Asignatura: Entornos de programación
Herramientas de construcción (*Build*)

Utilidad ‘make’ y similares

1. Objetivo y funciones

El objetivo general de las herramientas de construcción (*Build* o *Make*) es automatizar el proceso de generación y/o actualización de un conjunto de ficheros (objetivo) que se construyen a partir de otros (fuente). Y no sólo automatizarlo, sino hacerlo de manera eficiente.

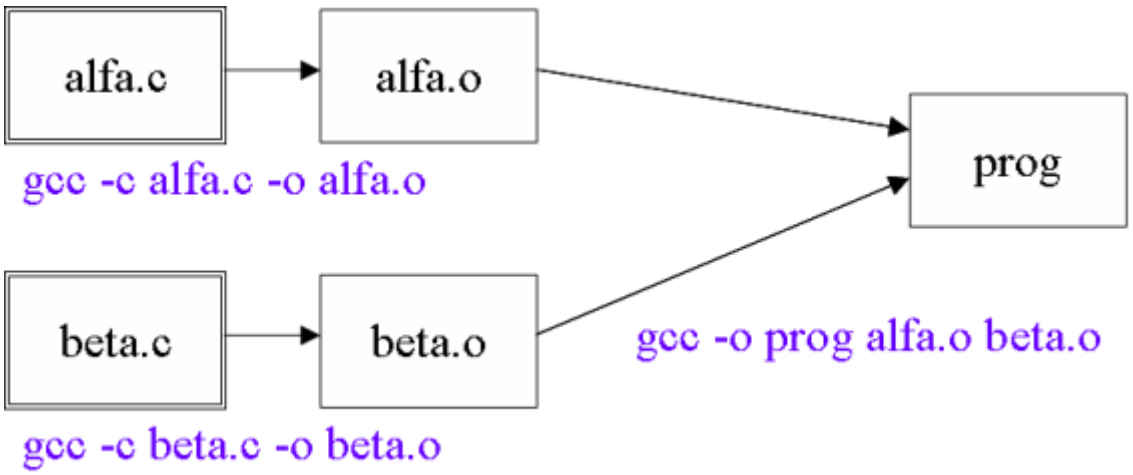
La automatización del proceso de construcción se podría hacer simplemente con un guión fijo de órdenes. Esto resulta adecuado para construir el objetivo por primera vez, pero no es lo más eficiente para actualizar un objetivo ya construido tras algunos cambios en los ficheros fuente, ya que se reconstruirían siempre todos los ficheros cada vez que se invocase esta operación, cuando en muchos casos sólo algunos de los ficheros a reconstruir necesitan realmente ser actualizados.

Una forma sencilla de conseguir un buen grado de eficiencia es comparar la fecha de actualización de cada fichero a reconstruir con la de los ficheros a partir de los cuales hay que regenerarlo. Si el fichero objetivo es posterior a todos aquellos ficheros fuente de los que depende, entonces no necesitaría ser regenerado.

La aplicación típica de estas herramientas de construcción es la compilación y montaje de programas ejecutables a partir de los ficheros fuente correspondientes. Esto es especialmente importante en el caso de lenguajes tales como C y C++, cuyos compiladores no tienen una forma razonablemente segura de identificar todos los ficheros fuente necesarios para construir un programa ejecutable completo partiendo sólo del fichero fuente principal.

2. Compilación y montaje de programas

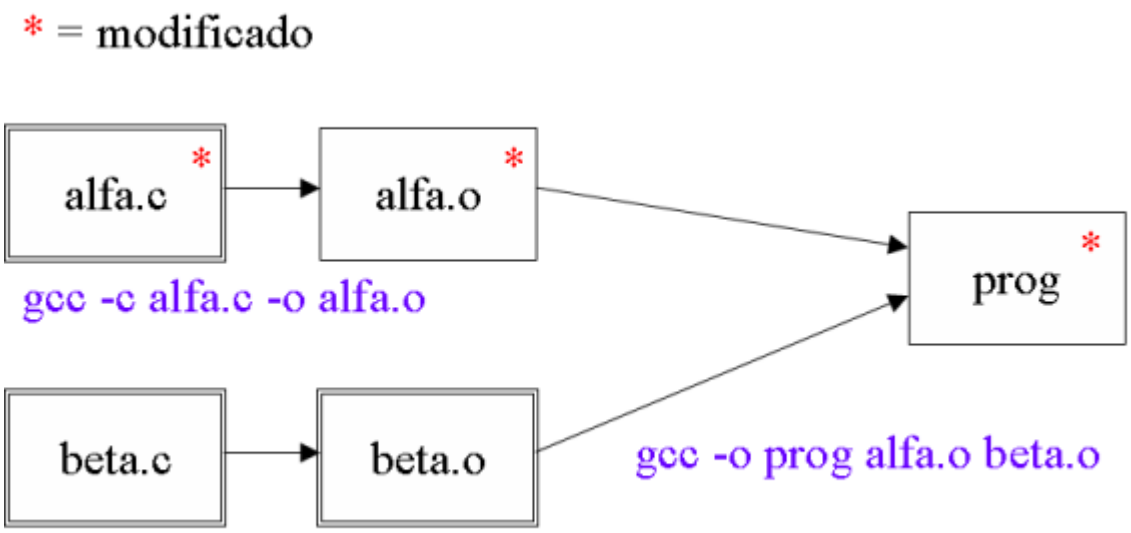
El proceso clásico de construcción de un programa ejecutable a partir de los ficheros fuente consiste en compilar cada fichero fuente por separado, para obtener el correspondiente fichero objeto, y luego combinar los ficheros objeto (junto con funciones tomadas de la librería estándar) para obtener el programa ejecutable. En el ejemplo de la figura se muestra la construcción de un programa ejecutable a partir de dos ficheros fuente en lenguaje C:



