Compilación de grandes proyectos

Desarrollo Colaborativo de Aplicaciones

Introducción

Cómo compilamos *software*

Hasta ahora, hemos compilado nuestros proyectos en C y C++ a través de comandos.

Cómo compilamos *software*

Hasta ahora, hemos compilado nuestros proyectos en C y C++ a través de comandos.

Primero generamos los objetos individualizados de nuestros ficheros (**compilación**).

```
1 g++ -c src/<ficheros>.cpp -I ...
```

Cómo compilamos *software*

Hasta ahora, hemos compilado nuestros proyectos en C y C++ a través de comandos.

Después enlazamos las llamadas entre objetos y librerías para generar un único ejecutable (**enlazado**).

```
1 g++ -o game src/<ficheros>.o -L ... -llibreria
```

Cómo compilamos *software*

Compilar a mano en un proyecto pequeño en C++ es asumible, aunque pesado.

Cómo compilamos *software*

Compilar a mano en un proyecto pequeño en C++ es asumible, aunque pesado.

Sin embargo… ¿qué pasa cuando tenemos doscientos ficheros?

Una vía de escape...

Para solucionarlo, podemos generar un único *shell script* que ejecutaremos cada vez que realicemos cambios en el proyecto.

```
1
2 g++ -o game src/*.cpp -I src/ -I <librerias> -L <librerias>
```

Un problema de escalado

Esta solución **recompila todo el proyecto** independientemente de los cambios que hayamos realizado.

Un problema de escalado

Esta solución **recompila todo el proyecto** independientemente de los cambios que hayamos realizado.

En proyectos pequeños, no notamos este detalle…

Un problema de escalado

Esta solución **recompila todo el proyecto** independientemente de los cambios que hayamos realizado.

En proyectos pequeños, no notamos este detalle…

¿Pero y en los grandes?

Un problema de escalado

Esta solución **recompila todo el proyecto** independientemente de los cambios que hayamos realizado.

En proyectos pequeños, no notamos este detalle…

¿Pero y en los grandes?

> En los proyectos grandes, esta política genera **cuello de botella notable**.

Un problema de escalado

Este motivo es lo que justifica la aparición de los **entornos de desarrollo**

- Ecosistemas de compilación (Make, Ninja, CMake, etc)
- IDEs (Visual Studio, CLion)

Capas y ecosistema de compilación

Existe una **jerarquía** de capas para la compilación y la producción de proyectos *software*.

Capas y ecosistema de compilación

Existe una **jerarquía** de capas para la compilación y la producción de proyectos *software*.

Capa 0: Compiladores

- Traducen el código fuente a objetos y ejecutables.
- `g++`, `clang`, `msvc`.

Capas y ecosistema de compilación

Existe una **jerarquía** de capas para la compilación y la producción de proyectos *software*.

Capa 1: *Build Systems*

- Motores de reglas para compilar un proyecto.
- **Compilación incremental**.
- `make` es el estándar tradicional
- `ninja` es una versión ultrarrápida, centrada en **CI**.

Capas y ecosistema de compilación

Existe una **jerarquía** de capas para la compilación y la producción de proyectos *software*.

Capa 1.5: Optimizadores de compilación

- Mejoras de rendimiento para la compilación de proyectos.
- `ccache / sccache` para realizar un cacheo local de la compilación.
- `distcc / Icecream` para hacer compilación distribuida en red.

Capas y ecosistema de compilación

Existe una **jerarquía** de capas para la compilación y la producción de proyectos *software*.

Capa 2: *Meta-Build Systems* (Tema 6)

- Generadores de ficheros para *Build Systems*.
- Multiplataforma
  - Son capaces de generar ficheros para compilar en varios Sistemas Operativos
- `CMake` es el estándar.

Make

Conceptos fundamentales de `make`

Make es un ***Build System***

Conceptos fundamentales de `make`

Make es un ***Build System***

- Convierte los ficheros fuente en **cualquier objetivo que propongas**.
- Sabe qué comandos se deben ejecutar y en **qué orden**.
- Usa *timestamps* garantizar la **compilación incremental**.

Conceptos fundamentales de `make`

Se ejecuta a través del comando `make` en el terminal.

Conceptos fundamentales de `make`

Se ejecuta a través del comando `make` en el terminal.

