se estructura en diferentes ficheros fuente y se utilizan, además, funciones ya incorporadas en ficheros de biblioteca. Durante el proceso de desarrollo se modifica frecuentemente el sotware y las modificaciones incorporadas pueden afectar a otros módulos: la modificación de una función en un módulo afecta *necesariamente* a los módulos que usan dicha función, que deben actualizarse. Estas modificaciones deben *propagarse* a los módulos que dependen de aquellos que han sido modificados, de forma que el programa ejecutable final refleje las modificaciones introducidas.

Esta cascada de modificaciones afectará forzosamente al programa ejecutable final. Si esta secuencia de modificaciones no se realiza de forma ordenada y metódica podemos encontrarnos con un programa ejecutable que no considera las modificaciones introducidas. Este problema es tanto más acusado cuanto mayor sea la complejidad del proyecto software, lo que implica unos complejos diagramas de dependencias entre los módulos implicados, haciendo tedioso y propenso a errores el proceso de propagación hacia el programa ejecutable de las actualizaciones introducidas.

La utilidad `make` proporciona los mecanismos adecuados para la gestión de proyectos software. Esta utilidad mecaniza muchas de las etapas de desarrollo y mantenimiento de un programa, proporcionando mecanismos simples para obtener versiones actualizadas de los programas, por complicado que sea el diagrama de dependencias entre módulos asociado al proyecto. Esto se logra proporcionando a la utilidad `make` la secuencia de mandatos que crean ciertos archivos, y la lista de archivos que necesitan otros archivos (*lista de dependencias*) para ser actualizados antes de que se hagan dichas operaciones. Una vez especificadas las dependencias entre los distintos módulos del proyecto, cualquier cambio en uno de ellos provocará la creación de una nueva versión de los módulos dependientes de aquel que se modifica, reduciendo al mínimo necesario e imprescindible el número de modulos a recompilar para crear un nuevo fichero ejecutable.

La utilización de la orden `make` exige la creación previa de un fichero de descripción llamado genéricamente `makefile`, que contiene las órdenes que debe ejecutar `make`, así como las dependencias entre los distintos módulos del proyecto. Este archivo de descripción es un fichero de texto.

La sintaxis del fichero `makefile` varía ligeramente de un sistema a otro, al igual que la sintaxis de `make`, si bien las líneas básicas son similares y la comprensión y dominio de ambos en un sistema hace que el aprendizaje para otro sistema sea una tarea trivial. Esta visión de generalidad es la que nos impulsa a estudiar esta utilidad y descartemos el uso de gestores de proyectos como los que proporcionan los entornos de programación integrados. En esta sección nos centraremos en la descripción de la utilidad `make` y en la sintaxis de los ficheros `makefile` de GNU.

Resumiendo, el uso de la utilidad `make` conjuntamente con los ficheros makefile proporcionan el mecanismo para realizar una gestión inteligente, sencilla y precisa de un proyecto software, ya que permite:

1. Una forma sencilla de especificar la dependencia entre los módulos de un proyecto software,
2. La recompilación únicamente de los módulos que han de actualizarse,
3. Obtener siempre la versión última que refleja las modificaciones realizadas, y
4. Un mecanismo *casi estándar* de gestión de proyectos software, independiente de la plataforma en la que se desarrolla.

1. El programa *make*

La utilidad `make` utiliza las reglas descritas en el fichero `makefile` para determinar qué ficheros ha de construir y cómo construirlos.

Examinando las listas de dependencia determina qué ficheros ha de reconstruir. El criterio es muy simple, se comparan fechas y horas: si el fichero fuente es más reciente que el fichero destino, reconstruye el destino. Este sencillo mecanismo (suponiendo que se ha especificado correctamente la dependencia entre módulos) hace posible mantener siempre actualizada la última versión.

1.1. Sintaxis

La sintaxis de la llamada al programa `make` es la siguiente:

`make` [*opciones*] [*destino(s)*]]

donde:

- cada ***opción*** va precedida por un signo `-` o una barra inclinada `/`. En la sección 1.2 enumeramos las opciones más importantes.
- ***destino*** indica el destino que debe crear. Generalmente se trata del fichero que debe crear o actualizar, estando especificado en el fichero makefile el procedimiento de creación/actualización del mismo (sección 2.3). Una explicación detallada de los destinos puede encontrarse en las secciones 2.2 y 2.4.

