Fundamentos del Software

Prácticas

Módulo II

Compilación y depuración de programas

22-sep-2014

Grado en Ingeniería Informática y
Doble Grado en Ingeniería Informática y Matemáticas

Departamento de Lenguajes y Sistemas Informáticos
E.T.S. Ingenierías Informática y de Telecomunicación
Universidad de Granada

Contenido

Práctica 8: Compilación de programas

1 Objetivos principales

- Conocer cómo la utilidad **`gcc/g++`** realiza las distintas etapas del proceso de generación de un archivo ejecutable a partir de distintos archivos de código fuente.
- Conocer las dependencias que se producen entre los distintos archivos implicados en el proceso de generación de un archivo ejecutable.
- Saber construir un archivo `makefile` sencillo que permita mantener las dependencias entre los distintos módulos de un pequeño proyecto software.

Además, se verán las siguientes órdenes:

Utilidades		
`gcc/g++`	**`ar`**	**`make`**

Tabla 8.1. Órdenes de la sesión.

2 Introducción a la compilación de programas con `gcc/g++`

GCC es un compilador integrado del proyecto GNU para los lenguajes C, C++, Objective C y Fortran. A partir de un archivo que contiene un programa escrito en cualquiera de estos lenguajes puede generar un programa ejecutable binario en el lenguaje de la máquina donde queremos que se ejecute el programa. En la figura 1 podemos ver el proceso normal de compilación.

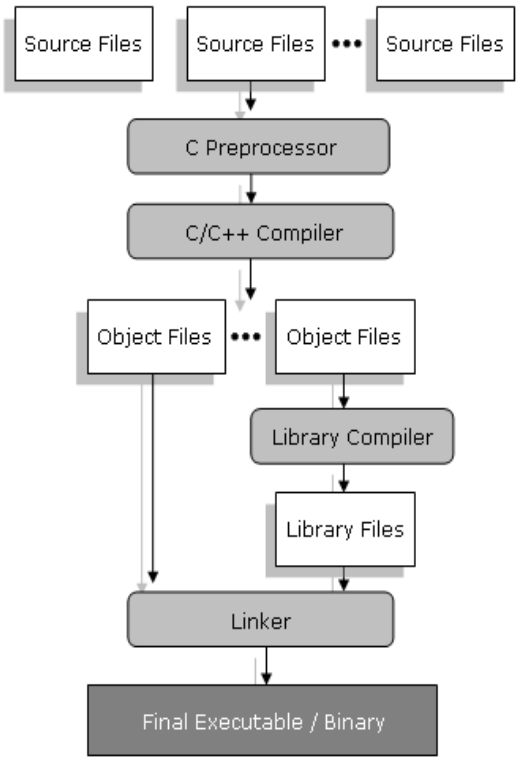

Figura 8.1. Proceso normal de compilación en C/C++.

El *preprocesador* (del inglés, *preprocessor*) acepta como entrada un archivo que contiene código fuente (**`archivo.c`** o **`archivo.cpp`**) y se encarga de eliminar los comentarios e interpretar las *directivas de preprocesamiento*. Estas directivas siempre comienzan por el símbolo **#** (sí, el mismo que precede a los comentarios en Bash). La más interesante por ahora de cara a nuestra práctica es **`#include <archivoinc>`** o **`#include "archivoinc.h"`** que sustituye la línea por el contenido del archivo `archivoinc` o `archivoinc.h`.

El *compilador* (del inglés, *compiler*) analiza el código fuente preprocesado y lo traduce a un código objeto que se almacena en un archivo **`archivoinc.o`**, conocido como **módulo objeto**. En el proceso de compilación se realiza la traducción del código fuente a código objeto pero no se resuelven las posibles referencias a elementos externos al archivo. Las referencias externas se refieren a variables y funciones que, aunque se utilizan en el archivo, y por tanto deben estar declaradas, no se encuentran definidas en éste, sino en otro archivo distinto.

El *enlazador* (del inglés, *linker*) se encarga de resolver las referencias externas y generar un archivo ejecutable. Estas referencias se pueden encontrar en otros archivos compilados, ya sea en forma de archivos ".o" o en forma de bibliotecas. Para generar un archivo ejecutable es necesario que

uno de los archivos ".o" que se están enlazando contenga la función **main()**. Veamos ejemplos de todo esto utilizando los archivos del directorio dirprograma1 que están incluidos en la sesión.

```
$ g++ -c main.cpp #genera, si no hay errores de compilación, el módulo objeto main.o
```

La orden previa se completa con éxito generando un módulo objeto, main.o. La opción -c indica a g++ que solo realice la etapa de preprocesado y compilación. De esta forma, todavía no se ha llevado a cabo la resolución de referencias externas.