En el directorio debe haber un fichero `Makefile`.

Makefiles básicos

Su funcionamiento consiste en establecer una lista de **dependencias**.

Makefiles básicos

Su funcionamiento consiste en establecer una lista de **dependencias**.

- Hay un **fichero fuente** y un **fichero destino**.
  - Si el **fuente cambia** el destino **también** (*timestamps*).
- Especificamos una serie de reglas para poder llegar al **destino**.
  - Las reglas siempre van con **tabulador** delante.

Makefiles básicos

Estructura básica de un makefile

```
1  OBJETIVO: FUENTE
2      REGLA 1
3      REGLA 2
```

Makefiles básicos

Estructura básica de un makefile

```
1 main.o: main.cpp
2     g++ -c main.cpp -I .
```

Makefiles básicos

Definición de múltiples objetivos

```
1 main.o: main.cpp
2     g++ -c main.cpp -I .
3
4 auxiliar.o: auxiliar.cpp
5     g++ -c auxiliar.cpp -I .
```

Makefiles básicos

Definición de múltiples objetivos

```
1 main.o: main.cpp
2     g++ -c main.cpp -I .
3
4 auxiliar.o: auxiliar.cpp
5     g++ -c auxiliar.cpp -I .
```

¿Qué resultado da?

Makefiles básicos

Definición de múltiples objetivos

Un `Makefile` debe tener **estructura de árbol**.

Makefiles básicos

Definición de múltiples objetivos

Un `Makefile` debe tener **estructura de árbol**.

- Una instrucción raíz que tiene varias dependencias.
  - Dichas dependencias son **objetivos**, los cuales tienen otras dependencias.

Makefiles básicos

Definición de múltiples objetivos

Un `Makefile` debe tener **estructura de árbol**.

- Una instrucción raíz que tiene varias dependencias.
  - Dichas dependencias son **objetivos**, los cuales tienen otras dependencias.

> ⚠ **Advertencia**
>
> Por defecto, el comando `make` ejecuta el **primer** objetivo del fichero y asume que éste es el **objetivo raíz**.

Makefiles básicos

Definición de múltiples objetivos

```
1 ejecutable: main.o auxiliar.o
2
3 main.o: main.cpp
4     g++ -c main.cpp -I .
5
6 auxiliar.o: auxiliar.cpp
7     g++ -c auxiliar.cpp -I .
```

Makefiles básicos

Definición de múltiples objetivos

```
1 ejecutable: main.o auxiliar.o
2
3 main.o: main.cpp
4     g++ -c main.cpp -I .
5
6 auxiliar.o: auxiliar.cpp
7     g++ -c auxiliar.cpp -I .
```

Makefiles básicos

Definición de múltiples objetivos

```
1 ejecutable: main.o auxiliar.o
2
3 main.o: main.cpp
4     g++ -c main.cpp -I .
5
6 auxiliar.o: auxiliar.cpp
7     g++ -c auxiliar.cpp -I .
```

Makefiles básicos

Definición de múltiples objetivos

```
1 ejecutable: main.o auxiliar.o
2
3 main.o: main.cpp
4     g++ -c main.cpp -I .
5
6 auxiliar.o: auxiliar.cpp
7     g++ -c auxiliar.cpp -I .
```

¿Qué salida genera este makefile?

Generalizando los makefiles

Así, deberíamos añadir los objetivos uno a uno y actualizar el principal.

Aún siendo viable… ¿es lo mejor?

Generalizando los makefiles

Así, deberíamos añadir los objetivos uno a uno y actualizar el principal.

Aún siendo viable… ¿es lo mejor?

> Los makefiles aceptan **variables** y **macros** para generalizar y simplificar el proceso de compilación.

Generalizando los makefiles

Macros

Símbolos específicos que los makefiles interprentan para obtener información en la regla actual. Algunos conocidos:

- `%` representa el TODO, como en el terminal el `*`.
- `$^` representa la lista de dependencias completa de ese objetivo.
- `$@` representa al objetivo actual.