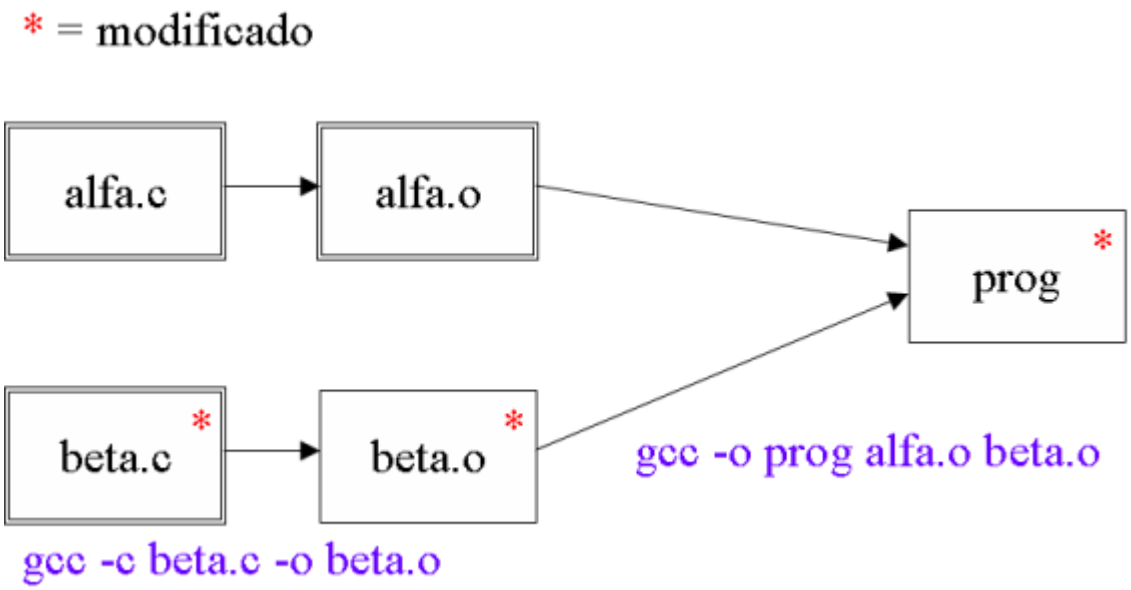
3. Recompilación selectiva

Tras construir el programa por primera vez y conservar los ficheros objeto intermedios, sólo será necesarios recompilar los ficheros fuente que se hayan modificado desde la última construcción. En general será siempre necesario repetir el paso final de combinar los objetos para obtener el ejecutable. Las figuras siguientes muestran el proceso de reconstruir el ejemplo anterior tras modificar sólo uno u otro de los ficheros fuente, pero no los dos a la vez.

- Tras modificar sólo `alfa.c`:

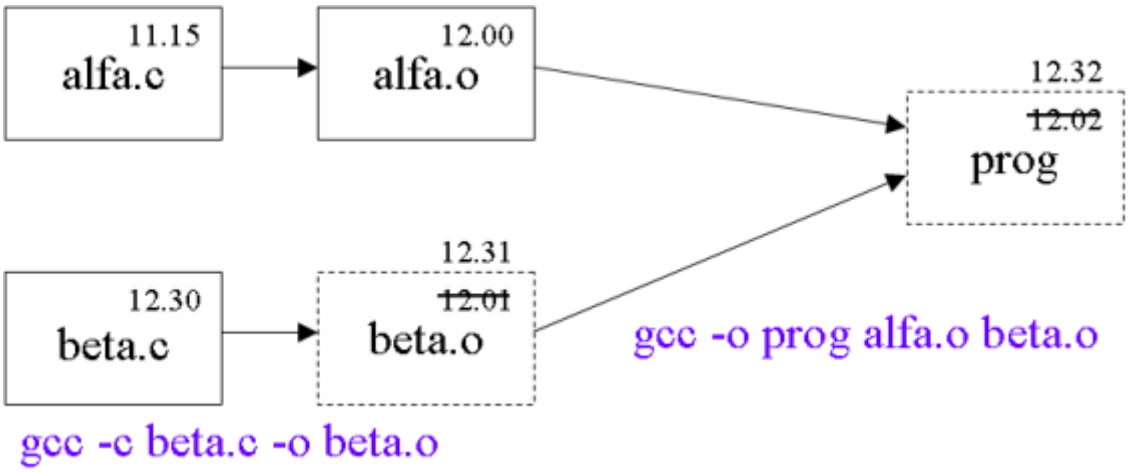


- Tras modificar sólo `beta.c`:



4. Recompilación automática mediante comparación de fechas

El proceso de recompilación selectiva puede automatizarse mediante una herramienta que compare las fechas de actualización de los ficheros fuente y las de los ficheros objeto y ejecutable. Sólo hay que regenerar los ficheros cuya fecha de actualización sea anterior a la de aquellos a partir de los cuales se reconstruye. En la figura se muestran ejemplos de la hora de actualización de cada fichero, y los cambios que podrían producirse al actualizar los que no están al día:



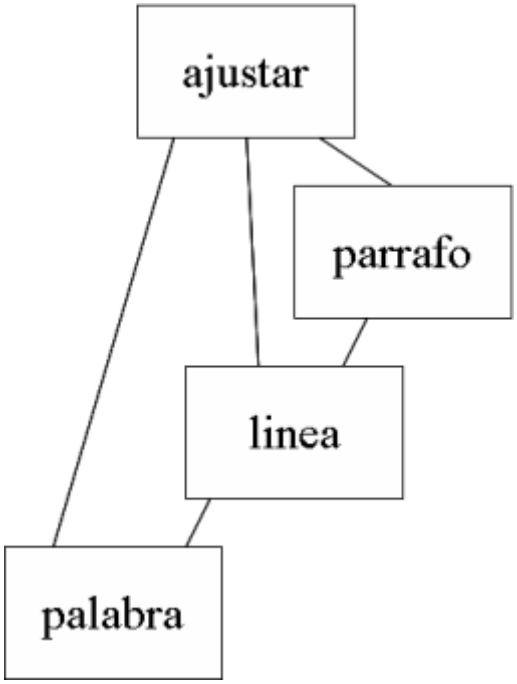
Las acciones hay que realizarlas en un orden adecuado. No tiene sentido regenerar un fichero en particular a partir de otros que no están actualizados, sino que la regeneración de cada fichero debe hacerse después de haber actualizado a su vez, en caso necesario, cada uno de los ficheros a partir de los cuales hay que hacer su reconstrucción.

5. Dependencias múltiples

En casos muy sencillos como el del ejemplo anterior no hace falta ninguna herramienta especial que ayude a construir el programa. Se podrían invocar las operaciones manualmente. Pero en la mayoría de los casos reales, con un mayor número de ficheros fuente y dependencias complejas entre ellos, es necesario disponer de alguna herramienta de ayuda. A continuación se muestra otro ejemplo de programa modular algo más complejo que el anterior. Realmente sigue siendo muy sencillo, pero ya se empiezan a apreciar las dificultades para reconstruirlo de manera eficiente sin herramientas de ayuda.