Ejercicio 8.1. Pruebe a comentar en el archivo fuente main.cpp la directiva de procesamiento "#include "functions.h". La línea quedaría así: //#include "functions.h". Pruebe a generar ahora el módulo objeto con la orden de compilación mostrada anteriormente. ¿Qué ha ocurrido?

Probemos ahora a generar el programa ejecutable directamente sin parar en la etapa de compilación. Es decir, realizaremos preprocesado, compilación y enlazado.

```
$ g++ main.cpp
. . .
ld returned 1 exit status
```

La orden previa no se completa con éxito. Podemos observar que el enlazador de GNU, **ld**, que es utilizado por la orden **g++**, no ha podido resolver las referencias externas, y además, no se ha generado como resultado intermedio de compilación el archivo main.o.

Para conseguir nuestro objetivo es necesario proporcionar a **ld** otros módulos objeto que contengan la definición de las referencias externas. En nuestro caso son: **hello.o** y **factorial.o**. Vamos a proceder a la compilación de estos módulos y a intentar de nuevo la generación del programa ejecutable.

```
$ g++ -c factorial.cpp            # genera el archivo factorial.o
$ g++ -c hello.cpp                # genera el archivo hello.o
$ g++ -c main.cpp                 # genera el archivo main.o
$ g++ main.o factorial.o hello.o  # genera el archivo ejecutable a.out
```

Como resultado del proceso disponemos de un archivo ejecutable **a.out** que funciona en nuestra plataforma. Si queremos que el archivo ejecutable tenga un nombre distinto al que utiliza g++ por omisión podemos usar la opción -o de la siguiente forma:

```
$ g++ -o programa1 main.o factorial.o hello.o       # archivo ejecutable programa1
```

Como apunte final sobre el proceso de generación de un programa ejecutable usando gcc/g++, podemos realizar todo el procedimiento con una sola orden, utilizando para ello todos los módulos de código fuente que componen el programa. La orden g++ realiza todo el proceso llamando al preprocesador y compilador para cada módulo, y al enlazador para que utilice todos los módulos objeto para generar el programa programa1.

```
$ rm *.o programa1 a.out
$ g++ -o programa1 main.cpp factorial.cpp hello.cpp
```

3 Introducción a las bibliotecas

Una biblioteca es una colección de módulos objeto. Estos módulos contienen constantes, variables y funciones que pueden ser utilizados por otros módulos si se enlazan de forma adecuada. Si disponemos de un conjunto de módulos objeto podemos generar una biblioteca utilizando la orden **ar**. Puede comprobar lo que significan las diferentes opciones utilizadas mediante el manual en línea.

En este caso, los siguientes archivos se encuentran en el subdirectorio `dirprograma2`.

```
$ g++ -c sin.cpp
$ g++ -c cos.cpp
$ g++ -c tan.cpp
$ ar -rvs libmates.a sin.o cos.o tan.o
```

Para comprobar el uso que podemos hacer de nuestra biblioteca vamos a generar un programa ejecutable, `programa2`, a partir de los siguientes módulos de código objeto: `main2.o`, `factorial.o`, `hello.o` que fueron compilados cada uno mediante `g++ -c`.

```
$ g++ -o programa2 main2.o factorial.o hello.o
main2.o: In function 'main':
main2.cpp:(.text+0x58): undefined reference to 'print_sin(float)'
main2.cpp:(.text+0x65): undefined reference to 'print_cos(float)'
main2.cpp:(.text+0x72): undefined reference to 'print_tan(float)'
collect2: ld returned 1 exit status
```

Ejercicio 8.2. Explique por qué el enlazador no ha podido generar el programa archivo ejecutable `programa2` del ejemplo anterior y, sin embargo, ¿por qué sí hemos podido generar el módulo `main2.o`?