Obsérvese que tanto las opciones como los destinos son opcionales, por lo que podría ejecutarse `make` sin más. El efecto de esta ejecución, así como el funcionamiento detallado de `make` cuando se especifican destinos se explica en la sección 1.3.

1.2. Opciones más importantes

Las opciones más frecuentemente empleadas son las siguientes:

`-h` ó `--help` Proporciona ayuda acerca de `make`.

`-f` *fichero*. Utilizaremos esta opción si se proporciona a `make` un nombre de fichero distinto del de `makefile` o `Makefile`. Se toma el fichero llamado `fichero` como el fichero *makefile*.

`-n` , `--just-print`, `--dry-run` ó `--recon`: Muestra las instrucciones que *ejecutaría* la utilidad `make`, pero **no** los ejecuta. Sirve para verificar la corrección de un fichero makefile.

`-p` , `--print-data-base`: Muestra las reglas y macros asociadas al fichero makefile, incluidas las *predefinidas*.

1.3. Funcionamiento de *make*

El funcionamiento de la utilidad `make` es el siguiente:

1. En primer lugar, busca el fichero makefile que debe interpretar. Si se ha especificado la opción `-f` *fichero*, busca ese fichero. Si no, busca en el directorio actual un fichero llamado `makefile` ó `Makefile`. En cualquier caso, si lo encuentra, lo interpreta; si no, da un mensaje de error y termina.
2. Intenta construir el(los) destino(s) especificado(s). Si no se proporciona ningún destino, intenta construir *solamente* el primer destino que aparece en el fichero makefile. Para construir un destino es posible que deba construir antes otros destinos si el destino especificado depende de otros que no están construidos. Para saber qué destinos debe construir comprueba las listas de dependencias. Esta reacción en cadena se llama **dependencia encadenada**.
3. Si en cualquier paso falla al construir algún destino, se detiene la ejecución, muestra un mensaje de error y borra el destino que estaba construyendo.

2. Ficheros *makefile*

Un fichero makefile contiene las órdenes que debe ejecutar la utilidad `make`, así como las dependencias entre los distintos módulos del proyecto. Este archivo de descripción es un fichero de texto.

Los elementos que pueden incluirse en un fichero makefile son los siguientes:

1. Comentarios.
2. Reglas explícitas.
3. Órdenes.
4. Destinos simbólicos.

2.1. Comentarios

Los comentarios tienen como objeto clarificar el contenido del fichero makefile. Una línea del comentario tiene en su primera columna el símbolo `#`. Los comentarios tienen el ámbito de una línea.

Ejercicio

Crear un fichero llamado `makefile` con el siguiente contenido:

```
# Fichero: makefile
# Construye el ejecutable saludo a partir de saludo.cpp
bin/saludo : src/saludo.cpp
     g++ src/saludo.cpp -o bin/saludo
```

Se incluyen dos líneas de comentario al principio del fichero `makefile` que indican las tareas que realizará la utilidad `make`.
Si el fichero `makefile` se encuentra en el directorio actual, para realizar las acciones en él indicadas tan solo habrá que ejecutar la orden:

```
% make
```

ya que el fichero makefile se llama `makefile`. El mismo efecto hubiéramos obtenido ejecutando la orden:

```
% make -f makefile
```

2.2. Reglas. Reglas explícitas

Las reglas constituyen el mecanismo por el que se indica a la utilidad `make` los destinos (*objetivos*), las listas de dependencias y cómo construir los destinos. Como puede deducirse, son la parte fundamental de un fichero makefile. Las reglas que instruyen a `make` son de dos tipos: explícitas e implícitas y se definen de la siguiente forma:

- Las **reglas explícitas** dan instrucciones a `make` para que construya los ficheros especificados.
- Las **reglas implícitas** dan instrucciones generales que `make` sigue cuando no puede encontrar una regla explícita.