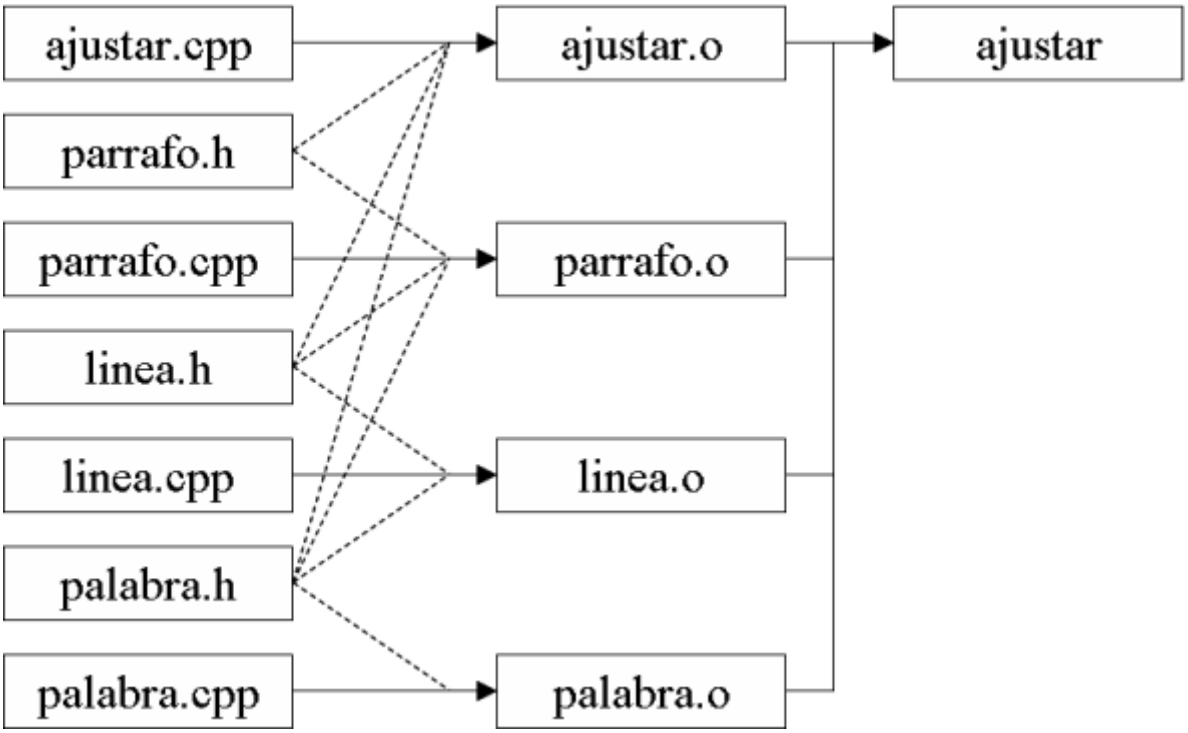
5.1 Ejemplo: diagrama modular

El diagrama modular de este ejemplo es:



5.2 Ejemplo: dependencias entre ficheros al compilar

Cada módulo (excepto quizá el programa principal) debe corresponder a dos ficheros fuente: uno de cabecera (`.h`) con la descripción de su interfaz, y otro (`.cpp`) con el código de implementación. Las dependencias de los ficheros objeto y ejecutable con los ficheros fuente individuales sería:

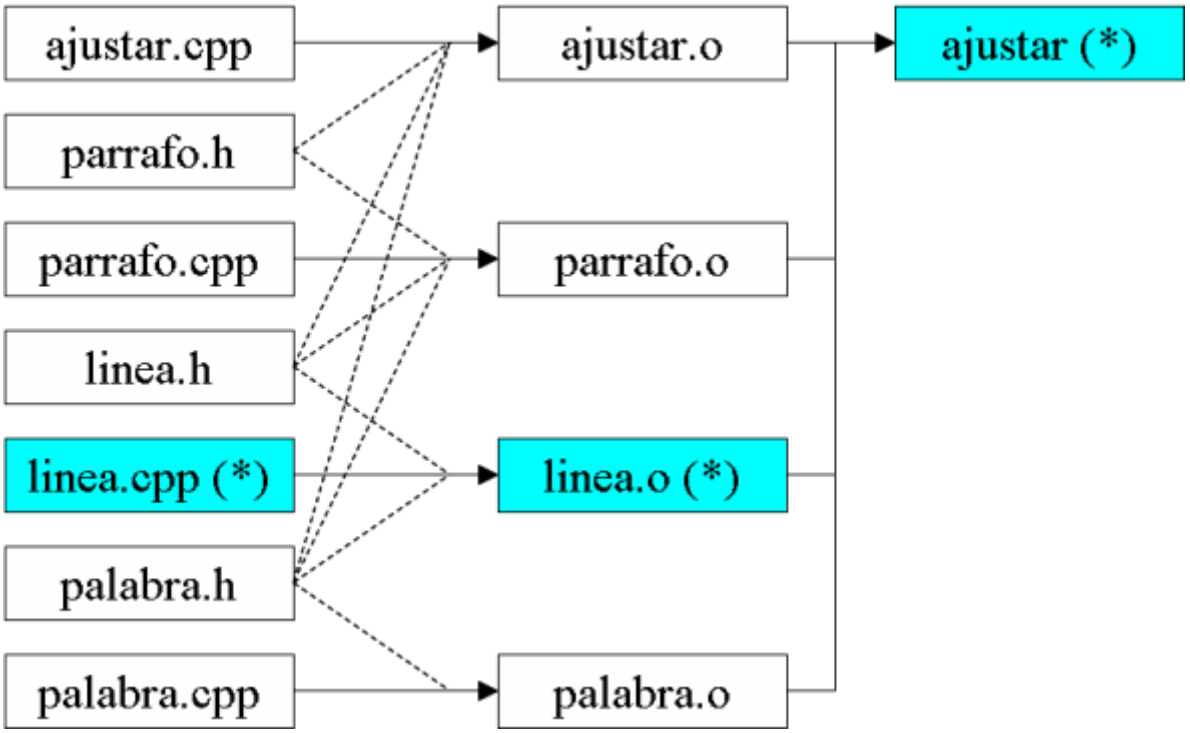


La flechas con línea continua representan dependencias directas, y las de línea discontinua representan dependencias indirectas, debidas a que ciertos ficheros fuente importan (con `#include`) ciertos ficheros de cabecera.

Tanto las dependencias directas como las indirectas deben ser tenidas en cuenta para determinar si un determinado fichero objeto está al día o no.