Para generar el programa ejecutable hay que especificar explícitamente la(s) biblioteca(s) que se utilizan (vamos, las definiciones de las funciones).

```
$ g++ -L./ -o programa2 main2.o factorial.o hello.o -lmates
```

La opción `-L` permite especificar directorios en donde `g++` puede buscar las bibliotecas necesarias. Por omisión `g++` las busca en los directorios `/lib` y `/usr/lib`. Además, en el caso de `-lmates`, esa opción `-l` busca la biblioteca cuya raíz es `mates`, con prefijo `lib` y sufijo `.a`, es decir, busca la biblioteca `libmates.a`.

Pruebe lo siguiente.

```
$ mkdir includes
$ mv *.h includes
$ rm *.o programa2
$ g++ -L./ -o programa2 main2.cpp factorial.cpp hello.cpp -lmates
main2.cpp:2:23: error: functions.h: No such file or directory
main2.cpp:3:19: error: mates.h: No such file or directory
main2.cpp: In function ‘int main()’:
main2.cpp:8: error: ‘print_hello’ was not declared in this scope
main2.cpp:10: error: ‘factorial’ was not declared in this scope
main2.cpp:11: error: ‘print_sin’ was not declared in this scope
main2.cpp:12: error: ‘print_cos’ was not declared in this scope
main2.cpp:13: error: ‘print_tan’ was not declared in this scope
factorial.cpp:1:23: error: functions.h: No such file or directory
hello.cpp:2:23: error: functions.h: No such file or directory
```

Ejercicio 8.3. Explique por qué la orden `g++` previa ha fallado. Explique los tipos de errores que ha encontrado.

De forma análoga a la opción `-L`, la opción `-I` permite especificar directorios en donde `g++` puede buscar los archivos de cabecera (por omisión, se buscan en `/usr/include`). Esta opción no tiene sentido si todos los archivos de cabecera se encuentran en el directorio donde estamos ejecutando todas estas órdenes. En ese tipo de circunstancias, NO es necesario definir tal opción e indicar directorios concretos.

```
$ g++ -I./includes -L./ -o programa2 main2.cpp factorial.cpp hello.cpp -lmates
```

Como puede comprobar, cuando se trabaja con varios módulos de código fuente se van generando una serie de ***dependencias*** entre ellos. En nuestro caso, el módulo `main.cpp` depende de los módulos `factorial.cpp` y

`hello.cpp`. Además, el módulo `main2.cpp` depende de los anteriores y de la biblioteca `libmates.a`. Si en algún momento es necesario modificar algún módulo, se hace necesario conocer estas dependencias para generar un nuevo programa ejecutable. Por ejemplo si suponemos que el módulo `hello.cpp` cambia la línea: **`cout << "Hello World!";`** por **`cout << "Hello happy World!";`** sería necesario generar de nuevo el módulo objeto correspondiente y generar de nuevo el `programa1` y el `programa2`.

4 Uso de archivos de tipo makefile

Afortunadamente existe una utilidad **`make`** que permite gestionar las dependencias (y otras muchas cosas), 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. Por ejemplo, si sólo hemos modificado uno de los archivos fuente o hemos perdido el archivo objeto correspondiente, la utilidad `make` solamente volverá a generar ese archivo objeto y realizará la fase de enlazado para construir el archivo ejecutable.

Por ejemplo, como se ha visto en la parte introductoria de esta sesión, si deseamos compilar y enlazar de forma manual una serie de archivos fuente como son `main.cpp`, `factorial.cpp` y `hello.cpp` para obtener el archivo ejecutable ejemplo, habría que ejecutar lo siguiente:

```
$ g++ -o programa1 main.cpp factorial.cpp hello.cpp
```

Esta ejecución compila cada archivo fuente y genera sus correspondientes archivos objeto `main.o`, `factorial.o` y `hello.o`, respectivamente, y a continuación los enlaza para obtener el archivo ejecutable denominado `programa1`.

Es posible automatizar el proceso de compilación y enlazado construyendo un archivo de tipo makefile en el que se especifiquen las acciones a realizar. En las siguientes secciones se describirá cómo hacerlo.

4.1 Ejecución de la utilidad `make`

La utilidad `make` admite entre sus opciones la especificación del nombre de un archivo de tipo makefile. Esa opción es `-f` y, a continuación, el nombre del archivo. Si el nombre elegido para el archivo es `makefileGNU`, `makefile` o `Makefile`, en ese caso, no es necesario especificar la opción anterior junto al nombre del achivo, es decir, bastará con ejecutar la utilidad `make` sin argumentos.