El formato habitual de una regla explícita es el siguiente:

objetivo: ***lista de dependencia***
orden(es)

donde:

- El **objetivo** identifica la regla e indica el fichero a crear.

- La ***lista de dependencia*** especifica los ficheros de los que depende **objetivo**. Esta lista contiene los nombres de los ficheros separados por espacios en blanco.

  Si alguno de los ficheros especificados en esta lista se ha modificado, se busca una regla que contenga a ese fichero como destino y se construye. Una vez se han construido las últimas versiones de los ficheros especificados en ***lista de dependencia*** se construye **objetivo**.

- Las ***órden(es)*** son órdenes válidas para el sistema operativo en el que se ejecute la utilidad `make`. Pueden incluirse varias instrucciones en una regla, cada uno en una línea distinta. Usualmente estas instrucciones sirven para construir el **objetivo** (en esta asignatura, habitualmente son llamadas al compilador `g++`), aunque no tiene porque ser así.

  MUY IMPORTANTE: Cada línea de órdenes empezará con un TABULADOR. Si no es así, `make` mostrará un error y no continuará procesando el fichero makefile.

Ejercicio

En el ejemplo anterior (fichero `makefile`) encontramos una única regla:

```
bin/saludo : src/saludo.cpp
     g++ src/saludo.cpp -o bin/saludo
```

que indica que para construir el objetivo `saludo` se requiere la existencia de `saludo.cpp` (`saludo` *depende de* `saludo.cpp`). Esta dependencia se esquematiza en el diagrama de dependencias mostrado en la figura 1. Finalmente, el destino se construye ejecutando la orden:

```
g++ src/saludo.cpp -o bin/saludo
```

que compila el fichero `saludo.cpp` generando un fichero objeto temporal y lo enlaza con las bibliotecas adecuadas para generar finalmente `saludo`.

1. Ejecutar las siguientes órdenes e interpretar el resultado:

   ```
   % make
   % make -f makefile
   % make bin/saludo
   % make -f makefile bin/saludo
   ```

2. Antes de volver a ejecutar `make` con las cuatro variantes enumeradas anteriormente, modificar el fichero `saludo.cpp`. Interpretar el resultado.

3. Probar la orden `touch` sobre `saludo.cpp` y volver a ejecutar `make`. Interpretar el resultado.

Figura 1: Diagrama de dependencias para `saludo`

Ejercicio

A partir del diagrama de dependencias mostrado en la figura 2 construir el fichero makefile llamado `makefile2.mak`.
Ejecutar las siguientes órdenes e interpretar el resultado:

```
% make -f makefile2.mak bin/saludo

% make -f makefile2.mak
```

Figura 2: Diagrama de dependencias para `makefile2.mak`

2.3. Órdenes. Prefijos de órdenes

Como se indicó anteriormente, se puede incluir cualquier orden válida del sistema operativo en el que se ejecute la utilidad `make`. Pueden incluirse cuantas órdenes se requieran como parte de una regla, cada una en una línea distinta, y como nota importante, recordamos que es *imprescindible* que cada línea empiece con un tabulador para que `make` interprete correctamente el fichero `makefile`.

Las órdenes pueden ir precedidas por **prefijos**. Los más importantes son:

`@` Desactivar el *eco* durante la ejecución de esa orden.

`-` Ignorar los errores que puede producir la orden a la que precede.

Ejercicio

Copiar el fichero `makefile3.mak` en vuestro directorio de trabajo.
En este fichero makefile se especifican dos órdenes en cada una de las reglas, entre ellas una para mostrar un mensaje en pantalla (`echo`), indicando qué acción se desencadena en cada caso. Las órdenes `echo` van precedidos por el prefijo `@` en el fichero makefile para indicar que debe desactivarse el eco durante la ejecución de esa instrucción.

1. Ejecutar `make` sobre este fichero makefile e interpretar el resultado.
2. Poner el prefijo `@` en la llamada a `g++` y volver a ejecutar `make`.
3. Eliminar los prefijos y volver a ejecutar `make`.

