Examen Parcial FS - Módulo 2

Comandos útiles

- `rm -rf` fuerza a borrar el directorio, incluso si no está vacío.

`g++` : compilación, enlazado, librerías...

- Generar el módulo objeto:
  `g++ -c main.cpp`
- Realizar preprocesado, compilación y enlazado, directamente:
  `g++ main.cpp`
- Mayoría de veces es necesario proporcionar más módulos:
  `g++ -c factorial.cpp`
  `g++ -c hello.cpp`
  `g++ -c main.cpp`
  `g++ main.o factorial.o hello.o`
- Lo anterior genera ejecutable `a.out`, para cambiar el nombre:
  `g++ -o programa1 main.o factorial.o hello.o`
- Creación de una **biblioteca**:
  `g++ -c sin.cpp`
  `g++ -c cos.cpp`
  `g++ -c tan.cpp`
  `ar -rvs libmates.a sin.o cos.o tan.o`
- Generar programa ejecutable a partir de módulos objeto y biblioteca:
  `g++ -L./ -o programa2 main2.o factorial.o hello.o -lmates`
  Por omisión lo buscará en `/lib` y `/usr/lib`.
- Usar **archivos de cabecera**:
  `g++ -I./includes -L./ -o programa2 main2.cpp factorial.cpp hello.cpp -lmates`

Makefiles

Los `makefile` son los archivos donde se hace la compilación automática del programa.

Para acceder a un makefile sólo hace falta:

```
makefile -f [nombre_makefile]
```

Si el makefile se llama `makefile`, no es necesario escribir `-f [nombre_makefile]`.

En los makefile se muestra el orden de compilación de un programa, con elemento y dependencias:

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

También podemos añadir reglas.

Variables:

- `$@` = objetivo
- `$<` = primera dependencia
- `$?` = dependencias que hayan sido modificadas
- `$^` = dependencias

Anotaciones importantes

- Usar **siempre** `g++`, en lugar de `gcc`.

Ejemplos de `makefile`

Muy completo (con subdirectorios y variables)

```
#Autor:
#    Sergio Quijano Rey
#    sergiquijano@gmail.com
#Descripción:
#    Prototipo de make que funciona en la mayoría de los casos con mínimas
# modificaciones

#Variables
CC = g++            #Compilador a usar
CFLAGS = -Wall        #Opciones que queremos pasarle al compilador
INCLUDES = -I./Includes
LIB = -L./librerias

#Compilacion del programa
all: main

main: main.o ./librerias/libfunciones.a ./librerias/libobjetos.a
```

```
    $(CC) $(CFLAGS) $(INCLUDES) $(LIB) main.o -lfunciones -lobjetos -o main

main.o: main.cpp
    $(CC) $(CFLAGS) $(INCLUDES) -c main.cpp -o main.o

./source/mates.o: ./source/mates.cpp ./Includes/func.h
    $(CC) $(CFLAGS) $(INCLUDES) -c ./source/mates.cpp -o ./source/mates.o

./source/salida.o: ./source/salida.cpp ./Includes/func.h
    $(CC) $(CFLAGS) $(INCLUDES) -c ./source/salida.cpp -o ./source/salida.o

./source/objetos/complejo.o: ./source/objetos/complejo.cpp ./Includes/complejo.h
    $(CC) $(CFLAGS) $(INCLUDES) -c ./source/objetos/complejo.cpp -o ./source/objetos

./librerias/libfunciones.a: ./source/mates.o ./source/salida.o
    ar -rvs ./librerias/libfunciones.a ./source/mates.o ./source/salida.o

./librerias/libobjetos.a: ./source/objetos/complejo.o
    ar -rvs ./librerias/libobjetos.a ./source/objetos/complejo.o

#Utilidades
clean:
    rm main.o ./source/salida.o ./source/mates.o ./source/objetos/complejo.o ./libr

cleanAll:
    rm main.o ./source/salida.o ./source/mates.o ./source/objetos/complejo.o ./libr
    rm main

exe:
    ./main
```

Examen 1

```
INSTALL_DIR=./programa

all: programa.out

programa.out: pr.o ./libmates.a
    g++ -I./ -L./ pr.o -lmates -o programa.out

pr.o: pr.cpp
    g++ -I./ -c pr.cpp

libmates.a: geom.o complex.o vector.o
    ar -rvs libmates.a geom.o complex.o vector.o

geom.o: geom.cpp
    g++ -I./ -c geom.cpp

complex.o: complex.cpp
    g++ -I./ -c complex.cpp
```

```
vector.o: vector.cpp
    g++ -I./ -c vector.cpp

install:
    cp programa.out $(INSTALL_DIR)

uninstall:
    rm $(INSTALL_DIR)/programa.out

clear:
    rm pr.o libmates.a geom.o complex.o vector.o
```

