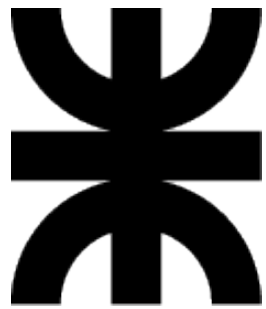
Informática II
Construcción de proyectos con `make`

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2024 –

Contrucción de proyectos con `make` – Necesidad

hola.h

```
1 #ifndef HOLA_H
2 #define HOLA_H
3
4 void hola(const char *nombre);
5
6 #endif
```

hola.c

```
1 #include <stdio.h>
2 #include "hola.h"
3
4 void hola(const char *nombre)
5 {
6   printf("Hola, %s!\n", nombre);
7 }
```

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5   hola("mundo");
6   return 0;
7 }
```

Contrucción de proyectos con `make` – Necesidad

hola.h

```c
#ifndef HOLA_H
#define HOLA_H

void hola(const char *nombre);

#endif
```

hola.c

```c
#include <stdio.h>
#include "hola.h"

void hola(const char *nombre)
{
  printf("Hola, %s!\n", nombre);
}
```

main.c

```c
#include "hola.h"

int main(void)
{
  hola("mundo");
  return 0;
}
```

Compilación

```
gcc -Wall main.c hola.c -o hola
```

Contrucción de proyectos con `make` – Necesidad

hola.h

```c
#ifndef HOLA_H
#define HOLA_H

void hola(const char *nombre);

#endif
```

main.c

```c
#include "hola.h"

int main(void)
{
  hola("mundo");
  return 0;
}
```

hola.c

```c
#include <stdio.h>
#include "hola.h"

void hola(const char *nombre)
{
  printf("Hola, %s!\n", nombre);
}
```

Compilación

```
gcc -Wall main.c hola.c -o hola
```

Se puede compilar separadamente cada archivo fuente

```
gcc -Wall -c main.c
gcc -Wall -c hola.c
```

Contrucción de proyectos con `make` – Necesidad

hola.h

```c
#ifndef HOLA_H
#define HOLA_H

void hola(const char *nombre);

#endif
```

hola.c

```c
#include <stdio.h>
#include "hola.h"

void hola(const char *nombre)
{
  printf("Hola, %s!\n", nombre);
}
```

main.c

```c
#include "hola.h"

int main(void)
{
  hola("mundo");
  return 0;
}
```

Compilación

```
gcc -Wall main.c hola.c -o hola
```

Se puede compilar separadamente cada archivo fuente

```
gcc -Wall -c main.c
gcc -Wall -c hola.c
```

Luego unirlo con el linker

```
gcc main.o hola.o -o hola
```

Contrucción de proyectos con `make` – Necesidad

hola.h

```c
#ifndef HOLA_H
#define HOLA_H

void hola(const char *nombre);

#endif
```

hola.c

```c
#include <stdio.h>
#include "hola.h"

void hola(const char *nombre)
{
  printf("Hola, %s!\n", nombre);
}
```

main.c

```c
#include "hola.h"

int main(void)
{
  hola("mundo");
  return 0;
}
```

Compilación

```
gcc -Wall main.c hola.c -o hola
```

Se puede compilar separadamente cada archivo fuente

```
gcc -Wall -c main.c
gcc -Wall -c hola.c
```

Luego unirlo con el linker

```
gcc main.o hola.o -o hola
```

Permite modifica un archivo fuente y recompilar solo el archivo modificado.

Contrucción de proyectos con `make` – Ejemplo

Construcción con `make`

```
$> make
gcc -c -o hola.o hola.c
gcc -c -o main.o main.c
gcc hola.o main.o -o hola
```

Contrucción de proyectos con `make` – Ejemplo

Construcción con `make`

```
$> make
gcc -c -o hola.o hola.c
gcc -c -o main.o main.c
gcc hola.o main.o -o hola
```

Ejecutar

```
Hola, mundo!
```

Contrucción de proyectos con `make` – Ejemplo

Construcción con `make`

```
$> make
gcc -c -o hola.o hola.c
gcc -c -o main.o main.c
gcc hola.o main.o -o hola
```

Ejecutar