Ejercicio

Usando como base `makefile3.mak` añadir (al final) la siguiente regla, y guardar el nuevo contenido en el fichero `makefile4.mak`:

```
# Esta regla especifica un destino sin lista de dependencia
clean :
     @echo Borrando ficheros objeto
     rm obj/*.o
```

Esta nueva regla, cuyo destino es `clean` no tiene asociada una lista de dependencia. La construcción del destino `clean` no requiere la construcción de otro destino previo ya que la construcción de ese destino no depende de nada. Para ello bastará ejecutar:

```
% make -f makefile4.mak clean
```

2.4. Destinos Simbólicos

Un destino simbólico se especifica en un fichero makefile en la primera línea operativa del mismo. En su sintaxis se asemeja a la especificación de una regla, con la diferencia que no tiene asociada ninguna orden. El formato es el siguiente:

destino simbólico: ***lista de destinos***

donde:

- **destino simbólico** es el nombre del destino simbólico. El nombre particular no tiene ninguna importancia, como se deducirá de nuestra explicación.
- ***lista de destinos*** especifica los destinos que se construirán cuando se invoque a `make`.

La finalidad de incluir un destino simbólico en un fichero makefile es la de que se construyan varios destinos sin necesidad de invocar a `make` tantas veces como destinos se desee construir.

Al estar en la primera línea operativa del fichero makefile, la utilidad `make` intentará construir el **destino simbólico**. Para ello, examinará la lista de dependencia (llamada ahora *lista de destinos*) y construirá cada uno de los destinos de esta lista: **debe existir una regla para cada uno de los destinos**. Finalmente, intentará construir el destino simbólico y como no habrá ningúna instrucción que le indique a `make` cómo ha de construirlo no hará nada más. Pero el objetivo está cumplido: se han construido varios destinos con una sóla ejecución de `make`. Obsérvese cómo el nombre dado al destino simbólico no tiene importancia.

Ejercicio

Copiar el fichero `unico.cpp` en vuestro directorio de trabajo. Construir el fichero `makefile5.mak` a partir de `makefile4.mak` con un destino simbólico llamado `todo` que cree los ejecutables `saludo` y `unico`. Ejecutar:

1. `% make -f makefile5.mak todo`
2. `% make -f makefile5.mak`

Ejercicio

Añadir el destino `clean` a la lista de dependencia del destino simbólico `todo`, así como la regla asociada.
Añadir un destino llamado `salva` a la lista de dependencia del destino simbólico `todo`, así como la regla asociada:

```
salva : saludo unico
     echo Creando directorio resultado
     mkdir resultado
     echo Moviendo los ejecutables al directorio resultado
     move $^ resultado
```

En la última orden asociada a la última regla se hace uso de la macro `$^` que se sustituye por todos los nombres de los ficheros de la lista de dependencias. O sea, `make` interpreta la orden anterior como:

```
     move saludo unico resultado
```

2.5. Destinos .PHONY

Si en un fichero makefile apareciera una regla:

```
clean :
  @echo Borrando ficheros objeto
  rm obj/*.o
```

y hubiera un fichero llamado `clean`, la ejecución de la orden:

```
make clean
```

no funcionará como está previsto (no borrará los ficheros) puesto que el destino `clean` no tiene ninguna dependencia y existe el fichero `clean`. Así, `make` supone que no tiene que volver a generar el fichero `clean` con la orden asociada a esta regla, pues el fichero `clean` está actualizado, y no ejecutaría la orden que borra los ficheros de extensión `.o`. Una forma de solucionarlo es declarar este tipo de destinos como *falsos (phony)* usando `.PHONY` de la siguiente forma:

```
.PHONY : clean
```

Esta regla la podemos poner en cualquier parte del fichero makefile, aunque normalmente se coloca antes de la regla `clean`. Haciendo esto, al ejecutar la orden

```
make clean
```

todo funcionará bien, aunque exista un fichero llamado `clean`. Observar que también sería conveniente hacer lo mismo para el caso del destino simbólico `saludos` en los ejemplos anteriores.