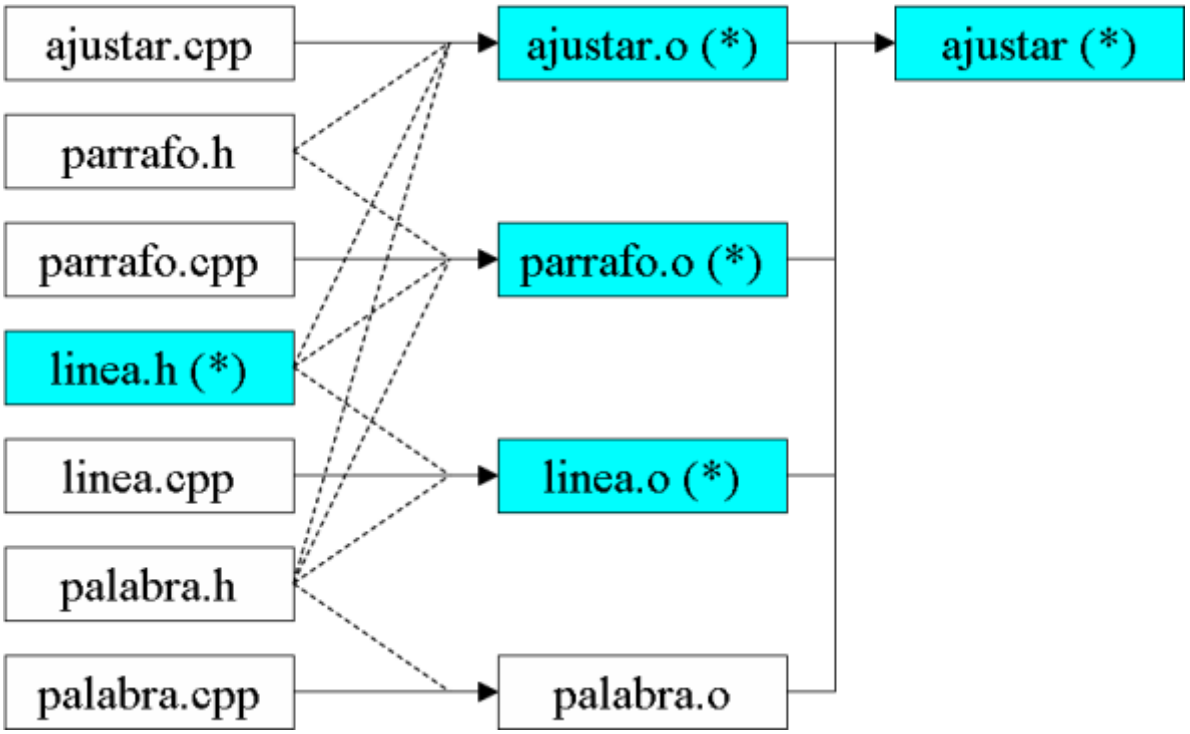
5.3 Ejemplo: regeneración simple

Cuando cambia algún fichero fuente hay que reconstruir todo lo que depende de él, directa o indirectamente. Si cambia un fichero de implementación la reconstrucción exige normalmente poco trabajo:



5.4 Ejemplo: regeneración compleja

Por el contrario, si cambia un fichero de cabecera, especialmente de los módulos de nivel inferior, entonces el trabajo de reconstrucción puede ser bastante grande:



A pesar de lo sencillo de este ejemplo, ya se empieza a ver la dificultad de realizar manualmente las reconstrucciones, porque es fácil olvidar alguna de las dependencias entre ficheros, especialmente las indirectas.

6. Utilidad ‘*make*’

Este herramienta, originada en UNIX, sirve para automatizar el proceso de regeneración selectiva de ciertos ficheros a partir de otros de los que depende:

- Detecta los ficheros no actualizados mediante la comparación de fechas
- Necesita información de las dependencias entre ficheros y las acciones para regenerarlos
- Esta información se suministra en un fichero denominado genéricamente ‘*makefile*’, usando una notación específica, basada en reglas
- Cada regla contiene las dependencias de un fichero respecto a los demás, y las acciones para regenerarlo

La herramienta *make* opera de forma recursiva, partiendo del objetivo final. Para cada posible fichero a regenerar (fichero *objetivo*), primero se reconstruyen, si es necesario, los ficheros de los que depende de manera inmediata, y a continuación se comprueban las fechas de actualización (que pueden haber cambiado si se ha regenerado alguno de ellos) para ver si el objetivo está al día.

6.1 Formato del ‘*makefile*’

Los *makefile* deben contener la información que necesita *make* para operar. En concreto:

- Qué ficheros intervienen en el proceso de reconstrucción
- Dependencias de unos ficheros respecto a otros. Es decir, qué ficheros hay que regenerar cuando cambian otros ficheros
- Qué acciones hay que realizar para regenerar cada fichero dependiente

En los casos sencillos, un *makefile* contiene una regla por cada fichero que haya que regenerar automáticamente. Cada regla empieza por una línea de texto que indica el nombre del fichero objetivo y los ficheros de los que depende. A continuación se indican las órdenes a ejecutar para reconstruir el objetivo.

