Clase 25.Makefiles

Makefile

Introducción

Un Makefile es un archivo de texto que indica como se debe compilar y ligar uno o varios programas.

- Primero se leé un archivo de inicialización que sirve para **todos** los makes. Este archivo típicamente se llama make.ini, y se encuentra en el directorio make o mkf.

Introducción

Un Makefile es un archivo de texto que indica como se debe compilar y ligar uno o varios programas.

- Primero se leé un archivo de inicialización que sirve para **todos** los makes. Este archivo típicamente se llama make.ini, y se encuentra en el directorio make o mkf.
- Un makefile tiene instrucciones para un proyecto en específico. El nombre por default de un Makefile es `makefile` pero se puede cambiar.

Introducción

Un Makefile es un archivo de texto que indica como se debe compilar y ligar uno o varios programas.

- Primero se leé un archivo de inicialización que sirve para **todos** los makes. Este archivo típicamente se llama make.ini, y se encuentra en el directorio make o mkf.
- Un makefile tiene instrucciones para un proyecto en específico. El nombre por default de un Makefile es `makefile` pero se puede cambiar.
- Un Makefile está compuesto de: Componentes, líneas de dependencias, directivas, macros, archivos de respuesta, reglas y líneas del shell.

Makefile, sintaxis

Para romper una línea se puede poner el caracter: \

```
 primera parte de la línea \
segunda parte de la línea.
```

Los comentarios comienzan por #, para comentar una sola línea:

```
 # Comentario
```

Makefile, reglas

Las reglas son la parte más importante de los Makefiles, se ven como sigue:

```
 objetivo :  prerequisitos
receta
```

El **objetivo** es, usualmente el nombre de algun archivo que se generará.
Los **prerrequisitos** son archivo que se requieren para construir el **objetivo**.
La **receta** es la acción que llevará a cabo el Makefile, puede ser de una o varias líneas, pero debe de comenzar con un **TAB**, o cambiar el valor de la variable **.RECIPEPREFIX**

Makefile, ejemplo 1

En este ejemplo no se genera ningún archivo solo se especifica un **target/objetivo** y una receta.
Siempre debe de haber un target definido para poder poner una receta.

```
1  #Primer makefile
saluda: #Nombre del target
3         @echo "Hola" #receta, ver el comando comienza
    con @
          @ls  #Otra receta para el mismo target
```

Para ejecutar un archivo que no se llama **Makefile**:
`make -f Makefile0`

Ejecutar: make –help para mas opciones.

Makefile, ejemplo 2

A este tipo de reglas que se les denomina reglas explícitas

```
#Regla, con el nombre del target: ejecutable.o
ejecutable.o: main.cpp #se pide como prerequisito que exista/modifique main.cpp
	g++ -c main.cpp
```

Makefile, ejemplo 3

Cuando el objetivo es un archivo, este se recompila o se vincula (linking) cada vez que un prerequisito cambia, también los prerequisitos a su vez se recompilaran o relinkearan cada que sus prerequisitos cambien.

En este ejemplo **ejecutable** es el objetivo y main.o, hello.o y factorial.o son prerequisitos, make vuelve a vincular **ejecutable** si sus prerequisitos cambian.

```
1

ejecutable: main.o hello.o algo factorial.o functions.h
3       g++ –fopenmp –g main.o hello.o factorial.o –o
    ejecutable
        @echo ”Aplica regla 1, genera ejecutable”
```

Recuerde que tanto prerequisitos y objetivos pueden ser archivos u otras reglas.

Como se procesa

- **make** comienza procesando el primer objetivo. A este se le denomina **objetivo por default.**

Como se procesa

- **make** comienza procesando el primer objetivo. A este se le denomina **objetivo por default.**
- Antes de procesar la receta del primer objetivo make procesa las reglas de las cuales depende esta regla.

Como se procesa

- **make** comienza procesando el primer objetivo. A este se le denomina **objetivo por default.**
- Antes de procesar la receta del primer objetivo make procesa las reglas de las cuales depende esta regla.
- Si no existen reglas para las dependencias y **son archivos** make revisa si la fecha de modificación de los archivos pre-requisito son mas recientes que el archivo generado por la regla actual.

