Módulo II

SESIÓN 8: Compilación de programas

Generar el ejecutable

.cpp -> .o

```
$ g++ -c <archivo>.cpp
```

Se genera, con el mismo nombre:

```
<archivo>.o
```

.o -> .out

```
$ g++ <archivo1>.o <archivo2>.o <archivo3>.o
```

Se genera el **ejecutable**:

```
a.out
```

Si queremos ponerle nombre:

```
$ g++ -o <nombre> <archivo1>.o <archivo2>.o <archivo3>.o
```

En resumen

Usando solo una orden:

```
$ g++ -o <nombre> <archivo1>.cpp <archivo2>.cpp <archivo3>.cpp
```

Bibliotecas

Si disponemos de un conjunto de módulos objeto podemos **generar una biblioteca** utilizando la orden **ar**.

```
$ ar -rvs <libreria>.a <archivo1>.o <archivo2>.o <archivo3>.o
```

Para generar el programa ejecutable hay que especificar explícitamente la(s) biblioteca(s) que se utilizan, usando:

```
$ g++ -L./ -o <nombre> <archivo1>.o <archivo2>.o <archivo3>.o
-l<biblioteca>
```

- *-L* permite especificar directorios en donde *g++* puede buscar las bibliotecas necesarias.
- *-l* busca la biblioteca cuya raíz es . Por ejemplo, una biblioteca llamada *mates*, la orden sería *-lmates*.

- *-I* permite especificar directorios en donde *g++* puede buscar los archivos de cabecera. Esta opción no tiene sentido si todos los archivos de cabecera se encuentran en el directorio donde estamos ejecutando todas estas órdenes.

```
$ g++ -I./<directorio> -L./ -o <nombre> <archivo1>.cpp <archivo2>.cpp
<archivo3>.cpp -l<biblioteca>
```

Donde <*nombre*> es el archivo ejecutable.

Archivos de tipo MAKEFILE

make permite gestionar las dependencias, comprobando qué archivos se han modificado desde la última vez que se ejecutó para construir el archivo ejecutable y, en su caso, vuelve a construirlo haciendo de nuevo sólo lo que sea necesario, es decir, compilando exclusivamente aquellos archivos que hubieran sido modificados.

La idea es especificar en un documento de texto las dependencias entre los archivos y las acciones que deben llevarse a cabo si se produce alguna modificación.

Para ejecutar un archivo makefile escribimos:

```
$ make -f <makefile>
```

Donde <*makefile*> es el nombre del archivo makefile.

ESTRUCTURA

```
objetivo: dependencias
    orden1
    orden2
    ...
    ordenN
```

El *objetivo* ha de ser un nombre que resulte característico para la acción que representará y ha de escribirse comenzando desde la primera columna de la línea. El objetivo suele coincidor con el nombre de un archivo.

Por ejemplo:

```
<nombre>:
    g++ -I./<directorio> -o <nombre> <archivo1>.cpp <archivo2>.cpp
    <archivo3>.cpp
```

- Aquí, se considera como **objetivo principal** y se ha de poner antes de otros objetivos.
- Las *dependencias* especifican los archivos u otros objetivos posteriores de los que depende el objetivo al cual están asociadas. Si hay una lista de dependencias, éstas deberán estar separadas por un espacio en blanco.
- Las *órdenes* son un conjunto de una o más líneas de orden del shell y siempre deben tener un tabulador al principio de la línea de orden.

Ejemplo.

```
programa1: main.o factorial.o hello.o
    g++ -o programa1 main.o factorial.o hello.o
main.o: main.cpp
    g++ -I./includes -c main.cpp
factorial.o: factorial.cpp
    g++ -I./includes -c factorial.cpp
hello.o: hello.cpp
    g++ -I./includes -c hello.cpp
```

Además, se puede invocar una regla sin dependencias a la hora de ejecutar la utilidad make. A ese tipo de reglas se las denomina **reglas virtuales**. Se sitúan al final de las demás reglas. Por ejemplo:

```
...
clean:
    rm *.o
```

Para ejecutar esta última regla:

```
$ make -f <makefile> clean
```

Variables

Definimos las siguientes variables:

```
CC = g++
CPPFLAGS = -Wall
SRCS = main.cpp factorial.cpp hello.cpp
OBJS = main.o factorial.o hello.o
HDRS = functions.h
```