Generalizando los makefiles

```
1 ejecutable: main.o auxiliar.o
2
3 %.o : %.cpp
4     g++ -o $@ -c $^ -I.
```

Generalizando los makefiles

También se pueden definir variables que usamos en el fichero

```
1 OBJECTS := main.o auxiliar.o
2
3 ejecutable: $(OBJECTS)
4     g++ -o $@ $^ <def_librerias>
5
6 %.o : %.cpp
7     g++ -o $@ -c $^ -I.
```

Generalizando los makefiles

También existen funciones especiales para facilitar el proceso

```
1 SOURCES := $(wildcard *.cpp)
2 OBJECTS := $(subst .cpp,.o,$(SOURCES))
3
4 ejecutable: $(OBJECTS)
5     g++ -o $@ $^ <def_librerias>
6
7 %.o : %.cpp
8     g++ -o $@ -c $^ -I.
```

Múltiples objetivos

```
1  SOURCES := $(wildcard *.cpp)
2  OBJECTS := $(subst .cpp,.o,$(SOURCES))
3
4  ejecutable: $(OBJECTS)
5      g++ -o $@ $^ <def_librerias>
6
7  %.o : %.cpp
8      g++ -o $@ -c $^ -I.
9
10 info:
11     $(info $(SOURCES))
12     $(info $(OBJECTS))
```

Múltiples objetivos

```
1  SOURCES := $(wildcard *.cpp)
2  OBJECTS := $(subst .cpp,.o,$(SOURCES))
3
4  ejecutable: $(OBJECTS)
5      g++ -o $@ $^ <def_librerias>
6
7  %.o : %.cpp
8      g++ -o $@ -c $^ -I.
9
10 info:
11     $(info $(SOURCES))
12     $(info $(OBJECTS))
```

> ⚠ **Advertencia**
>
> Recuerda que, para que se ejecute `info`, debes explicitarlo cuando llames a `make`.

Múltiples objetivos

```
1  SOURCES := $(wildcard *.cpp)
2  OBJECTS := $(subst .cpp,.o,$(SOURCES))
3  .PHONY  := clean
4
5  ejecutable: $(OBJECTS)
6      g++ -o $@ $^ <def_librerias>
7
8  %.o : %.cpp
9      g++ -o $@ -c $^ -I.
10
11 info:
12     $(info $(SOURCES))
13     $(info $(OBJECTS))
14
15 clean:
16     rm *.o
17     rm -rf dist/
```

¿Cuál es la diferencia entre `info` y `clean`?

Objetivos especiales

En un proyecto *software* se esperan, al menos, los siguientes objetivos:

- `all`: Será el primero y representa el objetivo por defecto.
- `clean`: Borra todos los ficheros intermedios que se puedan volver a generar al ejecutar make.
- `dist`: Crea un fichero que contiene la distribución del proyecto.
- `install`: Instala el resultado de crear el proyecto.
- `uninstall`: Desinstala el resultado de crear el proyecto.

Dependencias de orden

Es posible condicionar el orden de ejecución de las instrucciones de un `makefile` mediante dependencias de orden.

Dependencias de orden

Es posible condicionar el orden de ejecución de las instrucciones de un `makefile` mediante dependencias de orden. Se definen a través del símbolo `|`.

Dependencias de orden

Su único propósito es **garantizar que la dependencia existe** antes de ejecutar la regla.

Dependencias de orden

Su único propósito es **garantizar que la dependencia existe** antes de ejecutar la regla.

¿Para qué crees que son útiles?

Dependencias de orden

```
1 build/%.o: %.cpp
2     g++ $(CFLAGS) -c -Iinclude $< -o $@
```

Dependencias de orden

```
1  build/%.o: %.cpp | build
2      g++ $(CFLAGS) -c -Iinclude $< -o $@
3
4  build:
5      mkdir build/
```

Funciones

`make` tiene algunas funciones y utilidades para trabajar con cadenas de caracteres.

Funciones

`make` tiene algunas funciones y utilidades para trabajar con cadenas de caracteres.

Se definen como `$(func arg1,arg2,...)`

Funciones

`make` tiene algunas funciones y utilidades para trabajar con cadenas de caracteres.