Examen 2

```
```

`gdb` : depuración

Para que el programa sea depurable, es necesario insertar `-g` al usar `g++`:

```
g++ -g main.cpp hello.cpp factorial.cpp -o ejemplo1
gdb ejemplo1
```

Orden de `gdb`	Descripción
`display [variable]`	Muestra el valor de la variable 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 asignado a la orden `display` correspondiente. Este valor toma 1 para la primera orden, 2 para la segunda y así sucesivamente.
`examine [dirección]`	Examina el contenido de una dirección de memoria, expresada en hexadecimal (ej. `0x000f1`).

`show values`	Muestra la historia de valores por 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 dato de una variable.
`info locals`	Lista todas las variables locales.

Puntos de ruptura

- Poner un punto de ruptura: `break` seguido de línea, nombre de función o dirección lógica.
- Avanzar a la siguiente instrucción del programa: `next` o `step` (ver más adelante).
- Ver puntos de ruptura activos: `info breakpoints`.
- Eliminar punto de ruptura: `delete`.

Guiones

El guion de gdb contiene la información que pondríamos en la pantalla de gdb, de forma que se ejecute al llamarlo. Por ejemplo, si el guion se llama `guion.gdb`:

```
gdb -x guion.gdb ejemplo1
```

Depuración avanzada de programas: marcos (*frames*)

La **pila de llamadas** es el lugar del programa donde se almacena información sobre las direcciones donde se van a ejecutar determinadas funciones del mismo. A su vez se divide en secciones contiguas, llamadas **pilas de marcos**. Cada marco es el conjunto de datos asociado a la llamada de una función. Contiene los argumentos de la función, las variables locales y la dirección en la cual la función se ejecuta.

El marco de la función que se está ejecutando se llama ***innermost frame***.

Un marco se identifica por su dirección, que normalmente se almacena en un registro llamado **registro puntero al marco**.

Orden de `gdb`	Descripción
`info frame`	Muestra información acerca del marco actual.
`backtrace full`	Muestra información referente a las variables locales y al resto de información asociada al marco.
`step`	Entra en el marco donde se encuentra un subprograma cuya ejecución ha sido detenida en una instrucción de llamada a un subprograma, ejecutando las instrucciones paso a paso.
`next`	Ejecuta el subprograma como si se tratase de una instrucción simple todo él.
`down`	Bajar en la pila de marcos, de forma que podemos ir a la función más interna.
`up`	Subir en la pila de marcos, de forma que podemos subir a donde se hizo la última llamada a dicha función.

Puntos de ruptura condicionales

El siguiente ejemplo aplica el punto de ruptura si `tmp > 10`:

```
(gdb) break 13 if tmp > 10
```

Camio de valores en variables

Podemos cambiar el valor de una variable mientras se está depurando el programa. La sintaxis es `set variable [variable]=[valor]`:

```
(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
(gdb)
```

Depurar progamas en ejecución

Gracas a `gdb` podemos depurar programas que ya se estén ejecutando en el sistema operativo, La orden sería `attach PID` donde `PID` es el identificador del proceso en ejecución que se desea depurar.

Podemos obtener el `PID` cuando ejecutamos el programa:

```
./ej1 & ## nos devolverá el PID
```

O si ya está en ejecución:

```
px ax | grep [nombre]
```

> El resto de funcionalidades secundarias están en los apuntes.

Ejemplos

Examen 1

Este es el código que debería ponerse en `makefile`. Es **muy importante** que aparezca `-g` para hacerlo *debuggable*.

```
INSTALL_DIR=./programa

all: programa.out

programa.out: pr.o ./libmates.a
    g++ -g -I./ -L./ pr.o -lmates -o programa.out

pr.o: pr.cpp
    g++ -g -I./ -c pr.cpp

libmates.a: geom.o complex.o vector.o
    ar -rvs libmates.a geom.o complex.o vector.o

geom.o: geom.cpp
```

```
    g++ -g -I./ -c geom.cpp

complex.o: complex.cpp
    g++ -g -I./ -c complex.cpp

vector.o: vector.cpp
    g++ -g -I./ -c vector.cpp

debug:
    gdb -x ./guion.gdb programa.out


install:
    cp programa.out $(INSTALL_DIR)

uninstall:
    rm $(INSTALL_DIR)/programa.out

clear:
    rm pr.o libmates.a geom.o complex.o vector.o
```

Y en un archivo `guion.gdb` :

```
break vsum
break garea if cr.r <= 10
run
print i
continue
print i
continue
print i
continue
print i
continue
print i
continue
```