2.6. Macros en ficheros *makefile*

Una **macro o variable MAKE** es una *cadena* que se expande cuando se usa en un fichero makefile.

Las macros permiten crear ficheros makefile genéricos o *plantilla* que se adaptan a diferentes proyectos software. Una macro puede representar listas de nombres de ficheros, opciones del compilador, programas a ejecutar, directorios donde buscar los ficheros fuente, directorios donde escribir la salida, etc. Puede verse como una versión más potente que la directiva `#define` de C++, pero aplicada a ficheros makefile.

La sintaxis de definición de macros en un fichero makefile es la siguiente:

NombreMacro = texto a expandir

donde:

- **NombreMacro** es el nombre de la macro. Es sensible a las mayúsculas y no puede contener espacios en blanco. La costumbre es utilizar nombres en mayúscula.
- **texto a expandir** es una cadena que puede contener cualquier carácter alfanumérico, de puntuación o espacios en blanco

Para definir una macro llamada, por ejemplo, `OBJ` que representa a la cadena `~/MP/obj` se especificará de la siguiente manera:

```
OBJ = ~/MP/obj
```

Si esta línea se incluye en el fichero makefile, cuando `make` encuentra la construcción `$(OBJ)` en él, sustituye dicha construcción por `~/MP/obj`. Cada macro debe estar en una línea separada en un fichero makefile y se situan, normalmente, al principio de éste. Si `make` encuentra más de una definición para el mismo nombre (no es habitual), la nueva definición reemplaza a la antigua.

La expansión de la macro se hace *recursivamente*. O sea, que si la macro contiene referencias a otras macros, estas referencias serán expandidas también. Veamos un ejemplo. Podemos definir una macro para cada uno de los directorios de trabajo:

```
SRC = src
BIN = bin
OBJ = obj
INCLUDE = include
LIB = lib
```

Si el fichero makefile está situado en la misma carpeta que los directorios, y en el fichero aparece:

```
$(BIN)/saludo: $(SRC)/saludo.cpp
     g++ -o $(BIN)/saludo $(SRC)/saludo.cpp
```

se sustituye por:

```
bin/saludo: src/saludo.cpp
     g++ -o bin/saludo src/saludo.cpp
```

¿y si el fichero makefile estuviera en una carpeta distinta? Podría añadirse una macro (la primera):

```
HOMEDIR = /home/users/app/new/MP
```

y se modifican las anteriores por:

```
SRC = $(HOMEDIR)/src
BIN = $(HOMEDIR)/bin
OBJ = $(HOMEDIR)/obj
INCLUDE = $(HOMEDIR)/include
LIB = $(HOMEDIR)/lib
```

Ahora, la regla anterior se sustituye por:

```
/home/users/app/new/MP/bin/saludo: /home/users/app/new/MP/saludo.cpp
     g++ -o /home/users/app/new/MP/saludo /home/users/app/new/MP/saludo.cpp
```

Si los directorios se situaran en otra carpeta bastará con cambiar la macro `HOMEDIR`. En nuestro caso podríamos escribir:

```
HOMEDIR = ~/MP
```

Ejercicio

Modificar el fichero `makefile5.mak` para que incluya las macros referentes a los directorios que hemos enumerado anteriormente.

2.7. Macros predefinidas

Las macros predefinidas utilizadas habitualmente en ficheros makefile son las que enumeramos a continuación:

- `$@` Nombre del fichero *destino* de la regla.
- `$<` Nombre de la *primera dependencia* de la regla.
- `$^` Equivale a *todas* las dependencias de la regla, con un espacio entre ellas.
- `$?` Equivale a *las dependencias de la regla más nuevas que el destino*, con un espacio entre ellas.

Ejercicio

Modificar el fichero `makefile5.mak` de manera que se muestren los valores de las macros predefinidas `$@` , `$<`, `$^`, y `$?` en las reglas que generan algún fichero como resultado. Usad la orden `@echo`

A. Sustituciones en macros

La utilidad `make` permite sustituir caracteres temporalmente en una macro previamente definida. La sintaxis de la sustitución en macros es la siguiente:

`$`**(NombreMacro:TextoOriginal = TextoNuevo)**

que se interpreta como: sustituir en la cadena asociada a **NombreMacro** todas las apariciones de **TextoOriginal** por **TextoNuevo**. Es importante resaltar que:

1. **No** se permiten espacios en blanco antes o despues de los dos puntos.
2. **No** se redefine la macro **NombreMacro**, se trata de una sustitución *temporal*, por lo que **NombreMacro** mantiene el valor dado en su definición.