Como se procesa

- **make** comienza procesando el primer objetivo. A este se le denomina **objetivo por default.**
- Antes de procesar la receta del primer objetivo make procesa las reglas de las cuales depende esta regla.
- Si no existen reglas para las dependencias y **son archivos** make revisa si la fecha de modificación de los archivos pre-requisito son mas recientes que el archivo generado por la regla actual.
- Si no existe regla con un nombre del pre-requisito y no es un archivo make aborta con un error.

Ejemplo 5

Note que hay listas de objetivos como variables

```
ejecutable: main.o hello.o algo factorial.o functions.h
	g++ –fopenmp –g  main.o hello.o factorial.o –o
    ejecutable
	@echo ”Aplica regla 1, genera ejecutable”
main.o: main.cpp
	g++ –fopenmp –g –c  main.cpp
	@echo ”Aplica regla 2, genera main.o”
hello.o: hello.cpp
	g++ –fopenmp –g –c  hello.cpp
	@echo ”Aplica regla 3, genera hello.o”
factorial.o: factorial.cpp
	g++ –fopenmp –g –c  factorial.cpp
	@echo ”Aplica regla 4, genera factorial.o”
clean:
	rm –f main.o factorial.o hello.o
```

Makefile, macros, variables

Los macros son reglas de sustitción, utilizadas usualmente para definir variables, como el nombre del compilador y/o las banderas de compilación. Se definen como asignaciones:

```
1 CCOMP= g++
  CLINK=g++
3 FLAGS=-g -c -fopenmp
  imprime:
5         @echo "Compilador= $(CCOMP)"
          @echo "Linkeador= $(CLINK)"
7         @echo "Banderas= $(FLAGS)"
```

Makefile, ejemplo 4

Los macros son reglas de sustitución, utilizadas usualmente para definir variables, como el nombre del compilador y/o las banderas de compilación. Se definen como asignaciones:

```
1 CCOMP= g++
  CLINK=g++
3 FLAGS=-g -c -fopenmp
  ejecutable.o: main.cpp
5         $(CCOMP) $(FLAGS) main.cpp
```

Macros, variables

Makefile

- Tarea 14

```
CCOMP= g++
CLINK=g++
CFLAGS=-g -c -fopenmp
LFLAGS=-g -fopenmp
OBJS=main.o hello.o factorial.o
ejecutable: $(OBJS)  functions.h
	$(CCOMP) $(LFLAGS) $(OBJS) -o ejecutable
	@echo "Aplica regla 1, genera ejecutable"
main.o: main.cpp
	$(CCOMP) $(CFLAGS) main.cpp
	@echo "Aplica regla 2, genera main.o"
hello.o: hello.cpp
	$(CCOMP) $(CFLAGS) hello.cpp
	@echo "Aplica regla 3, genera hello.o"
factorial.o: factorial.cpp
	$(CCOMP) $(CFLAGS) factorial.cpp
	@echo "Aplica regla 4, genera factorial.o"
clean:
	rm -f main.o factorial.o hello.o
```

Reglas implícitas

Make tiene algunas reglas implícitas para compilar archivos ,c, .cpp, .f, etc. En linux los archivos .c son compilados con cc (un vinculo a gcc) y los cpp con g++.

```
CLINK=g++
CFLAGS=-g -c -fopenmp
LFLAGS=-g -fopenmp
OBJS=main.o hello.o factorial.o
CPPS=main.cpp hello.cpp factorial.cpp
ejecutable: $(OBJS)  functions.h
        $(CCOMP) $(LFLAGS) $(OBJS) -o ejecutable
$(OBJS):  $(CPPS) functions.h
```