Por ejemplo, suponiendo que el único archivo makefile se denomina `makefileA`, se observa el siguiente resultado tras la ejecución de la orden `make`.

```
$ make
make: *** No se especificó ningún objetivo y no se encontró ningún makefile.  Alto.
```

Para que admita el ese archivo makefile, se ha de especificar su nombre mediante la opción `-f`.

```
$ make -f makefileA
```

En cambio, si el archivo se denominase `makefile`, entonces la ejecución de la orden `make` no necesitaría tal opción.

4.2 Estructura de un archivo makefile

Un archivo makefile está compuesto por una o varias reglas cada una de las cuales estará asociada a la consecución de un objetivo concreto. Las *reglas* están formadas por un objetivo, una lista de dependencias (posiblemente vacía) y las acciones u órdenes que son necesarias para alcanzar ese objetivo.

La sintaxis básica para una regla es como sigue:

objetivo$_1$: dependencias

TABULADOR orden$_1$

TABULADOR orden$_2$
…

TABULADOR orden$_N$

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. En la mayoría de los casos, el objetivo coincide con el nombre de un archivo, tal y como se podrá comprobar en los sucesivos ejemplos.

A partir del ejemplo que hemos reseñado anteriormente, resulta claro que el objetivo es la construcción del programa ejecutable. Una regla que permite alcanzar ese objetivo según la estructura de un archivo makefile sería la siguiente:

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

Si guardamos la descripción anterior en un archivo denominado `makefileA`, para ejecutarlo y obtener el mismo resultado que si se hubiera ejecutado la orden de forma manual tendrá que realizarse así:

```
$ make –f makefileA
g++ -I./includes -o programa1 main.cpp hello.cpp factorial.cpp
```

En este ejemplo, el objetivo denominado `programa1`, 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. Cuando se ejecuta la utilidad `make`, ésta comprueba si ha habido algún cambio en alguna de sus dependencias; si es así, se busca el objetivo correspondiente de las dependencias modificadas y se ejecuta su lista de órdenes asociadas. Una vez actualizada la lista de dependencias se construye (ejecutando las órdenes especificadas) el objetivo.

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. Estas órdenes permiten normalmente construir el objetivo aunque, como hemos visto en la regla del ejemplo anterior, también pueden servir para realizar otras tareas.

Visto el ejemplo anterior, en algunas ocasiones es muy útil usar diferentes objetivos de tal forma que, si sólo modificamos un único archivo fuente, únicamente se recompile dicho archivo y luego se enlace, sin tener que realizar el proceso de compilación y enlazado con todos los pasos.

Una descripción de lo anterior independizando el proceso de compilación y el de enlazado daría como resultado un archivo makefile compuesto por varias reglas (con objetivos, dependencias y órdenes), denominado `makefileB`, podría quedar así:

```
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
```

Nótese que para el objetivo `programa1` existen tres dependencias que son las de la existencia de los archivos objeto `main.o`, `factorial.o` y `hello.o`. Dichas dependencias son, a su vez, otros objetivos descritos en el mismo archivo makefile.

Suponiendo que no existen los archivos objeto ni el ejecutable, la ejecución de la utilidad `make` con este archivo arrojaría el siguiente resultado:

```
$ make –f makefileB
g++ -I./includes -c main.cpp
g++ -I./includes -c factorial.cpp
g++ -I./includes -c hello.cpp
g++ -o programa1 main.o factorial.o hello.o
```

Si ahora borramos uno de los archivos objeto, por ejemplo `factorial.o`, el resultado de ejecutar la utilidad `make` sería el siguiente:

```
$ make –f makefileB
g++ -I./includes -c factorial.cpp
g++ -o programa1 main.o factorial.o hello.o
```

En esta ejecución podemos observar que sólo será necesaria la obtención del archivo objeto `factorial.o` y el posterior enlazado dado que los otros dos archivos objeto ya existían y no han sido modificados sus archivos fuente.