Por ejemplo, dada una macro llamada `FUENTE` definida como:

```
FUENTES = f1.cpp f2.cpp f3.cpp
```

se pueden sustituir *temporalmente* los caracteres `.cpp` por `.o` escribiendo `$(FUENTES:.cpp=.o)` que da como resultado `f1.o f2.o f3.o`. El valor de la macro `FUENTES` **no** se modifica, ya que la sustitución es temporal.

B. Macros como parámetros en la llamada a `make`

Además de las opciones básicas indicadas en la sección 1.2, hay una opción que permite especificar el valor de una constante simbólica que se emplea en un fichero makefile en la llamada a `make` en lugar de especificar su valor en el fichero makefile.

El mecanismo de sustitución es similar al expuesto anteriormente, salvo que ahora `make` no busca el valor de la macro en el fichero makefile. La sintaxis de la llamada a `make` con macros es la siguiente:

`make` ***NombreMacro***[=***cadena***] [*opciones...*] [*destino(s)*]]

> ***NombreMacro***[=***cadena***] define la constante simbólica ***NombreMacro*** con el valor especificado (si lo hubiera) después del signo `=`. Si ***cadena*** contiene espacios, será necesario encerrar ***cadena*** entre comillas.
>
> Si ***NombreMacro*** también está definida dentro del fichero makefile, se ignorará la definición del fichero.

El uso de macros en llamadas a `make` permite la construcción de ficheros makefile genéricos, ya que el mismo fichero puede utilizarse para diferentes tareas que se deciden en el momento de invocar a `make` con el valor apropiado de la macro.

Ejercicio

Modificar el fichero `makefile_ppal_2` para que el resultado (el ejecutable `ppal_2`) lo guarde en un directorio cuyo nombre **completo** (camino absoluto, desde la raiz `/`) se indica como parámetro al fichero makefile con una macro llamada `DESTDIR`.
Importante: Como el directorio puede no existir, crearlo en el propio fichero makefile.

1. Ejecutar `make` sobre este fichero especificando el directorio destino apropiadamente.
2. ¿Qué ocurre si se vuelve a ejecutar la orden anterior?
3. Modificar apropiadamente el fichero `makefile_ppal_2` para evitar el error.

Ejercicio

Extender el makefile anterior con una opción para incluir información de depuración o no.

C. Reglas implícitas

En los ficheros makefile aparecen, en la mayoría de los casos, reglas que se parecen mucho. En los ejercicios anteriores se puede ver, por ejemplo, que las reglas que generan los ficheros objeto son idénticas, sólo se diferencian en los nombres de los ficheros que manipulan.

Las reglas implícitas son reglas que `make` interpreta para actualizar destinos sin tener que escribirlas dentro del fichero makefile. De otra forma: *si no se especifica una regla explícita para construir un destino, se utilizará una regla implícita (que no hay que escribir)*.

Existe un catálogo de reglas implícitas predefinidas que pueden usarse para distintos lenguajes de programación (ver `http://www.gnu.org/software/make/manual/make.html#Implicit-Rules`) . El que `make` elija una u otra dependerá del nombre y extensión de los ficheros. Por ejemplo, para el caso que nos interesa, compilación de programas en C++, existe una regla implícita que dice cómo obtener el fichero objeto (`.o`) a partir del fichero fuente (`.cpp`). Esa regla se usa cuando no existe una regla explícita que diga como construir ese módulo objeto. En tal caso se ejecutará la siguiente orden para construir el módulo objeto cuando el fichero fuente sea modificado:

```
$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)
```