Se definen como `$(func arg1,arg2,...)`

> ⚠ **Advertencia**
>
> Las funciones en los *makefiles* son **sensibles a los espacios**, ya que se tratan de utilidades con cadenas de caracteres, por lo que `$(func arg1,arg2)` no devuelve el mismo resultado que `$(func arg1, arg2)`

Ejemplos de funciones

- `wildcard`: Equivalente a usar `*` y `?` en el terminal.
- `info`: Equivalente a hacer `echo` en el terminal.
- `subst`: Reemplazo de cadenas literal.
- `patsubst`: Reemplazo de cadenas por patrones.
- `addprefix`: Añade prefijos a una cadena o lista.
- `shell`: Ejecuta un comando del terminal y devuelve el resultado a una variable.

Makfiles multinivel

Si el proyecto que desarrollamos tiene la entidad suficiente, lo dividiremos en varios módulos, estando el código de cada uno de ellos en un subdirectorio.

Makfiles multinivel

Si el proyecto que desarrollamos tiene la entidad suficiente, lo dividiremos en varios módulos, estando el código de cada uno de ellos en un subdirectorio.

Por ejemplo: `src\`, `vendor\`, `tests\`

Makfiles multinivel

Si el proyecto que desarrollamos tiene la entidad suficiente, lo dividiremos en varios módulos, estando el código de cada uno de ellos en un subdirectorio.

Por ejemplo: `src\`, `vendor\`, `tests\`

> ¿Cómo gestionamos el proceso de compilación cuando incluimos esta complejidad estructural?

Makfiles multinivel

Proyecto de ejemplo

```
proyecto/
├─ src/
│   ├─ main.cpp
│   └─ util.cpp
├─ lib/
│   └─ mathx.cpp
└─ include/
    └─ app.h
```

Estrategia monolítica

One Makefile Rules All

Creamos un único `Makefile` (ubicado en la raíz) que gestiona absolutamente todas las dependencias

Estrategia monolítica

```
proyecto/
├─ Makefile
├─ src/
│  ├─ main.cpp
│  └─ util.cpp
├─ lib/
│  └─ mathx.cpp
└─ include/
   └─ app.h
```

Estrategia monolítica

Debemos descubrir los ficheros fuente y generar sus `.o` correspondientes

```
1 BUILD   := build
2 SRC_DIR := src #Directorio principal
3 SRCS     := $(shell find $(SRC_DIR) -type -f -name '*.cpp')
4 OBJS     := $(patsubst $(SRC_DIR)/%.cpp,$(OBJ_DIR)/%.o,$(SRC))
```

Estrategia monolítica

También hay que guardar las carpetas para actualizar la ruta de includes

```
1 BUILD   := build
2 SRC_DIR := src #Directorio principal
3 SRCS     := $(shell find $(SRC_DIR) -type -f -name '*.cpp')
4 OBJS     := $(patsubst $(SRC_DIR)/%.cpp,$(OBJ_DIR)/%.o,$(SRC))
5 INC_DIRS := $(shell find $(SRC_DIR) -type d)
6 INC_FLAGS := $(addprefix -I,$(INC_DIRS))
```

Estrategia monolítica

Aplicación en compilación

```
1
2 $(OBJ_DIR)/%.o:$(SRC_DIR)/%.cpp:
3     g++ -c $^ -o $@ $(INC_FLAGS)
```

Estrategia monolítica

Ventajas:

- Simple: mantienes un solo `Makefile` por proyecto.
- Mejor manejo de dependencias.
- Paralelizable: `make -j` no rompe las dependencias.

Estrategia recursiva

`make` **permite realizar llamadas recursivas.**

Estrategia recursiva

`make` **permite realizar llamadas recursivas.**

Cada carpeta contiene su propio `Makefile`.

Estrategia recursiva

`make` **permite realizar llamadas recursivas.**

Cada carpeta contiene su propio `Makefile`.

Un `Makefile` coordinador se encarga de llamar a cada uno de ellos.