Basándonos en este mismo ejemplo, el archivo `functions.h` está incluido en los archivos fuente `main.cpp`, `factorial.cpp` y `hello.cpp`. ¿Qué ocurriría si modificásemos el archivo `functions.h`? En ese caso, no se recompilaría absolutamente nada porque en el archivo `makefileB` no existe ninguna dependencia en la que intervenga. Para solucionarlo, sólo hemos de incluir este archivo de cabecera en las dependencias de aquellos objetivos donde su archivo fuente lo incluya. Si el archivo `functions.h` estuviera ubicado en el mismo directorio que el resto de archivos fuente, el nuevo archivo makefile, que denominaremos `makefileC`, quedaría de la siguiente forma:

```
programa1: main.o factorial.o hello.o
	g++ -o programa1 main.o factorial.o hello.o

main.o: main.cpp functions.h
	g++ -c main.cpp

factorial.o: factorial.cpp functions.h
	g++ -c factorial.cpp

hello.o: hello.cpp functions.h
	g++ -c hello.cpp
```

Suponiendo que se modifica el archivo `functions.h`, se deberían recompilar los archivos `main.cpp`, `factorial.cpp` y `hello.cpp`. Pruébelo y compruebe que se realiza tal y como se ha explicado.

Es importante reseñar que este ejemplo, descrito según el makefile anterior, funciona perfectamente pues detectaría posibles cambios realizados al archivo `functions.h` al encontrarse éste en el mismo directorio que el resto de los archivos fuente.

Si el archivo `functions.h` estuviese en otro directorio, habría que indicar la ruta para acceder a él tanto en la dependencia como en la orden donde interviene. En el siguiente ejemplo, suponiendo que se encuentra en un subdirectorio denominado `includes`, el archivo `makefileC`, ahora denominado `makefileD`, pasaría a ser el siguiente:

```
programa1: main.o factorial.o hello.o
	g++ -o programa1 main.o factorial.o hello.o

main.o: main.cpp ./includes/functions.h
	g++ -I./includes -c main.cpp

factorial.o: factorial.cpp ./includes/functions.h
	g++ -I./includes -c factorial.cpp

hello.o: hello.cpp ./includes/functions.h
	g++ -I./includes -c hello.cpp
```

Es posible construir una regla que no tenga órdenes asociadas. En este caso, cuando `make` trate de construir el objetivo de esta regla simplemente comprobará que los archivos de la lista de dependencias están actualizados. Si es necesario construir alguno, pasa a ejecutar la regla que tiene este archivo como objetivo. Este tipo de reglas se conocen como *objetivos simbólicos* o *destinos simbólicos*. Si no se especifica ninguna regla en la ejecución de la utilidad `make`, se procesan las reglas desde la primera que se encuentre en adelante. Para aprender más sobre la ejecución de la orden `make` puede consultar el manual en línea.

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*. Por ejemplo, podemos crear una regla virtual cuyo objetivo denominaremos `clean` y cuya funcionalidad sea la de eliminar los archivos objeto generados. Recuerde que, al no ser un objetivo que intervenga en la consecución del objetivo principal, generalmente se sitúan al final de las demás reglas.

A continuación se puede ver cómo quedaría dentro de un archivo makefile cualquiera:

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

Para ejecutar esta última regla es necesario especificar el nombre del objetivo a la hora de invocar a la utilidad `make`, tal y como se muestra a continuación:

```
$ make -f makefile clean
```

Busque en internet el problema que se puede dar en las reglas virtuales cuando se crea un archivo en el directorio con el mismo nombre de la regla (por ejemplo, `clean`). ¿Cómo se soluciona?

En general, la utilidad `make`, permite la ejecución de una regla cualquiera invocando el nombre del objetivo de la regla como argumento del `make`.

En un archivo de tipo makefile, también se pueden añadir comentarios anteponiendo el símbolo **#** en su primera columna y se pueden extender a lo largo de toda una línea de texto. Si deseamos varias líneas de comentario cada una de ellas deberá comenzar por **#**.

A continuación se muestra cómo debería ser un archivo makefile algo más complejo, al que denominaremos `makefileE`, que nos permitirá automatizar el proceso de compilación y enlazado para la obtención de un programa ejecutable que llamaremos `programa2`. Los archivos fuente para la obtención del mismo serán `main2.cpp`, `factorial.cpp`, `hello.cpp` y los archivos fuente necesarios para construir la biblioteca `libmates.a`, en concreto, `sin.cpp`, `cos.cpp` y `tan.cpp`. Además, existen los archivos de cabecera `mates.h` y `functions.h` que se encuentran alojados en el subdirectorio `includes` (para ver qué archivos fuente los incluyen puede usar la orden `grep`).

```
# Nombre archivo: makefileE
# Uso: make –f makefileE
# Descripción: Mantiene todas las dependencias entre los módulos y la biblioteca
#              que utiliza el programa2.

programa2: main2.o factorial.o hello.o libmates.a
	g++ -L./ -o programa2 main2.o factorial.o hello.o -lmates

main2.o: main2.cpp
	g++ -I./includes -c main2.cpp

factorial.o: factorial.cpp
	g++ -I./includes -c factorial.cpp

hello.o: hello.cpp
	g++ -I./includes -c hello.cpp

libmates.a: sin.o cos.o tan.o
	ar -rvs libmates.a sin.o cos.o tan.o

sin.o: sin.cpp
	g++ -I./includes -c sin.cpp

cos.o: cos.cpp
	g++ -I./includes -c cos.cpp

tan.o: tan.cpp
	g++ -I./includes -c tan.cpp
```