Las reglas implícitas que usa `make` utilizan una serie de macros predefinidas, tales como las del caso anterior (`CXX`, `CPPFLAGS` y `CXXFLAGS`). Para más información ver
`http://www.gnu.org/software/make/manual/make.html#Implicit-Variables`

El valor de estas macros pueden ser definidas:

1. Dentro del fichero makefile,
2. A través de argumentos pasados a `make`, o
3. Con valores predefinidos. Por ejemplo,
   - `CXX`: Programa para compilar programas en C++; Por defecto: `g++`.
   - `CPPFLAGS`: Modificadores extra para el preprocesador de C. El valor por defecto es la cadena vacía.
   - `CXXFLAGS`: Modificadores extra para el compilador de C++. El valor por defecto es la cadena vacía.

Si deseamos que `make` use reglas implícitas, en el fichero makefile escribiremos la regla *sin ninguna orden*. Es posible añadir nuevas dependencias a la regla.

Ejercicio

Usar como base `makefile_ppal_2` y copiarlo en `makefile_ppal_3`, modificando la regla que crea el ejecutable para que éste se llame `ppal_3`.

1. Eliminar las reglas que crean los ficheros objeto y ejecutar `make`.
2. Forzar la dependencia de los módulos objeto respecto a los ficheros de cabecera (`.h`) adecuados para que cuando se modifique algún fichero de cabecera se ejecute la regla implícita que actualiza el o los ficheros objeto necesarios, y finalmente el ejecutable.

   Nota: Los ficheros de cabecera se encuentran en un subdirectorio del directorio `MP` llamado `include`. Modificar la variable `CXXFLAGS` con el valor `-I$(INCLUDE)` (se expanderá a `-I./include` para cada ejecución de la regla implícita).

   a) Modificar (`touch`) `adicion.cpp` y ejecutar `make`. Observad qué instrucciones se ejecutan y en qué orden.

   b) Modificar (`touch`) `adicion.h` y ejecutar `make`. Observad qué instrucciones se ejecutan y en qué orden.

Otra regla que puede usarse es la regla implícita que permite enlazar el programa. La siguiente regla:

```
$(CC) $(LDFLAGS) n.o $(LOADLIBES) $(LDLIBS)
```

se interpreta como sigue: `n` se construirá a partir de `n.o` usando el enlazador (`ld`). Esta regla funciona correctamente para programas *con un solo fichero fuente* aunque también funcionarácorrectamente en programas con múltiples ficheros objeto si uno de los cuales tiene el mismo nombre que el ejecutable.

Ejercicio

Copiar `ppal_2.cpp` en `ppal_4.cpp`
Usar como base `makefile_ppal_3` y copiarlo en `makefile_ppal_4`,
Eliminar la orden en la regla que crea el ejecutable manteniendo la lista de dependencia y ejecutar `make`.

C.1. Reglas implícitas patrón

Las reglas implícitas patrón pueden ser utilizadas por el usuario para definir nuevas reglas implícitas en un fichero makefile. También pueden ser utilizadas para redefinir las reglas implícitas que proporciona `make` para adaptarlas a nuestras necesidades. Una *regla patrón* es parecida a una regla normal, pero el destino de la regla contiene el caracter `%` en alguna parte (sólo una vez). Este destino constituye entonces un patrón para emparejar nombres de ficheros.

Por ejemplo el destino `%.o` empareja a todos los ficheros con extensión `.o`. Una regla patrón `%.o:%.cpp` dice cómo construir cualquier fichero `.o` a partir del fichero `.cpp` correspondiente. En una regla patrón podemos tener varias dependencias que también pueden contener el caracter `%`. Por ejemplo:

```
%.o : %.cpp %.h comun.h
    g++ -c $< -o $@
```

significa que cada fichero `.o` debe volver a construirse cuando se modifique el `.cpp` o el `.h` correspondiente, o bien `comun.h`. Una reglas patrón del tipo `%.o:%.cpp` puede simplificarse escribiendo `.cpp.o:`
La *regla implícita patrón predefinida* para compilar ficheros `.cpp` y obtener ficheros `.o` es la siguiente:

```
%.o : %.cpp
    $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@
```

Esta regla podría ser redefinida en nuestro fichero makefile escribiendo esta regla con el mismo destino y dependencias, pero modificando las órdenes a nuestra conveniencia. Si queremos que `make` ignore una regla implícita podemos escribir una regla patrón con el mismo destino y dependencias que la regla implícita predefinida, y sin ninguna orden asociada.

