
Interfaces y Make

Prof. Ing. José Maria Sola.

Este documento presenta los siguientes conceptos y técnicas fundamentales de la programación en general y de lenguaje C y sus derivados.

- Uso de **readme.md**.
- Construcción de abstracciones.
- Dependencia del cliente con respecto a una interfaz, no a una implementación.
- Guardas de inclusión.
- Proceso de compilación.
- Compilación separada.
- Cadena de compilación.
- Automatización de construcción mediante make.
- Notación declarativa de los makefiles.

1. Interfaces y Abstracciones

1.1. Componentes

Interfaz de la Abstracción o Contrato.

Regla: Depender de la abstracción, no de la implementación.

La interfaz establece el contrato de comunicación entre la abstracción y el usuario de la abstracción.



La relación entre el Cliente y la Interfaz puede describirse como:

- Importa la interfaz.
- Depende de la interfaz.

La relación entre la Implementación y la Interfaz puede describirse como:

- Exporta la interfaz.
- Implementa la interfaz.

Para este caso en particular:



FahrCel.c

```
/* K&R
 * Exercise 1.15. Rewrite the temperature conversion program
 * of Section 1.2 to use a function for conversion.
 * JMS
 * 2016
 */

#include <stdio.h>
#include "Conversion.h"

int main(void){
    const int LOWER = 0;    // lower limit of table
    const int UPPER = 300;  // upper limit
    const int STEP  = 20;   // step size

    for(int fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, GetCelsFromFahr(fahr) );
}
```

Conversion.h

```
/* K&R
 * Exercise 1.15. Rewrite the temperature conversion program
 * of Section 1.2 to use a function for conversion.
 * JMS
 * 2016
 */

#ifndef CONVERSION_H_INCLUDED
#define CONVERSION_H_INCLUDED

double GetCelsFromFahr(double);
```

```
#endif
```

Conversion.c

```
/* K&R
 * Exercise 1.15. Rewrite the temperature conversion program
 * of Section 1.2 to use a function for conversion.
 * JMS
 * 2016
 */

#include "Conversion.h"

double GetCelsFromFahr(double f){
    return (5.0/9.0)*(f-32);
}
```

2. Make

Compilar un proyecto resulta complicado si el proyecto está compuesto por varios archivos y para compilar se requiere escribir comandos extensos.

Los **makefiles** junto con la utilidad **make** proponen una solución.

Un **makefile** es una receta declarativa que indica como construir un producto, la simple invocación a **make** construye ese producto, siempre que la *receta* se encuentra en la carpeta actual.

Para simplificar el proceso la buena práctica es contener los archivos fuente en y en el **makefile** en una misma carpeta.

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 las componentes que fueron actualizadas y reconstruye el producto.

Del punto de vista más fundamental, un makefile, es una secuencia de reglas. Cada regla tiene la siguiente sintaxis:

```
target: prerequisites
[tab]steps
```

Lo semántica de la regla es: Ante la actualización de alguno de los prerequisites, reconstruir el objetivo según los pasos indicados.

Por ejemplo, para el reconocido "Hello, World", el make file es el siguiente:

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

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

Si desde la línea de comando se escribe **make**, se construirá el programa ejecutable **hello**.

Por defecto, make busca en la especificación de construcción un archivo llamado **makefile**. Si se necesita llamarlo de otra manera o se necesita tener más de una especificación, **make** acepta la opción **-f**.

```
make -f othermakefile
```

2.1. Ejemplos

A continuación se presenta un ejemplo simple de make de **makefile** para el famoso **HelloWorld** con un solo archivo fuente, y otro para el programa conversor de temperatura, **FahrCel** que se compone por tres archivos.

makefile para HelloWorld

```
# Makes Hello.exe
# JMS
# 2016

BIN      = hello.exe
OBJ      = hello.o
CC       = gcc
CFLAGS   = -std=c11 -Wall -pedantic-errors -m32 -D __DEBUG__ -g3 $(INCS)
# LDFLAGS = -static-libgcc
INCS     = -I"C:/Program Files/Dev-Cpp/MingW64/x86_64-w64-mingw32/include"
LDLIBS   = -L"C:/Program Files/Dev-Cpp/MingW64/x86_64-w64-mingw32/lib32"
RM       = rm -f

$(BIN): $(OBJ)
    $(CC) $(OBJ) -o $(BIN) $(CFLAGS) $(LDFLAGS) $(LDLIBS)

hello.o: hello.c
```

```
$(CC) -c hello.c -o hello.o $(CFLAGS)

.PHONY: clean
clean:
    $(RM) $(OBJ) $(BIN)
```

makefile para FahrCel

```
# Makes FahrCel.exe
# JMS
# 2016
# K&R Exercise 1.15. Rewrite the temperature conversion program
# of Section 1.2 to use a function for conversion.

BIN      = FahrCel.exe
OBJ      = FahrCel.o Conversion.o
CC       = gcc
CFLAGS   = -std=c11 -Wall -pedantic-errors -m32 -D __DEBUG__ -g3 $(INCS)
# LDFLAGS = -static-libgcc
INCS     = -I"C:/Program Files/Dev-Cpp/MingW64/x86_64-w64-mingw32/include"
LDLIBS   = -L"C:/Program Files/Dev-Cpp/MingW64/x86_64-w64-mingw32/lib32"
RM       = rm -f

$(BIN): $(OBJ)
    $(CC) $(OBJ) -o $(BIN) $(CFLAGS) $(LDFLAGS) $(LDLIBS)

FahrCel.o: FahrCel.c Conversion.h
    $(CC) -c FahrCel.c -o FahrCel.o $(CFLAGS)

Conversion.o: Conversion.h Conversion.c
    $(CC) -c Conversion.c -o Conversion.o $(CFLAGS)

.PHONY: clean
clean:
    $(RM) $(OBJ) $(BIN)
```

Los repositorios para los anteriores ejemplo son:

- <https://github.com/utn-frba-ssl/HelloWorld>
- <https://github.com/utn-frba-ssl/FahrCel>

3. Continuar Leyendo

La utilida **make** y el compilador **gcc** tiene decenas de funcionalidades, esta es solo alguna de las referencias para profundizarlas.

3.1. Bibliografía

- Mrbook's stuff
- cs.colby.edu maketutor
- K&R1988
- <https://gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc.pdf>
- <https://www.gnu.org/software/make/manual/make.pdf>

3.2. Temas relacionados

Makes files genéricos

- <https://github.com/mbcrawfo/GenericMakefile>
- <https://github.com/jimenezrick/magic-makefile>

Make y Pruebas automatizadas

- <http://www.cs.toronto.edu/~penny/teaching/csc444-05f/maketutorial.html>