Ejercicio 8.4. Copie el contenido del makefile previo a un archivo llamado `makefileE` ubicado en el mismo directorio en el que están los archivos de código fuente `.cpp`. Pruebe a modificar distintos archivos `.cpp` (puede hacerlo usando la orden `touch` sobre uno o varios de esos archivos) y compruebe la secuencia de instrucciones que se muestra en el terminal al ejecutarse la orden `make`. ¿Se genera siempre la misma secuencia de órdenes cuando los archivos han sido modificados que cuando no? ¿A qué cree puede deberse tal comportamiento?

Ejercicio 8.5. Obtener un nuevo `makefileF` a partir del makefile del ejercicio anterior que incluya además las dependencias sobre los archivos de cabecera. Pruebe a modificar cualquier archivo de cabecera (usando la orden `touch`) y compruebe la secuencia de instrucciones que se muestra en el terminal al ejecutarse la orden `make`.

4.3 Uso de variables

La utilidad `make` permite la definición de variables de igual forma que podíamos hacerlo en los guiones de bash. Fíjese en el siguiente ejemplo:

```
# Nombre archivo: makefileG
# Uso: make –f makefileG
# Descripción: Mantiene todas las dependencias entre los módulos y la biblioteca
#              que utiliza el programa2.
```

```
# Variable que indica el compilador que se va a utilizar
CC=g++

# Variable que indica el directorio en donde se encuentran los archivos de cabecera
INCLUDE_DIR= ./includes

# Variable que indica el directorio en donde se encuentran las bibliotecas
LIB_DIR= ./

programa2: main2.o factorial.o hello.o libmates.a
	$(CC) -L$(LIB_DIR) -o programa2 main2.o factorial.o hello.o -lmates

main2.o: main2.cpp
	$(CC) -I$(INCLUDE_DIR) -c main2.cpp

factorial.o: factorial.cpp
	$(CC) -I$(INCLUDE_DIR) -c factorial.cpp

hello.o: hello.cpp
	$(CC) -I$(INCLUDE_DIR) -c hello.cpp

libmates.a: sin.o cos.o tan.o
	ar -rvs libmates.a sin.o cos.o tan.o

sin.o: sin.cpp
	$(CC) -I$(INCLUDE_DIR) -c sin.cpp

cos.o: cos.cpp
	$(CC) -I$(INCLUDE_DIR) -c cos.cpp

tan.o: tan.cpp
	$(CC) -I$(INCLUDE_DIR) -c tan.cpp
```

En concreto, se han definido las siguientes variables: `CC` , `INCLUDE_DIR` y `LIB_DIR`. Estas variables se pueden usar posteriormente en las declaraciones de las reglas simplemente incluyéndolas entre paréntesis o llaves y anteponiéndoles el signo `$`. Además existen variables especiales que actúan cuando `make` procesa cada regla. A continuación se muestran algunas de ellas:

Variable	Significado
$@	Representa el nombre del objetivo de la regla en la que nos encontramos
$<	Representa 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
$^	Representa todas las dependencias separadas por un espacio en blanco

Tabla 8.2. Variables especiales de la orden `make`.

En los siguientes epígrafes se mostrarán ejemplos de uso de estas variables. Para ello vamos a suponer las siguientes declaraciones en un archivo makefile:

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

En la variable `CPPFLAGS` se indican las opciones del compilador. En este caso, la opción `-Wall` sirve para mostrar todos los warnings que pudieran aparecer durante el proceso de compilación.

4.3.1 Uso de $@

El caso que se muestra a continuación sirve para representar el nombre que se le asociará al programa ejecutable usando para ello el mismo nombre asignado al objetivo:

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

Como puede verse, el valor de `$@` en la regla se sustituirá por `programa1`.