IMPORTANTE: Una regla regla implícita patrón puede aplicarse a cualquier destino que se empareja con su patrón, pero sólo se aplicará cuando el destino no tiene órdenes que lo construya mediante otra regla distinta, y sólo cuando puededan encontrase las dependecias. Cuando se puede aplicar más de una regla implícita, sólo se aplicará una de ellas: la elección depende del orden de las reglas.

C.2. Reglas patrón estáticas

Las reglas patrón estáticas son otro tipo de reglas que se pueden utilizar en ficheros makefile y que son muy parecidas en su funcionamiento a las reglas implícitas patrón. Estas reglas no se consideran implícitas, pero al ser muy parecidas en funcionamiento a las reglas patrón implícitas, las exponemos en esta sección. El formato de estas reglas es el siguiente:

destino(s): *patrón de destino* : *patrones de dependencia*
orden(es)

donde:

- La lista *destinos* especifica a qué destinos se aplicará la regla. Esta es la principal diferencia con las reglas patrón implícitas. Ahora la regla se aplica únicamente a la lista de destinos, mientras que en las implícitas se intenta aplicar a todos los que se emparejan con el patrón destino. Los destinos pueden contener caracteres comodín como `*` y `?`
- El *patrón de destino* y los *patrones de dependencia* dicen cómo calcular las dependencias para cada destino.

Veamos un pequeño ejemplo que muestra como obtener los ficheros `f1.o` y `f2.o`.

```
OBJETOS = f1.o f2.o

$(OBJETOS): %.o: %.cpp
        g++ -c $(CFLAGS) $< -o $@
```

En este ejemplo, la regla sólo se aplica a los ficheros `f1.cpp` y `f2.cpp`. Si existen otros ficheros con extensión `.cpp`, esta regla no se aplicará ya que no se han incluido en la lista *destinos*.

D. Directivas condicionales en ficheros *makefile*

Las directivas condicionales se parecen a las directivas condicionales del preprocesador de C++. Permiten a `make` dirigir el flujo de procesamiento en un fichero makefile a un bloque u otro dependiendo del resultado de la evaluación de una condición, evaluada con una directiva condicional.

Para más información ver `http://www.gnu.org/software/make/manual/make.html#Conditionals`

La sintaxis de un condicional simple sin else sería la siguiente:

directiva condicional
    *texto (si el resultado es **verdad**)*
`endif`

La sintaxis para un condicional con parte `else` sería:

directiva condicional
    *texto (si el resultado es **verdad**)*
`else`
    *texto (si el resultado es **falso**)*
`endif`

Las directivas condicionales para ficheros makefile son las siguientes:

`ifdef` *macro*: Actúa como la directiva `#ifdef` de C++ pero con macros en lugar de directivas `#define`.

`ifndef` *macro*: Actúa como la directiva `#ifndef` de C++ pero con macros, en lugar de directivas `#define`.

`ifeq (`*arg1*`,`*arg2*`)` ó `ifeq '`*arg1*`' '`*arg2*`'` ó `ifeq "`*arg1*`" "`*arg2*`"` ó `ifeq "`*arg1*`" '`*arg2*`'` ó `ifeq '`*arg1*`' "`*arg2*`"`

Devuelve verdad si los dos argumentos expandidos son iguales.

`ifneq (`*arg1*`,`*arg2*`)` ó `ifneq '`*arg1*`' '`*arg2*`'` ó `ifneq "`*arg1*`" "`*arg2*`"` ó `ifneq "`*arg1*`" '`*arg2*`'` ó `ifneq '`*arg1*`' "`*arg2*`"`

Devuelve verdad si los dos argumentos expandidos son distintos

`else`

Actúa como un `else` de C++.

`endif`

Termina una declaración `ifdef`, `ifndef` `ifeq` ó `ifneq`.