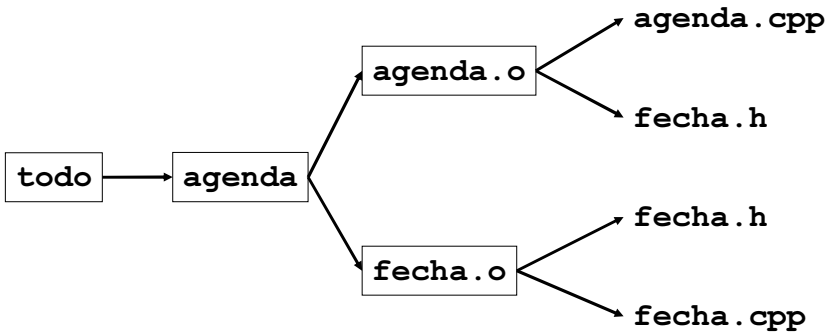
Ejemplo: Funcionamiento

- **Uso 1:**

  prompt$ make



Variables en `Makefile`

- **Es posible simplificar el contenido de los *`Makefiles`* mediante el uso de variables.**
- **Las variables simplifican la escritura y evitan repetir listas de palabras iguales en líneas diferentes.**
- **Facilitan los cambios en el Makefile:**
  - ✓ Ej.: Cambiar el compilador a emplear en todas las reglas.

Formato de las variables

- **Definición de variables:**

  nombre_variable = texto al que sustituye

  ✓ Ejemplo:

  ```
  CXX = g++
  ```

- **Referencia a variables (uso):**

  $(*nombre_variable*)

  ✓ Ejemplo:

  ```
  $(CXX) -c fecha.cpp -Wall
  ```

Ejemplo Makefile con Variables

Variables

```
objs = agenda.o fecha.o
CXX = g++
todo: agenda
fecha.o: fecha.h fecha.cpp
   $(CXX) -c fecha.cpp -Wall
agenda.o: agenda.cpp fecha.h
   $(CXX) -c agenda.cpp -Wall
agenda: $(objs)
   $(CXX) $(objs) -o agenda
limpiar:
   rm $(objs)
   rm agenda
```

2ª versión

```
todo: agenda
fecha.o: fecha.h fecha.cpp
   g++ -c fecha.cpp -Wall
agenda.o: agenda.cpp fecha.h
   g++ -c agenda.cpp -Wall
agenda: agenda.o fecha.o
   g++ agenda.o fecha.o -o agenda
limpiar:
   rm agenda.o
   rm fecha.o
   rm agenda
```

1ª versión

Reglas implícitas

- **`make` posee reglas implícitas (predefinidas) para generar archivos que se utilizan habitualmente como etiquetas.**
  - ✓ Ejemplo: Generar archivos `*.o` a partir de archivos en *C++* (`*.cpp`) o en *C* (`*.c`).
- **Si las reglas implícitas de `make` son suficientes para nuestros propósitos no hace falta definir nuevas reglas para generar archivos.**

Reglas implícitas para compilar *C*

- **El archivo `'n.o'` se genera automáticamente a partir del archvio `'n.c'` con una regla de la forma:**

```
$(CC) -c $(CPPFLAGS) $(CFLAGS)
```

- **Compilador de *C***
- **Opciones adicionales para el preprocesador de *C***
- **Opciones adicionales para el compilador de *C***

Reglas implícitas para compilar *C++*

- El archivo `'n.o'` se genera automáticamente a partir del archvio `'n.cpp'` con una regla de la forma:

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

- `$(CXX)`: Compilador de *C++*
- `$(CPPFLAGS)`: Opciones adicionales para el preprocesador de *C*
- `$(CXXFLAGS)`: Opciones adicionales para el compilador de *C++*

Ejemplo `Makefile` con reglas implícitas

```
objs = agenda.o fecha.o
CXX = g++
CXXFLAGS = -Wall

todo: agenda

fecha.o: fecha.h fecha.cpp

agenda.o: agenda.cpp fecha.h

agenda: $(objs)
   $(CXX) $(objs) -o agenda

limpiar:
   rm $(objs)
   rm agenda
```

Regla implícita, no hace falta indicarlo

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

```
g++ -c -Wall fecha.cpp -o fecha.o
```

Ejemplo `Makefile` con reglas implícitas (2)

```
objs = agenda.o fecha.o
CXX = g++
CXXFLAGS = -Wall

todo: agenda

fecha.o: fecha.h fecha.cpp

agenda.o: agenda.cpp fecha.h

agenda: $(objs)
   $(CXX) $(objs) -o agenda

limpiar:
   rm $(objs)
   rm agenda
```

La dependencia con el archivo fuente también es implícita, no hace falta indicarla

Ejemplo `Makefile` con reglas implícitas (3)

```
objs = agenda.o fecha.o
CXX = g++
CXXFLAGS = -Wall

todo: agenda

fecha.o: fecha.h

agenda.o: fecha.h

agenda: $(objs)
   $(CXX) $(objs) -o agenda

limpiar:
   rm $(objs)
   rm agenda
```

Ahora tienen las mismas dependencias, es posible agruparlas

Ejemplo `Makefile` con reglas implícitas (y 4)

```
objs = agenda.o fecha.o
CXX = g++
CXXFLAGS = -Wall

todo: agenda

$(objs): fecha.h

agenda: $(objs)
   $(CXX) $(objs) -o agenda

limpiar:
   rm $(objs)
   rm agenda
```

Etiquetas “falsas” (phony targets)

- Una etiqueta falsa (phony) es aquella que no es realmente el nombre de un archivo.
  - ✓ Ejemplo:

```
limpiar:
        rm $(objs)
        rm agenda
```

- Problema: ¿qué ocurre si existe un archivo con el nombre de la etiqueta?