El formato general del *makefile* será:

```
objetivo: dependencia dependencia ...
⇒ accion
⇒ accion
⇒ . . .
objetivo: dependencia ...
⇒ accion
⇒ . . .
```

- Nota: el símbolo ⇒ es el carácter de tabulación

La primera línea de una regla es la línea de dependencias. Debe empezar con el nombre del fichero objetivo, seguido del carácter dos puntos (`:`) y la lista de ficheros de los que depende, separados por espacio en blanco.

Las siguientes líneas de la regla son las órdenes para regenerar el objetivo, y dejan cierto margen a la izquierda. Por razones históricas deben empezar necesariamente con una carácter de tabulación. No es suficiente dejar el margen en blanco sólo con espacios. El analizador léxico del *makefile* usa precisamente el carácter de tabulación para identificar las líneas de órdenes que forman parte de la regla.

6.2 Ejemplo de ‘*makefile*’

El *makefile* correspondiente al ejemplo anterior (programa `ajustar`) podría ser el siguiente:

```
ajustar: ajustar.o parrafo.o linea.o palabra.o
    g++ -o ajustar ajustar.o parrafo.o linea.o palabra.o

palabra.o: palabra.cpp palabra.h
    g++ -c palabra.cpp -o palabra.o

linea.o: linea.cpp linea.h palabra.h
    g++ -c linea.cpp -o linea.o

parrafo.o: parrafo.cpp parrafo.h linea.h palabra.h
    g++ -c parrafo.cpp -o parrafo.o
```

6.3 Cómo invocar ‘*make*’

La utilidad *make* se invoca mediante la orden:

```
make [ -f makefile ] [ opciones ] [ objetivos ]
```

Los parámetros son:

- *makefile* – fichero de reglas (por defecto `makefile` o `Makefile`)
- *opción* **`-k`** – continuar aunque haya errores (para otras opciones consultar el manual)
- *objetivos* – objetivos a actualizar (por defecto el de la primera regla que haya en el fichero *makefile*)

En los casos sencillos no hace falta ningún parámetro. La orden:

```
make
```

busca un fichero que se llame `makefile` o `Makefile` y lo utiliza para actualizar el objetivo de la primera regla que haya en dicho fichero. En el ejemplo de la sección anterior la primera regla corresponde al programa ejecutable, que es el objetivo global.

6.4 Variables

En un makefile se pueden utilizar variables de manera similar a los lenguajes de programación imperativos. Las variables permiten almacenar valores de texto. Las variables no necesitan ser declaradas. Se crean automáticamente cuando se les asigna valor por primera vez. La asignación de valor a una variable se hace mediante la sentencia:

```
variable = valor
```

Estas sentencias aparecen fuera de las reglas de construcción de objetivos. Las variables pueden usarse en cualquier parte del *makefile* (dependencias, acciones, o asignación de variables), usando la notación:

```
... $(variable) ...
```

6.5 Reglas implícitas

Son plantillas genérica de reglas para reconstruir determinados tipos de ficheros. La línea de dependencias de la regla hace referencias a los tipos de ficheros por la extensión del nombre. El formato habitual es:

```
%.ext1: %.ext2
```

Esta notación corresponde a que un fichero objetivo que tenga la extensión `.ext1` depende de otro fichero con el mismo nombre y la extensión `.ext2`

Las líneas de acciones pueden contener códigos especiales (*macros*) para hacer referencia a determinados fragmentos de información que varían al aplicar la regla:

- `$@` - objetivo
- `$<` - primera dependencia
- `$?` - dependencias modificadas
- `$^` - dependencias (todas)
- `$*` - lo que se ajusta al patrón `%`

Las reglas se invocan con una línea de dependencias que se ajuste al patrón de dependencias. Sólo se escriben las dependencias, pero no las acciones, que son suministradas por la regla implícita. Lo que sí se puede hacer es indicar dependencias adicionales, además de las que se ajustan al patrón de dependencias de la regla implícita.

6.6 Ejemplo de *makefile* con variables y reglas implícitas

El *makefile* correspondiente al programa de ejemplo ajustar puede reescribirse utilizando variables y reglas implícitas de la siguiente manera:

```
OBJS = ajustar.o parrafo.o linea.o palabra.o

ajustar: $(OBJS)
   g++ -o ajustar $(OBJS)

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

palabra.o: palabra.cpp palabra.h
linea.o: linea.cpp linea.h palabra.h
parrafo.o: parrafo.cpp parrafo.h linea.h palabra.h
ajustar.o: ajustar.cpp parrafo.h linea.h palabra.h
```

Comparando esta versión con la que se había dado inicialmente se puede observar que:

- Las lista de los nombres de los ficheros objeto aparecía dos veces en la regla inicial. Ahora la lista se escribe sólo una vez, asignando este valor a la variable `OBJS`, que luego se usa dos veces en la regla inicial. De esta manera es más fácil reescribir la regla si cambia el nombre de algún fichero o se añaden más ficheros fuente al programa.
- Las reglas correspondientes a los ficheros objeto eran similares. En lugar de escribir las reglas completas para cada uno de ellos, ahora se define una regla implícita para regenerar un fichero objeto que depende de un fichero fuente `.cpp` con su mismo nombre. Gracias a eso sólo hay que escribir las líneas de dependencias de los ficheros objeto, que como puede verse dependen del fichero fuente correspondiente, entre otros, y las acciones vienen dadas por la aplicación de la regla implícita.