4.3.2 Uso de $<

Esta variable se utiliza para representar el archivo aportado como primera dependencia en una regla. De este modo, por ejemplo, en un proceso de compilación, a la hora de indicar la orden para la obtención del archivo objeto, en lugar de referenciar el nombre de ese primer archivo, se puede representar con esta variable.

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

Como puede verse en esta ocasión, el valor de `$<` en la regla se sustituirá por `hello.cpp`, es decir, la primera dependencia (y única en esta ocasión) de la regla.

4.3.3 Uso de $?

Cuando se desea hacer referencia a varias de las dependencias a la hora de actuar en consecuencia con las órdenes de una regla, es posible usar esta variable para indicarlas y, en concreto, referenciaría aquellas dependencias que se hubieran actualizado:

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

Con esta regla, si se ejecutase la utilidad `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. Si en un momento anterior se hubieran imprimido todos los archivos fuente y, posteriormente, se hubiera modificado sólo uno de ellos, la acción se llevaría a cabo imprimiendo únicamente ese archivo y no los demás.

4.3.4 Uso de $^

Cuando se desea referenciar a todos los archivos indicados en las dependencias de una regla, lo cual supondría el ahorro de tener que escribirlos en varios lugares del archivo makefile, se utiliza la variable `$^`.

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

Como puede verse en esta ocasión, el valor de `$^` en la regla se sustituirá por los nombres de todas sus dependencias, es decir, por la secuencia `main.o factorial.o hello.o`. Además, tal y como se ha mostrado anteriormente, se usa la variable `$@` para indicar el nombre del objetivo como nombre de archivo ejecutable.

4.3.5 Otras opciones y variables

Si la utilidad `make` se ejecuta con la opción `-p` nos muestra las variables predefinidas que se pueden usar dentro de la especificación de un archivo makefile.

Además, en el uso de esta utilidad, se puede sustituir una secuencia de cadenas en una variable definida previamente. La sintaxis es de la siguiente forma: `$(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`.

Como se puede ver, se ha creado una nueva variable a partir del contenido de otra pero cambiando únicamente las extensiones de los archivos.

Ejercicio 8.6. Usando como base el archivo `makefileG`, sustituya la línea de orden de la regla cuyo objetivo es `programa2` por otra en la que se use alguna de las variables especiales y cuya ejecución sea equivalente.

Ejercicio 8.7. Utilizando como base el archivo `makefileG` y los archivos fuente asociados, realice los cambios que considere oportunos para que, en la construcción de la biblioteca estática `libmates.a`, este archivo pase a estar en un subdirectorio denominado `libs` y se pueda enlazar correctamente con el resto de archivos objeto.

Ejercicio 8.8. Busque la variable predefinida de `make` que almacena la utilidad del sistema que permite construir bibliotecas. Recuerde que la orden para construir una biblioteca estática a partir de una serie de archivos objeto es **`ar`** (puede usar la orden `grep` para filtrar el contenido; no vaya a leer línea a línea toda la salida). Usando el archivo `makefileG`, sustituya la orden **`ar`** por su variable correspondiente.

Ejercicio 8.9. Dado el siguiente archivo makefile, explique las dependencias que existen y para qué sirve cada una de las líneas del mismo. Enumere las órdenes que se van a ejecutar a consecuencia de invocar la utilidad `make` sobre este archivo.

```
# Nombre archivo: makefileH
# Uso: make -f makefileH
# Descripción: Mantiene todas las dependencias entre los módulos que utiliza el
# programa1.

CC=g++
CPPFLAGS=-Wall -I./includes
SOURCE_MODULES=main.cpp factorial.cpp hello.cpp
OBJECT_MODULES=$(SOURCE_MODULES:.cpp=.o)
EXECUTABLE=programa1

all: $(OBJECT_MODULES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECT_MODULES)
	$(CC) $^ -o $@

# Regla para obtener los archivos objeto .o que dependerán de los archivos .cpp
# Aquí, $< y $@ tomarán valores respectivamente main.cpp y main.o y así sucesivamente
.o: .cpp
	$(CC) $(CPPFLAGS) $< -o $@
```

Ejercicio 8.10. Con la siguiente especificación de módulos escriba un archivo denominado `makefilePolaca` que automatice el proceso de compilación del programa final de acuerdo a la siguiente descripción:

Compilador: `gcc o g++`
Archivos cabecera: `calc.h` (ubicado en un subdirectorio denominado `cabeceras`)
Archivos fuente: `main.c stack.c getop.c getch.c`
Nombre del programa ejecutable: `calculadoraPolaca`

Además, debe incluir una regla denominada `borrar`, sin dependencias, cuya funcionalidad sea la de eliminar los archivos objeto y el programa ejecutable.

