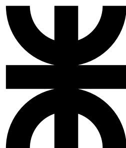
Informática II
Construcción de proyectos con `make`

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2019 –

Contrucción de proyectos con make – Necesidad

main.c

```c
#include "hola.h"

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

Contrucción de proyectos con make – Necesidad

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);
}
```

Contrucción de proyectos con make – Necesidad

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);
}
```

hola.h

```c
void hola(const char *nombre);
```

Contrucción de proyectos con make – Necesidad

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);
}
```

hola.h

```c
void hola(const char *nombre);
```

Compilación

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

Contrucción de proyectos con make – Necesidad

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);
}
```

hola.h

```c
void hola(const char *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

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);
}
```

hola.h

```c
void hola(const char *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
```

y unirlo con el linker

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

Contrucción de proyectos con make – Necesidad

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);
}
```

hola.h

```c
void hola(const char *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
```

y 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

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

(Ver ejemplos de construcción del ejemplo)

Ejemplo de dependencias

```c
/* main.c */
#include "a.h"


. . .
```

```c
/* 2.c */
#include "a.h"
#include "b.h"

. . .
```

```c
/* 3.c */
#include "b.h"
#include "c.h"

. . .
```

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

- Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

- Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- El archivo `3.c` depende del archivo `c.h`

Ejemplo de dependencias

```c
/* main.c */
#include "a.h"


. . .
```

```c
/* 2.c */
#include "a.h"
#include "b.h"

. . .
```

```c
/* 3.c */
#include "b.h"
#include "c.h"

. . .
```

- Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- El archivo `3.c` depende del archivo `c.h`
- Qué pasa si se modifica `b.h` y no se recompila `2.c`

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

- Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- El archivo `3.c` depende del archivo `c.h`
- Qué pasa si se modifica `b.h` y no se recompila `2.c`

Dependencias

```
myapp: main.o 2.o 3.o
```

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

- Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- El archivo `3.c` depende del archivo `c.h`
- Qué pasa si se modifica `b.h` y no se recompila `2.c`

Dependencias

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
```

Ejemplo de dependencias

```c
/* main.c */
#include "a.h"


. . .
```

```c
/* 2.c */
#include "a.h"
#include "b.h"

. . .
```

```c
/* 3.c */
#include "b.h"
#include "c.h"

. . .
```

- Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- El archivo `3.c` depende del archivo `c.h`
- Qué pasa si se modifica `b.h` y no se recompila `2.c`

Dependencias

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
2.o: 2.c a.h b.h
```

Ejemplo de dependencias

```c
/* main.c */
#include "a.h"


. . .
```

```c
/* 2.c */
#include "a.h"
#include "b.h"

. . .
```

```c
/* 3.c */
#include "b.h"
#include "c.h"

. . .
```

- Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- El archivo `3.c` depende del archivo `c.h`
- Qué pasa si se modifica `b.h` y no se recompila `2.c`

Dependencias

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

Construcción de proyectos con make

Makefile para el ejemplo del comienzo

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

Construcción de proyectos con `make`

`Makefile` para el ejemplo del comienzo

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

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

Construcción de proyectos con `make`

`Makefile` para el ejemplo del comienzo

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

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

Construcción de proyectos con `make`

`Makefile` para el ejemplo del comienzo

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

- `make` tiene muchas reglas por defecto llamdas *reglas implícitas*
- Por ejemplo: indicar que los archivos `.o` sean construidos desde archivos `.c`, y que el binario sea creado enlazando los archivos `.o` juntos
- Se definen mediantes las variables de `make` (`CC` y `CFLAGS`)

Construcción de proyectos con `make`

`Makefile` para el ejemplo del comienzo

```
CC=gcc
CFLAGS=-Wall

hola: main.o hola.o
```

- `make` tiene muchas reglas por defecto llamdas *reglas implícitas*
- Por ejemplo: indicar que los archivos `.o` sean construidos desde archivos `.c`, y que el binario sea creado enlazando los archivos `.o` juntos
- Se definen mediantes las variables de `make` (`CC` y `CFLAGS`)
- Para el lenguaje C
  - `CC` es el compilador
  - `CFLAGS` son opciones del compilador