6.7 Objetivos ficticios (`.PHONY`)

Los objetivos reales de `make` son ficheros a generar o regenerar. A veces resulta interesante especificar objetivos que no son realmente ficheros, sino nombres que se les dan a cierta operaciones. Por ejemplo, la siguiente regla serviría para eliminar ficheros temporales (ficheros objeto) producidos durante la compilación.

```
clean:
   rm *.o ...
```

La acción puede ser invocada mediante:

```
make clean
```

y funcionará correctamente mientras no exista un fichero o directorio real que se llame `clean`. Si se quiere que la acción se ejecute aunque exista un fichero con ese nombre bastará declarar el objetivo como ficticio, mediante:

```
.PHONY : clean
clean:
   rm *.o ...
```

6.8 Operaciones frecuentes

Los objetivos ficticios permiten usar *make* como ejecutor o coordinador de operaciones interesantes sobre el proyecto. En determinados ámbitos (organización, empresa, ...) se emplean por convenio nombres fijos para dichas operaciones. Por ejemplo, en los proyecto de software libre GNU de la FSF (Free Software Foundation) y en otras muchas aplicaciones Unix/Linux encontraremos habitualmente reglas `.PHONY` como las siguientes:

```
all:       <--- suele ser el primer objetivo
   compilar y montar todos los ejecutables del proyecto
check:
   ejecutar serie de pruebas
clean:
   eliminar ficheros intermedios de la compilación
install:
   instalar la aplicación ya compilada en directorios estándar
dist:
```

```
   generar archivos de distribución de la aplicación
distclean:
   eliminar los archivos de distribución
etc...
```

Estas operaciones se podrían haber programado como guiones de órdenes (*scripts*) independientes. La ventaja de hacerlo como reglas del *makefile* es que de esta manera se pueden reutilizar listas de ficheros, y tener agrupadas órdenes relacionadas entre sí en un mismo documento, lo cual facilita su mantenimiento

6.9 Otras aplicaciones

- Cualquier situación en que unos ficheros se actualizan a partir de otros que cambian
  - Hacer copias de *backup*
  - Generar formatos finales de documentos
  - Actualizar contenido de sitios web
  - Ejecutar pruebas de programas
  - Regenerar instalaciones de software
  - Borrar ficheros anticuados
  - … etc. …

6.10 Ejemplo: ‘*Backup*’/‘*Mirror*’ incremental

Este ejemplo es poco realista, ya que existen herramientas especializadas para realizar el proceso, pero servirá para ilustrar algunas de las posibilidades de la herramienta *make*. Se trata de mantener actualizada una copia completa de los ficheros de un directorio (en un subdirectorio BACKUP).

- Partiremos de una situación inicial en que se dispone ya de dicha copia actualizada. Por ejemplo:

```
directorio      BACKUP
 |- fichero1     |- fichero1
 |- fichero2     |- fichero2
 `- fichero3     `- fichero3
```

- Supongamos que cambia parte del contenido del directorio,. Por ejemplo, se modifica el `fichero2` y se añade el `ficheroN`:

```
directorio      BACKUP
 |- fichero1     |- fichero1
 |- fichero2     |- fichero2
 |- fichero3     `- fichero3
 |- ...
 `- ficheroN
```

- Para actualizar la copia no hay que volver a copiar todo, sino sólo lo que cambia:

```
directorio      BACKUP
 |- fichero1     |- fichero1
 |- fichero2     |- fichero2
```

```
 |- fichero3     |- fichero3
 |- ...          |- ...
 `- ficheroN     `- ficheroN
```

- Podríamos describir el proceso, de manera bastante rudimentaria, mediante el siguiente *makefile*:

```
BACKUP/fichero1: fichero1
   copy fichero1 BACKUP

BACKUP/fichero2: fichero2
   copy fichero2 BACKUP

. . .
```

  Este *makefile* es bastante inapropiado. Hace falta nombrar (y por tanto conocer de antemano) cada uno de los ficheros que hay en el directorio. Si se añaden ficheros hay que reescribir el *makefile*.

- Una solución mucho más apropiada, particularizada para un determinado grupo de ficheros (fuentes en C), sería la siguiente:

```
copia: *.c *.h
   copy $? BACKUP
   touch copia
```

  Esta solución aprovecha ciertas posibilidades de *make* y alguna herramienta externa básica.

  - Se usa un fichero auxiliar `copia`, vacío de contenido, que sirve para tener la fecha del último *backup*
  - La línea de dependencias usa comodines para indicar que el fichero `copia` depende de todos los demás ficheros a copiar.
  - Recordaremos que la macro $? se ajusta a la colección de dependencias para las que no está al día el objetivo.
  - La utilidad `touch` cambia la fecha de actualización del fichero que se indique, poniendo la fecha de ese momento, sin modificar su contenido.