Estrategia recursiva

```
proyecto/
├─ Makefile
├─ src/
│   ├─ main.cpp
│   ├─ util.cpp
│   └─ Makefile
├─ lib/
│   ├─ Makefile
│   └─ mathx.cpp
└─ include/
    ├─ Makefile
    └─ app.h
```

Estrategia recursiva

Código del `Makefile` coordinador

```
1  SUBDIRS = lib src
2  export BUILD = build
3
4  .PHONY: all $(SUBDIRS)
5
6  all: $/(SUBDIRS)
7      @echo "Proyecto compilado correctamente"
8
9  $(SUBDIRS):
10     $(MAKE) -C $@ all --no-print-directory
```

Estrategia recursiva

Código del `Makefile` de `src` (por ejemplo)

```
1  BIN      = game
2  SRCS     = %.cpp
3  OBJS     := $(patsubst %.cpp,$(BUILD)/src/%.o,$(SRCS))
4
5  .PHONY: all
6
7  all: $(BIN)
8
9  $(BIN): $(LIBSTATIC) $(OBJS)
10     $(CC) $(OBJS) $(LIBSTATIC) -o $@ $(LDFLAGS)
11
12 $(BUILD)/src/%.o: %.c | $(BUILD)/src/
13     $(CC) $(CFLAGS) -c $< -o $@
14
15 $(BUILD)/src/:
16     @mkdir $@
```

Estrategia recursiva

Ventajas:

- `Makefile` más sencillo y mayor limpieza visual.
- Coordinación del proyecto a través de un makefile global.
- Permite compilaciones parciales para *testear* módulos.

¿La recursividad es buena?

Tema polémico en la propia comunidad de C++:

- Peter Miller: *Recursive Make Considered Harmful*. Los `make` recursivos fragmentan el árbol de dependencias y eso lleva a builds incoherentes, largos tiempos de espera o necesidad de hacer make clean con frecuencia.

¿La recursividad es buena?

Tema polémico en la propia comunidad de C++:

- En *StackOverflow* también hay debate:

> Recursive makefiles are bad primarily because you partition your dependency tree into several trees. This prevents dependencies between make instances from being expressed correctly.

¿La recursividad es buena?

Tema polémico en la propia comunidad de C++:

- En *StackOverflow* también hay debate:

> Recursive make was originally meant … as long as there is a dependency across the recursive structure, make will make it very hard to fix the order.

¿La recursividad es buena?

Tema polémico en la propia comunidad de C++:

> Los sistemas de proyectos autoconfigurables como CMake, Meson o Premake han ganado popularidad porque abordan estos problemas: permiten gestionar dependencias completas sin renunciar a modularidad y escalabilidad.

Ejecución paralela

- `make` puede lanzar compilaciones en paralelo.
- Para ello debes emplear la opción `-j` seguida del numero de trabajos en paralelo a lanzar, por ejemplo: `make -j2`
- Lanzar trabajos en paralelo suele descubrir fallos a la hora de especificar las dependencias, no es lo mismo satisfacerlas de forma secuencial que en paralelo.

¿La herramienta `make` es exclusiva de `C` y `C++`?

Caché de compilación

Un gran límite de `make`

`make` garantiza que un fichero no se vuelva a recompilar a través de su *timestamp*.

Un gran límite de `make`

`make` garantiza que un fichero no se vuelva a recompilar a través de su *timestamp*.

Pero ¿qué sucede si **cambio un comentario en un fichero**?

Un gran límite de `make`

`make` garantiza que un fichero no se vuelva a recompilar a través de su *timestamp*.

Pero ¿qué sucede si **cambio un comentario en un fichero**?

`make` compilará dicho fichero, aunque **el resultado sea el mismo**.

Cachear compilaciones

La solución es guardar **compilaciones temporales** (caché) de la compilación del proyecto.

Cachear compilaciones

La solución es guardar **compilaciones temporales** (caché) de la compilación del proyecto.

Si el resultado va a ser el mismo, **devolver el resultado ya obtenido**.

Ccache

Compiler Cache (`ccache`) es una herramienta que nos permite realizar esto.

Ccache

Compiler Cache (`ccache`) es una herramienta que nos permite realizar esto.

Se utiliza como una **herramienta complementaria al compilador**.

```
1 ccache g++ -o app src.cpp
```

Características

- La versión actual de ccache soporta los lenguajes: `C`, `C++`, `Objective-C` y `Objective-C++`.
- Mantiene estadísticas de aciertos/fallos.
- Gestión automática del tamaño de la cache.
- Puede cachear compilaciones con “warnings”.
- Añade una sobrecarga mínima al proceso de compilación.
- Opcionalmente comprime los archivos en la cache para ahorrar sitio.

