Manuel Gachs Ballegeer 1º DGIIM Curso 2017/2018

FUNDAMENTOS DEL SOFTWARE: MÓDULO II.

Makefiles

Variables usuales (los nombres pueden cambiar):

Nombre variable	Contenido	Descripción
`CC`	`g++`	Compilador que se va a usar.
`CPPFLAG`	`-Wall`	Opciones del compilador.
`INCLUDE_DIR`	`[dirección]`	Directorio de los archivos de cabecera.
`LIB_DIR`	`[dirección]`	Directorio en el que se encuentran la biblioteca.
`SRCS`	`[archivos.cpp]`	Archivos fuente necesarios para el programa.
`OBJS`	`[archivos.o]`	Archivos objeto necesarios para el programa.
`EXECUTABLE`	`[nombre.out]`	Nombre del programa que crea el makefile.
`LIB_SRCS`	`[archivos.cpp]`	Archivos fuente necesarios para la biblioteca.
`LIB_OBJS`	`[archivos.o]`	Archivos objeto necesarios para la biblioteca.
`LIBNAME`	`[libnombre.a]`	Nombre de la biblioteca que crea el makefile.

Metacaracteres de los makefiles:

Nombre	Descripción
`$@`	Se refiere al objetivo de la regla en la que nos encontramos.
`$^`	Se refiere a todas las dependencias de la regla en la que nos encontramos.
`$<`	Se refiere a la primera dependencia de la regla en la que nos encontramos.
`$?`	Representa las dependencias de la presente regla que hayan sido actualizadas (modificadas) dentro del objetivo de la regla y separadas por un espacio en blanco.

Ejemplos de uso de los metacaracteres.

Ejemplo del uso de `$@`:

```
$(EXECUTABLE):$(OBJS)
    $(CC) $(CPPFLAG) -o $@ $(OBJS)
```

Ejemplo del uso de `$^`:

```
$(EXECUTABLE):$(OBJS) $(LIBNAME)
$(CC) $(CPPFLAG) -L$(LIB_DIR) $(EXECUTABLE) $^
```

Ejemplo del uso de `$<`:

```
$(EXECUTABLE):$(OBJS)
    $(CC) $(CPPFLAG) -o $(EXECUTABLE) $<
```

Ejemplo del uso de `$?`:

```
print:$(SRCS)
    lpr -p $?
```

Con esta orden se imprimirían en pantalla los archivos que hayan sido modificados

Picaresca de los makefiles:

- Las variables objeto se pueden declarar de una forma más rápida, sencilla y que da menos problemas. De esa forma le decimos que se trata de los mismos archivos fuente pero que en vez de acabar en `.cpp` acaban en `.o`. Esto es, `OBJS=$(SRCS:.cpp=.o)` (igual con las variables para las librerías).
- Una forma de ahorrarse problemas es crear la biblioteca en el mismo directorio en el que se está trabajando, esto es, `LIB_DIR=./`.
- En vez de escribir la compilación a archivos objeto uno a uno, se pueden hacer todos a la vez. Una forma de hacerlo es poner como objetivo de la regla `$(OBJS)` y como dependencias `$(SRCS)`.
- `all` se ejecuta por defecto, pero es la primera regla que hay que escribir.

Prefijos de traducción en las órdenes de las reglas:

- `-L`: Se usa en la regla de creación del archivo ejecutable y sirve para incluir en él la biblioteca que se ha creado. Va seguido de `LIB_DIR`.
- `-I`: Se usa para incluir los arhivos de cabecera, es decir, los "includes" de los archivos fuente. Va seguido de `INCLUDE_DIR`, que normalmente suele ser `./includes`.
- `-c`: Se usa para transformar los archivos fuente en archivos objeto.
- `-o`: Se usa para darle un nombre al ejecutable que se crea.
- `-g`: Se usa para realizar un ejecutable preparado para la depuración. Tras él se listan los archivos fuente necesarios.

En un makefile se pueden usar reglas cuyos objetivos no tienen dependencias. Los comandos contenidos en esas reglas siguen la misma sintaxis que el bash. Ejemplos: clean, install, etc.

Ejemplo completo/plantilla de un makefile

```
#Nombre archivo: makefile
#Uso: make -f makefile
#     make -f makefile clean
#     make -f makefile debug
```

```
#     make -f makefile install
#     make -f makefile uninstall
#Descripción: Mantiene todas las dependencias entre los módulos que utiliza programa1. También
#             puede borrar todos los archivos creados por el mismo, crear un ejecutable para
#             ser depurado, ser creado en una carpeta específica y borrarlo de la misma.
#Variables:
CC=g++
CPPFLAG=-Wall

INLUDE_DIR=./includes
LIB_DIR=./
INSTALL_DIR=/tmp

EXECUTABLE=programa1.out
EXECUTABLE_DEBUG=programa1_depurable.out
SRCS=a.cpp b.cpp
OBJS=$(SRCS:.cpp=.o)

LIBNAME=libuno.a
LIB_SRCS=uno.cpp dos.cpp
LIB_OBJS=$(LIB_SRCS:.cpp=.o)

all:$(EXECUTABLE) $(LIBNAME)

$(EXECUTABLE):$(OBJS) $(LIBNAME)
    $(CC) $(CPPFLAGS) -L$(LIB_DIR) -o $@ $^

$(OBJS):$(SRCS)
    $(CC) $(CPPFLAGS) -I$(INCLUDE_LIB) -c $^

$(LIBNAME):$(LIB_OBJS)
    ar -rvs $@ $^

$(LIB_OBJS):$(LIB_SRCS)
    $(CC) $(CPPFLAGS) -I$(INCLUDE_LIB) -c $^

debug: $(EXECUTABLE_DEBUG)

$(EXECUTABLE_DEBUG):$(SRCS) $(LIBNAME)
    $(CC) -g $^ -o $@

clean:
    rm $(EXECUTABLE) $(EXECUTABLE_DEBUG) $(LIBNAME) $(OBJS) $(LIB_OBJS)

install:
    mv $(EXECUTABLE) $(INSTALL_DIR)

uninstall:
    rm $(INSTALL_DIR)/$(EXECUTABLE)
```