- **$@**: nombre del objetivo de la regla.

```
programa1: $(OBJS)
    $(CC) -o $@ $(OBJS)

$@se sustituirá por programa1
```

- **$<**: primera dependencia de la regla.

```
hello.o: hello.cpp
    $(CC) -c $(CPPFLAGS) $<

$< se sustituirá por hello.cpp
```

- **$?**: dependencias de la regla que hayan sido actualizadas dentro del objetivo de la regla y separadas por un espacio en blanco. Si se ejecutase *make* sobre este archivo makefile junto con el argumento *print*, la orden imprimiría aquellos archivos que hubieran sido modificados hasta ese momento. Esta opción es muy útil cuando se desea disponer de una copia impresa de los archivos fuente.

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

- **$^**: todas las dependencias separadas por un espacio en blanco.

```
programa1: $(OBJS)
    $(CC) -o $@ $^
```

```
$^ en la regla se sustituirá por los nombres de todas sus
dependencias, es decir, por la secuencia main.o factorial.o hello.o.
```

- **Wall**: muestra todos los warnings que puedieran aparecer durante el proceso de compilación.
- **-p**: muestra todas las variables predefinidas que se pueden usar dentro de la especificación de un archivo makefile.

```
make -p
```

- Se puede sustituir una secuencia de cadenas en una variable definida previamente.

```
$(Nombre:TextoActual=TextoNuevo).
```

Por ejemplo, dada la variable

```
SRCS = main.cpp factorial.cpp hello.cpp
```

se puede crear otra, por ejemplo *OBJS,* en función de la existente simplemente sustituyendo su contenido de la siguiente forma:

```
OBJS=$(SRCS:.cpp=.o)
```

De esta manera se obtiene

```
OBJS = main.o factorial.o hello.o.
```

SESIÓN 9: Depuración de programas

Para depurar programas se utiliza **gdb**, que permite ver qué pasa dentro de un programa cuando éste se ejecuta, o qué pasó cuando el programa dio un fallo y abortó.

Esquema normal de funcionamiento:

1. Compilar el programa con **g++ con la opción -g** (para poder usar gdb).

```
$ g++ -g <archivo1>.cpp <archivo2>.cpp -o <nombre>
```

```
ó
```

```
$ g++ -g -o <nombre> <archivo1>.cpp <archivo2>.cpp
```

```
Donde <nombre> es el archivo ejecutable.
```

2. Ejecutar el depurador **gdb**.
3. Dentro del intérprete del depurador, ejecutar el programa con la orden **run**.
4. Mostrar el **resultado** que aparece tras la ejecución.

Órdenes

- **apropos**: busca en todo el manual por si hubiera ayuda a un término.

```
apropos run
```

- **quit / q**: abandonar el depurador.
- **list / l**: listar código.

```
list 1, 10
Para listar todas las líneas desde la 1a la 10
```

- **display** *variable*: muestra el valor de la variable durante la ejecución cada vez que el programa se detiene en un punto de ruptura. A cada orden display se le asigna un valor numérico que permite referenciarla.
- **print** *variable*: muestra el valor de una variable únicamente en el punto de ruptura en el que se da esta orden. Se puede aplicar tanto a variables de área global o de alcance local.
- **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. Ese valor toma 1 para la primera orden display, 2 para la siguiente y así sucesivamente.
- **examine** *dirección*: examina el contenido de una dirección de memoria. La dirección siempre se expresa en hexadecimal.
- **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 ejecutará usando el contador de programa.
- **disassemble**: muestra el código ensamblador de la parte que estamos depurando.
- **whatis** *variable*: devuelve el tipo de una variable.
- **info locals**: lista todas las variables locales.
- **break**: para añadir puntos de ruptura simple que permiten examinar qué hace el programa en un determinado lugar. Puede tomar como parámetro: el nombre de una función, la dirección lógica donde parar, o un número de línea.
- **continue**: para continuar el programa hasta el final o hasta el próximo punto de ruptura.
- **next / n**: avanzar a la siguiente instrucción del programa una vez detenidos a causa de un punto de ruptura. Ejecuta el subprograma como si se tratase de una instrucción simple (no entra en funciones, etc.).
- **step / s**: avanzar a la siguiente instrucción del programa una vez detenidos a causa de un punto de ruptura. Entra en el marco donde se encuentra el subprograma ejecutando sus instrucciones paso a paso (entra en funciones, etc.).
- **info breakpoints**: para ver los puntos de ruptura activos.
- **delete**: para eliminar puntos de ruptura.