6.11 Dependencias automáticas

La preparación manual del *makefile* es laboriosa y propensa a errores. Es fácil olvidarse de algunas dependencias, sobre todo las indirectas. Por esta razón resulta interesante disponer de herramientas de ayuda que faciliten en todo o en parte la creación del *makefile* para un proyecto dado.

En concreto, existen herramientas capaces de analizar los ficheros fuente y detectar las dependencias entre ellos y los ficheros objeto que se producirán al compilarlos. Ejemplos de posible alternativas:

- El compilador genera información de dependencias, invocándolo con opciones especiales
- Al análisis de dependencias lo hace una herramienta idepediente del compilador

6.12 Herramienta *makedepend*

Una de las primeras herramientas para análisis de dependencias fue *makedepend* para el lenguaje C. Esta utilidad:

- Invoca el preprocesador y analiza las directivas `#include`. A partir de ellas detecta las dependencias indirectas.
- Añade las líneas de dependencias correspondientes al final del *makefile*, precedidas de una marca especial. La organización del makefile resultante es:

```
... parte del makefile preparada manualmente ...
# DO NOT DELETE THIS LINE -- make depend depends on it.
... líneas de dependencias generadas por makedepend ...
```

- Esta marca permite regenerar (reescribir) las dependencias, analizando de nuevo los ficheros fuente después de ser modificados, pero conservando el resto del *makefile*.

6.13 Compilador *gcc/g++*

La herramienta *makedepend* ha quedado algo anticuada. Por ejemplo, los compiladores `gcc/g++` para los lenguajes C/C++ tienen una capacidad equivalente a *makedepend* para generar automáticamente las líneas de dependencias. Para ello se pueden invocar de la siguiente manera:

```
gcc -MM ficheros-fuente ...  (e igual para g++)
```

Las líneas de dependencias aparecen escritas por la salida estándar del compilador, que puede ser redirigida para añadirla al *makefile*, si es apropiado.

Como ejemplo, se muestra aquí su aplicación al programa `ajustar` mencionado anteriormente:

```
...> g++ -MM *.cpp
ajustar.o: ajustar.cpp palabra.h parrafo.h linea.h
linea.o: linea.cpp linea.h palabra.h
palabra.o: palabra.cpp palabra.h
parrafo.o: parrafo.cpp parrafo.h palabra.h linea.h
```

6.14 Desventajas de ‘*make*’

La herramienta *make* ha demostrado ser muy útil para automatizar las tareas de reconstrucción o regeneración de ficheros. Sin embargo tiene ciertos inconvenientes y limitaciones que conviene conocer. Entre ellos:

- La preparación del *makefile* es laboriosa. Incluso contando con herramientas para detección de dependencias, hace falta en muchos casos introducir manualmente gran parte de la información: listas de ficheros, opciones de compilación, etc.
- La reconstrucción no es segura, ya que no se detectan cambios hacia atrás en el tiempo. Esto ocurre cuando se reemplaza un fichero deficiente por una versión anterior correcta.
- La reconstrucción puede no ser tan eficiente como cabría esperar, a pesar de que la eficiencia es uno de los objetivos de la herramienta *make*. Por ejemplo, se

reconstruyen ficheros finales a partir de otros intermedios aunque éstos realmente no cambien, como ocurre con los ficheros objeto al recompilar después de cambiar sólo comentarios en el texto fuente.

- Por otra parte, y en relación con lo anterior, las dependencias se establecen e nivel de ficheros, y no de partes de ficheros. Esto hace en algunos casos que se recompile prácticamente toda una aplicación tras modificar ligeramente un fichero de cabecera básico, de bajo nivel.
- Los ficheros *makefile* son poco transportables. En particular las acciones son específicas para cada plataforma: ‘*shell*’, compilador, etc.

7. Otras herramientas de construcción

Hay muchas herramientas similares a ‘*make*’, algunas muy similares y otras más elaboradas. En bastantes casos tratan de subsanar alguna de las deficiencias mencionadas. Entre otras se pueden citar las siguientes:

- **Aap**: es un *make* más abstracto, que usa recetas generales relativamente independientes de la plataforma. Es parte del proyecto A-a-p, que trata de integrar diversas herramientas para desarrollo de software. Por ejemplo, hace referencia a `ccbuild`, que es a su vez una herramienta para analizar código fuente C y generar automáticamente la receta Aap o el *makefile* para generar los ejecutables.
- **Ant**: muy usado con Java, las reglas se representan con marcado XML.
- **Cook**: herramienta de construcción planteada como alternativa a *make*. Usa a su vez una herramienta separada para detectar las dependencias (`c_incl`). Las recetas se describen con una notación tipo Lisp
- **Jam**: muy sencilla de usar. Usa recetas independientes de la plataforma. Incluso puede trabajar sin una receta explícita, analizando directamente el código fuente.
- **SCons**: reconstrucción más segura, basado en Python para describir las acciones.