Uso de `ccache`

Se puede usar como **prefijo**.

```
1 ccache g++ -o app src.cpp
```

Uso de `ccache`

Se puede usar como **prefijo**.

```
1 ccache g+ -c -o app src.cpp
```

O *suplantando* al compilador

```
1 cp ccache /usr/local/bin/
2 ln -s ccache /usr/local/bin/gcc
3 ln -s ccache /usr/local/bin/g++
4 ln -s ccache /usr/local/bin/cc
5 ln -s ccache /usr/local/bin/c++
```

Cómo funciona `ccache`

`ccache` calcula un ***hash*** para identificar **información única** en la compilación

- Contenido del fichero fuente **en compilación**.
- Parámetros del compilador.
- Versión del compilador.

Cómo funciona `ccache`

`ccache` calcula un ***hash*** para identificar **información única** en la compilación

- Contenido del fichero fuente **en compilación**.
- Parámetros del compilador.
- Versión del compilador.

Utiliza el algoritmo `MD4` para codificar dicho *hash*.

Cómo funciona `ccache`

`ccache` calcula un ***hash*** para identificar **información única** en la compilación

Si encuentra un *hash* idéntico en su registro (*hit*): devuelve el resultado sin compilar.

Cómo funciona `ccache`

`ccache` calcula un ***hash*** para identificar **información única** en la compilación

Si encuentra un *hash* idéntico en su registro: devuelve el resultado sin compilar.

Si no encuentra nada (*miss*): compila y guarda el resultado.

Modos de ccache

`ccache` soporta tres modos:

- Directo: El hash se calcula desde el propio fichero.
- Preprocesador: El hash se calcula después de pasar por el preprocesador.
- Dependencias: No se usa nunca el preprocesador.

Modos de ccache

`ccache` soporta tres modos:

- Directo: Rápido, pero más sensible a cambios (dependencias).
- Preprocesador: Robusto, pero más lento.
- Dependencias: Híbrido, pero más exigente en compilación (*manifest de dependencias*).

Inspeccionando la caché

`ccache` acumula sus estadísticas internamente.

Para visualizarlas, llamamos a `ccache -s`.

Inspeccionando la caché

¿Qué nos encontramos?

Inspeccionando la caché

¿Qué nos encontramos?

Hits: Aciertos en la caché.

Inspeccionando la caché

¿Qué nos encontramos?

Hits: Aciertos en la caché.

- *Direct*: *Hashes* encontrados en crudo.
- *Preprocessed*: *Hashes* encontrados tras pasar por el preprocesador.

Inspeccionando la caché

¿Qué nos encontramos?

Miss: Aciertos en la caché.

- *Direct*: No encontrado en el modo directo.
- *Preprocessed*: No encontrado en modo preprocesador.

Inspeccionando la caché

La caché es acumulativa, por lo que se nos va a guardar **todo el historial** de compilaciones con `ccache`.

Inspeccionando la caché

La caché es acumulativa, por lo que se nos va a guardar **todo el historial** de compilaciones con `ccache`.

Para reiniciar la caché, ejecutamos:

```
1 ccache --zero-stats
```

Ninja

Los makefiles están bien pero...

- Los proyectos grandes generan **builds lentas**.
- Ni siquiera con la compilación paralela es suficiente
- Necesitamos **una herramienta más rápida**.

¿Qué es Ninja?

Ninja es un sistememma de construcción y compilación enfocado en la **velocidad**.

¿Qué es Ninja?

Ninja es un sistemema de construcción y compilación enfocado en la **velocidad**.

- Minimalista: tiene menos funciones y mayor rendimiento.
- *Do less, but do it quick.*
- No reemplaza a `make`, pero se usa **cada vez más** con *Meta-Build Systems* como CMake, Meson, etc.

Ninja básico

En **Make** escribes reglas y dependencias en un `Makefile`. El motor ya decide qué hacer y cómo ejecutarse.

Ninja básico

En **Make** escribes reglas y dependencias en un `Makefile`. El motor ya decide qué hacer y cómo ejecutarse.

En **Ninja** defines un **grafo de acciones mínimo** en `build.ninja`, el cual se ejecuta en el motor en paralelo.

Ninja básico

En **Make** escribes reglas y dependencias en un `Makefile`. El motor ya decide qué hacer y cómo ejecutarse.