- **info frame**: muestra información acerca del marco actual (frame = marco).
- **backtrace full**: muestra la información referente a las variables locales y el resto de información asociada al marco.
- **down**: bajar en la pila de marcos.
- **up**: subi en la pila de marcos.

Ejecución de guiones

Ejemplo.

```
break multiplica
run
display x
display y
display final
continue
continue
delete display 1
delete display 2
continue


$ gdb -x guion.gdb ejemplo1
```

Puntos de ruptura condicionales

Solo se habilita el punto de ruptura si se cumple la condición. Se utiliza la sintaxis del lenguaje depurado.

Ejemplo.

```
(gdb) break 13 if tmp > 10
Punto de interrupción 1 at 0x804866a: file mainsesion10.cpp, line 13.

(gdb) run
Starting program: /home/usuario/ejemplo10.1

(gdb) Breakpoint 1, cuenta (y=0) at mainsesion10.cpp:13
13  tmp = y + 2;

(gdb) print tmp
$3 = 11
```

Cambio de valores en variables

Se puede cambiar el valor de una variable mientras se está depurando un programa. Para ello se utiliza la orden **set**.

```
set variable <variable>=<valor>
```

Ejemplo.

```
(gdb) break 10
Punto de interrupción 1 at 0x804866a: file mainsesion10.cpp, line 10.
(gdb) run
Starting program: /home/usuario/ejemplo10.1
Breakpoint 1, cuenta (y=0) at mainsesion10.cpp:13
13  tmp = y + 2;
(gdb) print tmp
$1 = 6  //** Podría dar un valor diferente **//
(gdb) set variable tmp=10
(gdb) print tmp
$2 = 10
```

Depurar programas que se están ejecutando

Nada más ejecutar *gdb* podemos realizar la depuración de un proceso que ya se encuentre en ejecución. Se usa la orden **attach**.

```
attach <PID>
```

donde *PID* es el identificador del proceso que se encuentra en ejecución y se desea depurar.

Ejemplo.

```
$ g++ -g ejsesion10.cpp –o ej1

$ ./ej1 &   // Ejecutamos en segundo plano ej1
[1] 28942   // El número 28942 es el PID asignado a ej1.

$ gdb
(gdb) attach 28942  // Indicamos el PID obtenido anteriormente
Attaching to process 28942
Reading symbols from /home/usuario/ej1...hecho.
```

Funcionalidad adicional del gdb

La utilidad *gdb* permite integrarse con los editores del sistema, de tal forma que **podemos lanzar el editor en cualquier momento** desde éste. Para ello hay que cambiar la variable EDITOR del sistema:

```
$ EDITOR=/usr/bin/gedit
$ export EDITOR
```

A partir de aquí, una vez estamos depurando podemos escribir:

```
edit <número_línea>
edit <nombre_función>
```

El primero nos permite la edición del número de línea indicada y el segundo la función.

Desde gdb también podemos hacer **llamadas a la shell**. Para que podamos ejecutar cualquier orden de la shell desde la propia utilidad gdb como si estuviéramos en un terminal, basta con realizar lo siguiente:

```
shell <orden>
```

De esta manera ejecutamos la orden de la shell indicada en orden.

Ejemplo.
(gdb) shell cat ej1.cpp
#include

```
using namespace std;

  // Este programa trata de sumar una lista de numeros.
... ...
```

Otra posibilidad que nos permite *gdb* es la de, sin abandonar la ejecución del depurador, **poder introducir todas las órdens de la Shell que deseemos y posteriormente regresar a la depuración**. Para ello ejecutamos *shell*, damos las órdenes deseadas, y para regresar a la depuración ejecutamos *exit*.

Ejemplo.
(gdb) Shell

$ cat ej1.cpp

#include

```
using namespace std;
... ...
$ exit
(gdb)
```