```
Hola, mundo!
```

Modificar el archivo fuente `main.c` y reconstruir

```
$> make
gcc -c -o main.o main.c
gcc hola.o main.o -o hola
```

Contrucción de proyectos con `make` – Ejemplo

Construcción con `make`

```
$> make
gcc -c -o hola.o hola.c
gcc -c -o main.o main.c
gcc hola.o main.o -o hola
```

Ejecutar

```
Hola, mundo!
```

Modificar el archivo fuente `main.c` y reconstruir

```
$> make
gcc -c -o main.o main.c
gcc hola.o main.o -o hola
```

Ejecutar

```
Hola, InfoII!
```

Herramienta `make`

Herramienta para la construcción (re-construcción) de software.

Herramienta `make`

Herramienta para la construcción (re-construcción) de software.

- `make` simplifica el proceso de construcción de proyectos de múltiples archivos fuentes, que generalmente requieren varias llamadas al compilador.

Herramienta `make`

Herramienta para la construcción (re-construcción) de software.

- `make` simplifica el proceso de construcción de proyectos de múltiples archivos fuentes, que generalmente requieren varias llamadas al compilador.

- Automatiza: qué partes construir, cómo construirlas, y cuando.

Herramienta `make`

Herramienta para la construcción (re-construcción) de software.

- `make` simplifica el proceso de construcción de proyectos de múltiples archivos fuentes, que generalmente requieren varias llamadas al compilador.
- Automatiza: qué partes construir, cómo construirlas, y cuando.
- Le permite al programador poder concentrarse en el código.

Herramienta `make`

Herramienta para la construcción (re-construcción) de software.

- `make` simplifica el proceso de construcción de proyectos de múltiples archivos fuentes, que generalmente requieren varias llamadas al compilador.
- Automatiza: qué partes construir, cómo construirlas, y cuando.
- Le permite al programador poder concentrarse en el código.
- `make` minimiza el tiempo de construcción (determina qué archivos cambiaron), además trabaja con dependencias.

Herramienta `make`

Herramienta para la construcción (re-construcción) de software.

- `make` simplifica el proceso de construcción de proyectos de múltiples archivos fuentes, que generalmente requieren varias llamadas al compilador.
- Automatiza: qué partes construir, cómo construirlas, y cuando.
- Le permite al programador poder concentrarse en el código.
- `make` minimiza el tiempo de construcción (determina qué archivos cambiaron), además trabaja con dependencias.

Optimiza el tiempo del ciclo editar-compilar-verificar

Archivo `Makefile`

Un archivo `Makefile` es un archivo de texto que contiene *reglas* que le indican a `make` qué construir y cómo. Una *regla* consiste en:

Archivo `Makefile`

Un archivo `Makefile` es un archivo de texto que contiene *reglas* que le indican a `make` qué construir y cómo. Una *regla* consiste en:

- Un *target*: lo que se debe construir (objetivo)
- Una lista de una o más *dependencias*: archivos necesarios para construir el *target* (prerrequisito)
- Una lista de *comandos* a ejecutar para construir el objetivo (receta)

Archivo `Makefile`

Un archivo `Makefile` es un archivo de texto que contiene *reglas* que le indican a `make` qué construir y cómo. Una *regla* consiste en:

- Un *target*: lo que se debe construir (objetivo)
- Una lista de una o más *dependencias*: archivos necesarios para construir el *target* (prerrequisito)
- Una lista de *comandos* a ejecutar para construir el objetivo (receta)

```
target: dependency dependency [...]
        command
        command
        [...]
```

Ejemplo de regla:

```
main.o: main.c
        gcc -Wall -c main.c
```

Archivo `Makefile`

Un archivo `Makefile` es un archivo de texto que contiene *reglas* que le indican a `make` qué construir y cómo. Una *regla* consiste en:

- Un *target*: lo que se debe construir (objetivo)
- Una lista de una o más *dependencias*: archivos necesarios para construir el *target* (prerrequisito)
- Una lista de *comandos* a ejecutar para construir el objetivo (receta)

```
target: dependency dependency [...]
        command
        command
        [...]
```

Ejemplo de regla:

```
main.o: main.c
        gcc -Wall -c main.c
```

Cuando se ejecuta, `make` busca los archivos `GNUmakefile`, `makefile`, y `Makefile`, en ese orden.

Archivo `Makefile` – reglas implícitas

- `make` tiene muchas reglas por defecto llamdas *reglas implícitas*

Archivo `Makefile` – reglas implícitas

- `make` tiene muchas reglas por defecto llamdas *reglas implícitas*
- Por ejemplo:
  - indicar que los archivos `.o` sean construidos desde archivos `.c`
  - que el binario sea creado enlazando los archivos `.o` juntos

Archivo `Makefile` – reglas implícitas

- `make` tiene muchas reglas por defecto llamdas *reglas implícitas*
- Por ejemplo:
  - indicar que los archivos `.o` sean construidos desde archivos `.c`
  - que el binario sea creado enlazando los archivos `.o` juntos

`Makefile` para el ejemplo del comienzo

```
1 CC=gcc
2 CFLAGS=-Wall
3
4 hola: main.o hola.o
```

Archivo `Makefile` – reglas implícitas

- `make` tiene muchas reglas por defecto llamdas *reglas implícitas*
- Por ejemplo:
  - indicar que los archivos `.o` sean construidos desde archivos `.c`
  - que el binario sea creado enlazando los archivos `.o` juntos

`Makefile` para el ejemplo del comienzo

```
1 CC=gcc
2 CFLAGS=-Wall
3
4 hola: main.o hola.o
```

- Se definen mediantes las variables de `make` (`CC` y `CFLAGS`)

Archivo `Makefile` – reglas implícitas

- `make` tiene muchas reglas por defecto llamdas *reglas implícitas*
- Por ejemplo:
  - indicar que los archivos `.o` sean construidos desde archivos `.c`
  - que el binario sea creado enlazando los archivos `.o` juntos

`Makefile` para el ejemplo del comienzo

```
1 CC=gcc
2 CFLAGS=-Wall
3
4 hola: main.o hola.o
```

- Se definen mediantes las variables de `make` (`CC` y `CFLAGS`)
- Para el lenguaje C (C++)
  - `CC` (`CXX`) es el compilador
  - `CFLAGS` (`CXXFLAGS`) son opciones del compilador

Archivo `Makefile`

Ver Makefiles paso-a-paso.

Variables y comentarios

```
1  # Nombre de la aplicación
2  PROG = myapp
3
4  # Opciones de construcción
5  CC = gcc # Compilador a utilizar
6  INCLUDE = ./inc # Directorio de headers
7  SOURCES = main.c 1.c 2.c 3.c # Archivos fuentes
8  OBJECTS = $(SOURCES:.c=.o) # Archivos objetos
9  CFLAGS = -g -Wall -std=c90 # Opciones de desarrollo
10 #CFLAGS = -Wall -std=c90 # Opciones de release
11
12 all: $(PROG)
13
14 $(PROG): $(OBJECTS)
15   $(CC) -o $(PROG) $(OBJECTS)
16
17 %.o: %.c
18   $(CC) $(CFLAGS) -I$(INCLUDE) -o $@ -c $<
19
20 .PHONY: clean
21 clean:
22   rm -f $(OBJECTS)
```

Makefile para ATmega328

```
1  PROG = blink
2
3  # Características del microcontrolador
4  MMCU = atmega328p
5  F_CPU = 16000000UL
6  # Toolchain
7  CC = avr-gcc
8  OBJCOPY = avr-objcopy
9  CFLAGS = -Wall -mmcu=$(MMCU)
10 # Targets
11 all: $(PROG).elf
12
13 $(PROG).o: $(PROG).c
14     $(CC) $(CFLAGS) -Os -DF_CPU=$(F_CPU) -c -o $(PROG).o $(PROG).c
15
16 $(PROG).elf: $(PROG).o
17     $(CC) $(CFLAGS) $(PROG).o -o $(PROG).elf
18
19 $(PROG).hex: $(PROG).elf
20     $(OBJCOPY) -O ihex -R .eeprom $(PROG).elf $(PROG).hex
21
22 .PHONY: clean
23 clean:
24     rm -f $(PROG).o $(PROG).elf $(PROG).hex
```