En **Ninja** defines un **grafo de acciones mínimo** en `build.ninja`, el cual se ejecuta en el motor en paralelo.

> Piensa en Ninja como *el backend de compilación*: recibe un DAG muy explícito y lo ejecuta a la máxima velocidad.

De make a ninja

```
1  app: main.o
2      gcc main.o -o app
3
4  main.o: main.c
5      gcc -c main.c -o main.o
```

De make a ninja

```
app: main.o
    g++ main.o -o app

main.o: main.c
    g++ -c main.cpp -o main.o
```

```
rule cc
    command = g++ -c $in -o $out

rule link
    command = g++ $in -o $out

build main.o : cc main.cpp
build app: link main.o
default app
```

De `make` a `ninja`

- Las entradas `$in` y las salidas `$out` siempre tienen un ámbito **local** a la regla.
- Objetivo: `build salida: regla entradas`.
- Receta: `regla command = ...`

De `make` a `ninja`

La declaración de objetivos sin fichero asociado `.PHONY`

```
1 .PHONY clean
2 clean:
3     rm -rf build/
```

```
1 rule removebuild:
2     command = rm -rf buld/
3 build clean: phony removebuild
```

De `make` a `ninja`

Manejo de depenencias.

- Usamos `|` para definir dependencias implícitas.
  - No aparecen en `$in`. Sin embargo, **si cambian provocan recompilación**.

De `make` a `ninja`

Manejo de depenencias.

- Usamos `|` para definir dependencias implícitas.
  - No aparecen en `$in`. Sin embargo, **si cambian provocan recompilación**.
- Usamos `||` para definir dependencias de orden.
  - **Verificación de la existencia**.

De make a ninja

```
1  rule cc
2      command = g++ -c $in -o $out -Iinclude
3
4  rule mkdir
5      command = mkdir -p $out
6
7  build build/dir/: mkdir
8  build build/a.o: cc src/a.cpp | include/config.h || build/dir/
```

De make a ninja

Variables

Al igual que en make, Ninja permite la definición de variables

```
1 cflags = -O2 -Wall
2
3 rule cc
4 command = g++ $cflags -c $in -o $out
5
6 build a.o: cc a.cpp
```

De `make` a `ninja`

Variables

Al igual que en `make`, Ninja permite la definición de variables

```
1 cflags = -O2 -Wall
2
3 rule cc
4 command = g++ $cflags -c $in -o $out
5
6 build a.o: cc a.cpp
7     cflags = -O0
```

Es posible modificar las variables de forma local por regla

Multidirectorio con Ninja

Ninja permite realizar llamadas a otros ficheros `build.ninja` de forma explicita con `subninja`.

Multidirectorio con Ninja

Ninja permite realizar llamadas a otros ficheros `build.ninja` de forma explicita con `subninja`.

```
proj/
├─ app/
│  ├─ main.cpp
│  └─ build.ninja
└─ libmath/
   ├─ math.cpp
   └─ build.ninja
```

Multidirectorio con Ninja

Ninja permite realizar llamadas a otros ficheros `build.ninja` de forma explicita con `subninja`.

```
1 include toolchains.ninja #Necesario para ejecutar subninja
2 subninja libmath/build.ninja
3 subninja app/build.ninja
4
5 default all
```

Multidirectorio con Ninja

Ninja permite realizar llamadas a otros ficheros `build.ninja` de forma explicita con `subninja`.

```
1 rule cc
2     command = g++ -c $in -o $out
3 rule link
4     command = g++ -o $out $in
5
6 build math.o: cc math.c
7 build all: phony libmath.o
```

¿Merece la pena implementar Ninja a mano?

Según su documentación…

> Make's language was designed to be written by humans. Many projects find make alone adequate for their build problems. In contrast, Ninja has almost no features; just those necessary to get builds correct while punting most complexity to generation of the ninja input files.

¿Merece la pena implementar Ninja a mano?

Según su documentación…

> Ninja by itself is unlikely to be useful for most projects.

¿Merece la pena implementar Ninja a mano?

> ⚠ **Advertencia**
>
> Esto no significa que **Ninja sea inútil**. Significa que, *per se*, esta herramienta es extraño utilizarla por sí sola. Normalmente, se usa en conjunción con *Meta-Build Systems* como CMake.