Construcción de proyectos con `make`

`Makefile` para el ejemplo del comienzo

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

- `make` tiene muchas reglas por defecto llamdas *reglas implícitas*
- Por ejemplo: indicar que los archivos `.o` sean construidos desde archivos `.c`, y que el binario sea creado enlazando los archivos `.o` juntos
- Se definen mediantes las variables de `make` (`CC` y `CFLAGS`)
- Para el lenguaje C
  - `CC` es el compilador
  - `CFLAGS` son opciones del compilador
- Para el lenguaje C++
  - `CXX` es el compilador
  - `CXXFLAGS` son opciones del compilador

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

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

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

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

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

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

Archivo Makefile – Otro ejemplo

```
1  editor : editor.o screen.o keyboard.o
2    gcc -o editor editor.o screen.o keyboard.o
3
4  editor.o : editor.c editor.h keyboard.h screen.h
5    gcc -c editor.c
6
7  screen.o : screen.c screen.h
8    gcc -c screen.c
9
10 keyboard.o : keyboard.c keyboard.h
11   gcc -c keyboard.c
12
13 clean :
14   rm editor *.o
```

Archivo `Makefile` – Otro ejemplo

```
1  editor : editor.o screen.o keyboard.o
2    gcc -o editor editor.o screen.o keyboard.o
3
4  editor.o : editor.c editor.h keyboard.h screen.h
5    gcc -c editor.c
6
7  screen.o : screen.c screen.h
8    gcc -c screen.c
9
10 keyboard.o : keyboard.c keyboard.h
11   gcc -c keyboard.c
12
13 clean :
14   rm editor *.o
```

(Tiene 5 targets. Una por defecto.)

Archivo `Makefile` – Otro ejemplo

```
1  editor : editor.o screen.o keyboard.o
2    gcc -o editor editor.o screen.o keyboard.o
3
4  editor.o : editor.c editor.h keyboard.h screen.h
5    gcc -c editor.c
6
7  screen.o : screen.c screen.h
8    gcc -c screen.c
9
10 keyboard.o : keyboard.c keyboard.h
11   gcc -c keyboard.c
12
13 clean :
14   rm editor *.o
```

(Tiene 5 targets. Una por defecto.)

Construir/compilar el proyecto `editor`

```
$ make
```

Archivo `Makefile` – Otro ejemplo

```
1  editor : editor.o screen.o keyboard.o
2    gcc -o editor editor.o screen.o keyboard.o
3
4  editor.o : editor.c editor.h keyboard.h screen.h
5    gcc -c editor.c
6
7  screen.o : screen.c screen.h
8    gcc -c screen.c
9
10 keyboard.o : keyboard.c keyboard.h
11   gcc -c keyboard.c
12
13 clean :
14   rm editor *.o
```

(Tiene 5 targets. Una por defecto.)

Construir/compilar el proyecto `editor`

```
$ make
```

(`$ make clean`)

Comentarios y macros

```
1  all: myapp
2
3  # Which compiler
4  CC = gcc
5
6  # Where are include file kept
7  INCLUDE = .
8
9  # Options for development
10 CFLAGS = -g -Wall -std=c90
11 # Options for release
12 # CFLAGS = -O -Wall -std=c90
13
14 myapp: main.o 2.o 3.o
15   $(CC) -o myapp main.o 2.o 3.o
16
17 main.o: main.c a.h
18   $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c
19
20 2.o: 2.c a.h b.h
21   $(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c
22
23 3.o: 3.c b.h c.h
24   $(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
```

Actividad práctica

Descargar los archivos `Makefile` de ejemplo y recuerde usar la opción `-f` de la aplicación `make` para indicarle el archivo `Makefile` a utilizar. Realizar los siguientes pasos:

1. Copiar el archivo `Makefile1` a `Makefile`, ejecutar `make` y analizar el error

Actividad práctica

Descargar los archivos `Makefile` de ejemplo y recuerde usar la opción `-f` de la aplicación `make` para indicarle el archivo `Makefile` a utilizar. Realizar los siguientes pasos:

1. Copiar el archivo `Makefile1` a `Makefile`, ejecutar `make` y analizar el error
2. Editar `main.c`, `2.c` y `3.c`, como se muestra a continuación, volver a ejecutar `make` y analizar el error

Actividad práctica

Descargar los archivos `Makefile` de ejemplo y recuerde usar la opción `-f` de la aplicación `make` para indicarle el archivo `Makefile` a utilizar. Realizar los siguientes pasos:

1. Copiar el archivo `Makefile1` a `Makefile`, ejecutar `make` y analizar el error
2. Editar `main.c`, `2.c` y `3.c`, como se muestra a continuación, volver a ejecutar `make` y analizar el error
3. Crear archivos header haciendo
   ```
   > touch {a.h,b.h,c.h}
   ```
   y volver a ejecutar `make`

Actividad práctica

Descargar los archivos `Makefile` de ejemplo y recuerde usar la opción `-f` de la aplicación `make` para indicarle el archivo `Makefile` a utilizar. Realizar los siguientes pasos:

1. Copiar el archivo `Makefile1` a `Makefile`, ejecutar `make` y analizar el error
2. Editar `main.c`, `2.c` y `3.c`, como se muestra a continuación, volver a ejecutar `make` y analizar el error
3. Crear archivos header haciendo
   `> touch {a.h,b.h,c.h}`
   y volver a ejecutar `make`
4. `> make -f Makefile1`

Actividad práctica

Descargar los archivos `Makefile` de ejemplo y recuerde usar la opción `-f` de la aplicación `make` para indicarle el archivo `Makefile` a utilizar. Realizar los siguientes pasos:

1. Copiar el archivo `Makefile1` a `Makefile`, ejecutar `make` y analizar el error
2. Editar `main.c`, `2.c` y `3.c`, como se muestra a continuación, volver a ejecutar `make` y analizar el error
3. Crear archivos header haciendo
   `> touch {a.h,b.h,c.h}`
   y volver a ejecutar `make`
4. `> make -f Makefile1`
5. Hacer
   `> touch b.h`
   y volver a ejecutar `make`

Actividad práctica

Descargar los archivos `Makefile` de ejemplo y recuerde usar la opción `-f` de la aplicación `make` para indicarle el archivo `Makefile` a utilizar. Realizar los siguientes pasos:

1. Copiar el archivo `Makefile1` a `Makefile`, ejecutar `make` y analizar el error
2. Editar `main.c`, `2.c` y `3.c`, como se muestra a continuación, volver a ejecutar `make` y analizar el error
3. Crear archivos header haciendo
   `> touch {a.h,b.h,c.h}`
   y volver a ejecutar `make`
4. `> make -f Makefile1`
5. Hacer
   `> touch b.h`
   y volver a ejecutar `make`
6. Eliminar el archivo `2.o`, y probar nuevamente

Actividad práctica

```c
/* main.c */
#include <stdio.h>
#include "a.h"

extern void function_two();
extern void function_three();

int main()
{
  function_two();
  function_three();
  return 0;
}
```

```c
/* 2.c */
#include "a.h"
#include "b.h"

void function_two()
{
}
```

```c
/* 3.c */
#include "b.h"
#include "c.h"

void function_three()
{
}
```

Actividad práctica

Ejemplo de archivo `Makefile` básico

```
myapp: main.o 2.o 3.o
  gcc -o myapp main.o 2.o 3.o

main.o: main.c a.h
  gcc -c main.c

2.o: 2.c a.h b.h
  gcc -c 2.c

3.o: 3.c b.h c.h
  gcc -c 3.c
```

Actividad práctica

Ejemplo de archivo `Makefile` básico

```
1  myapp: main.o 2.o 3.o
2    gcc -o myapp main.o 2.o 3.o
3
4  main.o: main.c a.h
5    gcc -c main.c
6
7  2.o: 2.c a.h b.h
8    gcc -c 2.c
9
10 3.o: 3.c b.h c.h
11   gcc -c 3.c
```

Los demás archivos `Makefile` van agregando:

- Variables `CC`, `CFLAGS`, `INCLUDE`
- Comentarios y target `clean`
- Variables `@` y `<`