GDB

Para depurar un ejecutable, se escribe en la terminal lo siguiente (ten en cuenta que debes haber creado un ejecutable para ser depurado, es decir, con `-g`):

```
$ gdb [nombre del programa]
```

Órdenes del gdb:

Orden	Descripción
`display [variable]`	Muestra el valor de la variable en cada uno de los puntos de ruptura. A cada `display` se le asigna un identificador (número) para poder referenciarlas.
`print [variable]`	Muestra el valor de la variable en el punto de ruptura en el que nos encontramos.
`examine [dirección]`	Examina el contenido de una dirección de memoria.
`show values`	Muestra la historia de valores de las variables impresas.
`p/x $pc`	Muestra el contador de programa usando su dirección lógica.
`x/i $pc`	Muestra la siguiente instrucción que se va a ejecutar usando el contador de programa.
`dissasemble`	Muestra el código ensamblador de la parte en la que estamos trabajando.
`whatis [variable]`	Devuelve el tipo de la variable.
`info locals`	Lista todas las variables locales (puede no funcionar).
`delete display [id]`	Elimina el efecto de la orden `display` sobre una variable, donde `id` representa el valor numérico asociado a la orden display correspondiente.
`list [num], [num2]`	Lista las líneas del código fuente desde `num` hasta `num2`.
`info frame`	Muestra la información del marco actual. Con `down` y `up` podemos subir y bajar en la pila de marcos.
`backtrace full`	Muestra la información referente a las variables locales y el resto de información asociada al marco.
`run`	Ejecuta el programa.
`quit`	Sale del depurador.

Puntos de ruptura:

Para crear un punto de ruptura, solamente hay que escribir el siguiente comando dentro de la gdb:

```
break [expresión] [condición]
```

En el cual:

- `[expresión]`: Expresión puede ser la línea del código fuente en la que quieras crear el punto de ruptura, una función, una variable (en cuyo caso se crearía uno por cada vez que apareceira esa variable), etc.
- `[condición]`: Es opcional, y en ejecución solamente para en ese punto si se cumple la ejecución. La condición se escribe en el lenguaje de programación de los archivos fuente.

Para saber los puntos de ruptura que se encuentran activos, se usa la orden `info breakpoints`.

Paracontinuar la ejecución después de un punto de ruptura, existen dos formas distintas:

1. `next` o `step`: Ejecutan la siguiente instrucción a la del punto de ruptura. Son diferentes en cuanto al tratamiento de los marcos.
2. `continue`: Prosigue la ejecución hasta la siguiente ruptura.

Otras órdenes y guiones de gdb:

Se puede cambiar el valor de una variable, para ello se usa la siguiente orden:

`set variable [variable]=[valor]`

Se pueden hacer guiones para gdb, y su sintaxis es muy simple, pues es simplemente la secuencia de órdenes que se quieren ejecutar en el depurador. Se debe guardar como `archivo.gdb` y para ejecutarlos se usa el siguiente comando:

`$ gdb -x [nombre del archivo] [nombre del programa ejecutable]`

Ejemplo de guión para gdb

```
break 11
break num if num>3
info breakpoints
run
display num
display num2
continue
delete display 1
delete display 2
continue
print resultado
continue
```

Se puede depurar un programa en ejecución en segundo plano, para ello usamos la orden `attach` de la siguiente manera:

1. Ejecutamos un programa en segundo plano: `./programa1 &`
2. Esto nos dará en la pantalla un PID, que es el número largo: `[1] 28944`
3. Entramos en la gdb: `gdb`
4. Escribimos la orden `attach` con el PID: `attach 28944`
5. Con esto entramos en la depuración del programa (no funciona con todos los programas).

También se puede editar el archivo fuente directamente desde el depurador, cambiando previamente la variable `EDITOR` del sistema.

```
$ EDITOR=/usr/bin/gedit #o el programa de editor de texto de tu ordenador.
$ export EDITOR
```

Tras esto, una vez estemos depurando podemos editar una línea (`edit [numero de linea]`) o una función (`edit [nombre de la función]`). Finalmente, podemos usar el bash en la depuración con las siguientes órdenes:

- `shell [orden]` : Realiza una orden de bash,
- `shell` : Entra en el bash. Una vez ahí puedes realizar el número de órdenes de bash que quieras y para salir usa la orden `exit` .