Reglas para limipar


```
CCOMP= g++
CLINK=g++
CFLAGS=-g -c -fopenmp
LFLAGS=-g -fopenmp
OBJS=main.o hello.o factorial.o
CPPS=main.cpp hello.cpp factorial.cpp
ejecutable: $(OBJS)  functions.h
        $(CCOMP) $(LFLAGS) $(OBJS) -o ejecutable
main.o: main.cpp
        $(CCOMP) $(CFLAGS) main.cpp
hello.o: hello.cpp
        $(CCOMP) $(CFLAGS) hello.cpp
factorial.o: factorial.cpp
        $(CCOMP) $(CFLAGS) factorial.cpp
clean:
        rm -f $(OBJs)
cleanall:
        rm -f $(OBJS) ejecutable
```

Se ejecuta:

```
make clean
```

Macros, Variables especiales

- $@ es el nombre de la regla actual.
- $? Es el nombre de las dependencias que ejecutan en esa regla.
- $< Es el nombre de la dependencia de la regla que invocó la receta actual.
- % es un comodin que reemplaza cualquier palabra con cualquier tipo de caracteres de una función. Ej. %.c son todas las palabras terminan en .c que son devueltas por la función.

Variables especiales

Makefile

- Tarea 14

```
CCOMP= g++
CLINK=g++
LFLAGS=-g
CFLAGS=-g -Wall -c
OBJS=main.o hello.o factorial.o
CPPS=main.cpp hello.cpp factorial.cpp
main: $(OBJS)
	$(CCOMP) $(LFLAGS) $? -o $@
# #Regla para cualquier .o que depende del mismo .cpp
%.o: %.cpp
	$(CCOMP) $(CFLAGS) $<
clean:
	rm -f $(OBJs)
cleanall:
	rm -f $(OBJS) main
```

Funciones

- `$(wildcard *.c)`

  Encuentra todos los .c en el directorio local.

- `$(patsubst %.c,%.o,$(wildcard *.c))`

  Reemplaza en todos los elementos que regresa `$(wildcard *.c)` el sufijo .c por el sufijo .o.

- `$(addprefix $(OBJDIR)/,main.o factorial.o)`

  Concatena un prefijo a la lista del segundo argumento.

Ejemplo final

Makefile

- Tarea 14

```
CCOMP= g++
CLINK=g++
LFLAGS=-g
CFLAGS=-g -Wall -c
EXEC=ejecutable
CPPS:=$(wildcard *.cpp)
OBJS:=$(patsubst %.cpp,%.o,$(CPPS))
OBJDIR:=obj
DIROBJS:=$(addprefix $(OBJDIR)/,$(OBJS) )
$(EXEC): $(DIROBJS) functions.h
        $(CCOMP) $(LFLAGS) -o $@ $(DIROBJS)
$(OBJDIR)/%.o: %.cpp functions.h
        $(CCOMP) $(CFLAGS) $< -o $@
optimized:
        make -f Makefile8 CFLAGS="-O2 -Wall -c"; LFLAGS
    ="-O2 -Wall"
clean:
        rm -f $(DIROBJS) $(EXEC)
```

Tarea 14

14.1

Tomen 4 tareas de las ultimas (solo por si no entregaron alguna la pueden reemplazar con la inmediata anterior), y creen makefiles, el proyecto debe de cumplir los siguientes requisitos:

- El **Makefile** estará en el directorio raíz.
- Los fuentes estarán en un directorio “source” o “src”.
- Los objetos estarán en un directorio separado “obj”.
- Si son varios programas de la misma tarea, el Makefile sirve para compilar todos.
- Si se teclea **make help** se desplegará información acerca de la compilación (cuantos y cuales ejecutables se realizan, cuales son las opciones de compilación).
- Puede haber opciones de: optimización, debuggeo y paralelo, al menos las dos primeras deben de existir.
- El mismo Makefile compila con el comando: latex o pdflatex o pdftex, etc. Un documento que explica muy brevemente el proyecto. Los fuentes de latex estarán en una carpeta **latex** y el pdf compilador en una carpeta **docs**.
- El binario se colocará en una carpeta **bin**.
- La opción **clean** elimina todo lo generado por al compilación, incluyendo la documentación. Además elimina tambien los posibles backups de los fuentes (los que generan los editores).