Clase #09 de 27
Make & Makefiles

Jun 3, Martes
Jun 4, Miércoles

Agenda para esta clase

- Introducción a Make & Makefiles
  - ¿Qué es make y para qué sirve?
  - ¿Qué es un makefile? ¿Qué tiene?
  - Hello.c, pero ahora con Make
  - FahrCel, pero ahora con Make
  - Trabajo #2: Temperatura Make
- Aplicación en Trabajo #2: Interfaces & Makefile —Temperaturas

Make

Automatización del Proceso de Traducción

https://josemariasola.wordpress.com/ssl/papers/#Make

Introducción a make

Un proyecto informático con aplicación práctica en la industria, que sea que más un mínimo proyecto ejemplificativo, está formado por *muchos* archivos fuente. La compilación de estos proyectos se requiere escribir varios comandos y gestionar las dependencias entre los fuentes. El volumen de archivos en el proyecto es entonces un problema a tratar. Para traer un poco de perspectiva, la *calculadora de Windows* está formada por 500 archivos fuente, *Chrome* por 20 mil, *Linux* por 100 mil, *gcc* por 120 mil, *clang* por 150 mil, *Chromium* por 200 mil, *WebKit* por 330 mil, y el premio al repositorio más grande del mundo se lo lleva *Windows* con 3,5 millones de archivos fuente.

Claramente, la gestión manual no es eficiente para esta escala. La utilidad `make` junto con los *makefiles* proponen una solución. En este texto vamos a ver para qué es y como se usa la utilidad `make`.

1.1. ¿Qué es *make*?

Es una herramienta que determina automáticamete que partes de un programa o sistema grande formado por varios componentes necesitan recomplilarse, y emite los comandos para hacerlo. Efectivamente *hace* o *fabrica* (i.e., *makes*) el programa.

1.2. ¿Qué facilita *make*?

La actualización automática de archivos desde otros archivos, que se disparan cuando los segundos se modifican. Automatiza el proceso de *building* (i.e., "*buildeo*" o traducción) de un programa o sistema grande, formado por varios archivos. Permite actualizar solo lo que cambió, sin necesidad de recompilar todos los archivos fuente que componen el programa.

1.4. ¿Qué sintaxis tiene un *makefile*? ¿Qué partes tiene una regla?

Por convención, al *makefile* de un proyecto se lo nombra `Makefile`, con la `M` Del punto de vista más fundamental, un *makefile*, es una secuencia de reglas. mayúscula para aprovechar el ordenamiento de los archivos y sin extensión. Las reglas son la estructura principal, y esta es su sintaxis:

```
objetivo ... : prerequisitos ...
    comandos
    ...
    ...
```

Los comandos deben estar precedidos por exactamente un caracter tabulado, y no por espacios.

Un mismo archivo *makefile* puede tener múltiples reglas.

1.5. ¿Qué significan las regla de *make*? ¿Cuál es su semántica?

Las reglas tienen el siguiente significado:

- que el resultado *objetivo* depende de los *prerequisitos*,
- que el objetivo se produce siguiendo la receta formada por *comandos*.
- que si los *prerequisitos* están más actualizados que los *objetivos* que porducen, se vuelven a generar los *objetivos*.

Las reglas se evalúan ejecutando el programa `make`, y el resultado va a ser la *fabricación* (i.e., `make`) de los objetivos.

La utilidad `make` lee las dependencias declaradas en el *makefile* y determina que componentes de la solución fueron actualizados desde la última vez que se construyó el producto, `make` reconstruye solo los componentes que fueron actualizados y reconstruye el producto.

Ejercicio 4. Suponga el siguiente *Makefile*:

```
A.o: A.c A.h
    cc -c A.c -o A.o
```

1. ¿Qué ocurre si corremos `make` con ese *Makefile*?
2. ¿Qué ocurre si lo corremos de nuevo?
3. ¿Qué ocurre si hacemos una modificación en la interfaz (`A.h`) y ejecutamos `make` de nuevo?

2.2.1. Simple

```
hello: hello.o
 cc hello.o -o hello

hello.o: hello.c
 cc -c hello.c -o hello.o
```

2.2.2. Phonies

```
.PHONY: run clean

run: hello
 ./hello

clean:
 rm hello hello.o

hello: hello.o
 cc hello.o -o hello

hello.o: hello.c
 cc -c hello.c -o hello.o
```

2.2.3. Recetas por Defecto

```
.PHONY: run clean

run: hello
 ./hello

clean:
 rm hello hello.o

hello: hello.o

hello.o: hello.c
```

2.2.4. Reglas por Defecto

```
.PHONY: run clean

run: hello
 ./hello

clean:
 rm hello hello.o
```

2.2.5. Sin Makefile

Como `make` sabe como generar ejecutables a partir de fuentes, pasando por objetos, es posible no tener un *makefile* y simplemente usar el comando `make hello`. Esto solo funciona para programa muy simples, casi triviales, no para sistemas con varios módulos y dependencias. Esa situación se trata en el siguiente caso.

Caso: Conversión de Temperaturas

```
FahrCel : FahrCel.o Conversion.o
 cc FahrCel.o Conversion.o -o FahrCel

FahrCel.o : FahrCel.c Conversion.h
 cc -std=c23 -c FahrCel.c -o FahrCel.o

Conversion.o: Conversion.h Conversion.c
 cc -std=c23 -c Conversion.c -o Conversion.o

.PHONY : run clean

run : FahrCel
 ./FahrCel

clean :
 rm -f FahrCel.o Conversion.o FahrCel
```

Trabajo #2

Interfaces & Makefile — Temperaturas

Este trabajo está basado en los ejercicios 1-4 y 1-15 de [KR1988] y aplica los conceptos presentados en [Interfaces-Make]:

> 1-4. Escriba un programa para imprimir la tabla correspondiente de Celsius a Fahrenheit

> 1-15. Reescriba el programa de conversión de temperatura de la sección 1.2 para que use una función de conversión.

6.1. Objetivos

- Aplicar el uso de interfaces y módulos.
- Construir un programa formado por más de una unidad de traducción.
- Comprender el proceso de traducción o *Build* cuando intervienen varios archivos fuente.
- Aplicar el uso de `Makefile`.

Términos de la clase #09

Definir cada término con la bibliografía

- Make & Makefiles
  - Make
  - Makefile
  - Objetivo
  - Prerequsitos
  - Receta

Tareas para la próxima clase

1. Resolver Trabajo #2 (En equipo): §6. Interfaces & Makefile — Temperaturas

¿Consultas?

Fin de la clase