Etiquetas “falsas” (phony targets)

- **Problema: ¿qué ocurre si existe un archivo con el nombre de la etiqueta?**
  - ✓ Como la etiqueta no tiene dependencias asume que el archivo siempre está actualizado y no hace nada.
- **Solución: declarar explícitamente la etiqueta como falsa para que `make` no compruebe la existencia de un archivo con ese nombre:**

```
.PHONY: limpiar
limpiar:
        rm $(objs)
        rm agenda
```

Ejemplo `Makefile`

Comentario

```
# makefile para compilar el proyecto agenda
objs = agenda.o fecha.o
CXX = g++
CXXFLAGS = -Wall

todo: agenda

fecha.o: fecha.h

agenda.o: fecha.h

agenda: $(objs)
   $(CXX) $(objs) -o agenda

.PHONY: limpiar
limpiar:
   rm $(objs)
   rm agenda
```

Caracteres especiales

'@': suprime el eco por pantalla de la orden que se está ejecutando.

Ejemplo:

```
@rm $(objs)
```

'–': permite que continúe el procesamiento del makefile aunque la orden que se está ejecutando falle.

Ejemplo:

```
limpiar:
	-rm $(objs)
	-rm agenda
```

Resumen: Contenido de un `Makefile`

- Reglas explícitas
- Reglas implícitas
- Definición de variables
- Directivas
- Comentarios

Resumen: Contenido de un `Makefile` (2)

- **Reglas explícitas:**
  - ✓ Indican cómo y cuándo reconstruir ciertos archivos según órdenes que aparecen en la regla.
- **Reglas implícitas:**
  - ✓ Indican cómo y cuándo reconstruir ciertos archivos en base a su nombre y/o tipo. Las órdenes no aparecen en la regla.

Resumen: Contenido de un `Makefile` (y 3)

- **Definición de variables:**
  - ✓ Asocian cadenas de texto a identificadores de variable.
- **Directivas:**
  - ✓ Órdenes para realizar tareas especiales, tales como incluir otros archivos o decidir que partes del `makefile` deben o no ignorarse.
- **Comentarios:**
  - ✓ Comienzan con '#' y se pueden encontrar en cualquier parte de una línea.

Caracteres comodín (wildcards)

- **Es posible emplear caracteres comodín para hacer referencias a nombres de archivos:**
  - `*` = "cualquier cadena de caracteres"
  - `?` = "un carácter"
- **El carácter comodín se puede escapar con `'\'`.**
- **Ejemplos: Definición de variables**

```
limpiar:
    rm *.o agenda
```

```
objs = *.o
```

✓ No se expande, es una asignación literal.

Caracteres comodín (wildcards) (2)

- **Uso de variables:**

```
$(CXX) $(objs) -o agenda
```

✓ Ahora `*` sí se expande

✓ Si no encaja con nada toma el valor "`*.o`" → ¿problema?

- Para expandir en la definición de una variable usar la función `wildcard`:

```
objs = $(wildcard *.o)
```

✓ **Ojo: si no existen en el directorio archivos `*.o`, no se asigna nada.**

Reglas de patrones estáticos

- **Son reglas que especifican múltiples objetivos y construyen el nombre de las dependencias para cada objetivo basándose en su nombre (parecidas a reglas implícitas pero más flexibles).**
  - ✓ P.ej.: explicar que todos los objetivos del tipo "`nombre.o`" dependen de un archivo fuente del tipo "`nombre.cc`".

- **Sintaxis:**

```
<etiqueta>: <patrón_etiqueta>: <patrón_dependencia>
```

  - ✓ Ejemplo:

```
objs = fecha.o agenda.o
agenda: $(objs)
$(objs): %.o: %.cc
        $(CXX) -c ........
```

Variables automáticas

- **Variables de `make` que toman un valor diferente en función de la regla que se esté ejecutando.**

$@	**Nombre del archivo de la etiqueta de la regla (de cualquiera de las etiquetas si la regla es múltiple)**
$<	**Nombre de la primera dependencia.**
$?	**Nombre de todas las dependencia más nuevas que la etiqueta.**
$^	**Nombre de todas las dependencias**
$*	**La raíz que encaja con un patrón.**

Reglas de patrones estáticos + Var. Automáticas

- **Ejemplo:**

```
objs = fecha.o agenda.o
agenda: $(objs)
$(objs): %.o: %.cc
	$(CXX) -c $(CXXFLAGS) $< -o $@

.PHONY: limpiar
limpiar:
	rm *.o
	agenda
```

Ejercicio (Set. 02)

- **Supongamos que tenemos los siguiente ficheros en sus correspondientes directorios:**

`$HOME/SRC/demo1.cpp`

`$HOME/SRC/demo2.cpp`

`$HOME/SRC/demo1.h`

`$HOME/SRC/demo2.h`

`$HOME/SRC/util/general.h` (usado por ambos fuentes)

`$HOME/SRC/util/general.cpp`

Escribir dos `makefiles` que generen los ejecutables `demo1` y `demo2`, el primero, y el objeto `general.o`, el segundo.

Ejercicio (Juny 02)

- **Supongamos que tenemos los ficheros siguientes:**

`$HOME/SRC/ej1/fuente1.cpp` (permite generar `fuente1.o`)
`$HOME/SRC/ej1/fuente2.cpp` (permite generar `fuente2.o`)
`$HOME/SRC/ej1/cab1.h`
`$HOME/SRC/ej1/cab2.h`
`$HOME/SRC/ej1/lib/libcab.h` (usado por ambas fuentes).
`$HOME/SRC/ej1/lib/libfuente.cpp` (

Escribir los `makefiles` que generen los ejecutables `fuente1` y `fuente2`