Práctica 9: Depuración de programas

9.1 Objetivos principales

- Conocer cómo la utilidad `gdb` es capaz de seguir la traza de ejecución en ejecutables compilados con código fuente en un lenguaje admitido por `gcc` (C++).
- Conocer las herramientas básicas de depurado y obtención de información de `gdb`.
- Saber construir guiones para `gdb` para automatizar la depuración.
- Conocer el manejo de marcos (frames) en `gdb`.
- Saber utilizar las órdenes de `gdb` para modificar la ejecución de un programa y sus datos.
- Conocer las órdenes avanzadas de depuración de procesos en `gdb`.

Además, se verán las siguientes órdenes:

Utilidades		
g++	**gdb**	**make**

Tabla 9.1. Órdenes de la sesión.

9.2 Introducción a la depuración de programas con `gdb`

La utilidad `gdb` o GNU Debugger es el depurador estándar para el sistema operativo GNU. Es un depurador portable que se puede utilizar en varias plataformas Unix y funciona para varios lenguajes de programación como ensamblador, C, C++ o Fortran.

Esta utilidad ofrece la posibilidad de trazar y modificar la ejecución de un programa. El usuario puede controlar y alterar los valores de las variables internas del programa.

Además, la utilidad no contiene su propia interfaz gráfica de usuario y por defecto se controla mediante una interfaz de línea de órdenes. Existen diversos front-end que han sido diseñados para `gdb`, como DDD, GDBtk/Insight y el "modo GUD" en Emacs.

Por tanto, `gdb`, permite ver qué pasa dentro de un programa cuando éste se ejecuta, o qué pasó cuando el programa dio un fallo y abortó.

El esquema normal de funcionamiento es el siguiente:

1. Compilar el programa con `g++` (o `gcc`) con la opción **`-g`** que añade información necesaria para `gdb` de cara a poder depurar el programa. Esta opción **`-g`** se ha de situar en la fase de compilación, aunque si con una sola orden se efectúa la compilación y el enlazado, entonces se deberá incluir igualmente en esa orden (ver ejemplo más abajo).
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.

Tomando los archivos de esta sesión de prácticas, procederemos a generar un archivo ejecutable a partir de algunos de los archivos fuentes y luego ejecutaremos el depurador añadiendo, como argumento, el archivo ejecutable obtenido fruto del proceso de compilación.

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

Ejercicio 9.1. Compile los archivos `main.cpp factorial.cpp hello.cpp` y genere un ejecutable con el nombre `ejemplo1`. Lance `gdb` con dicho ejemplo y ejecútelo dentro del depurador. Describa la información que ofrece.

9.3 Comprobación de ayuda y listar código

La orden **`help`** de `gdb` permite obtener ayuda genérica del programa. También se puede buscar ayuda de una orden, por ejemplo, **`help run`**.

La orden **`apropos`** busca en todo el manual por si hubiera ayuda asociada a un término, por ejemplo, **`apropos run`**.

Todas las órdenes de `gdb` tienen dos formas de invocarse: la normal (indicando la palabra completa) y la resumida (abreviatura de la orden). La resumida se utiliza para ir más rápido en la depuración, como es el caso de la orden **`quit`**, que sirve para abandonar el depurador, la cual también puede reseñarse escribiendo únicamente la **`q`**.

Para listar código se utiliza la orden **`list`**, (la versión resumida o abreviada sería la letra **`l`**). Por ejemplo: **`list 1,10`** (también se puede indicar de forma abreviada mediante **`l 1,10`**) listaría todas las líneas desde la 1 a la 10.

Ejercicio 9.2. Usando la orden **`list`** muestre el código del programa principal y el de la función `factorial` utilizados en el ejercicio 1 (para ello utilice la orden **`help list`**).

9.4 Comprobación de variables y estados

Una de las ventajas de los depuradores es que podemos visualizar la información de las variables y su estado, así como información del contexto del programa. Para ello pueden ser interesantes las siguientes órdenes:

orden de gdb	Descripción
display variable	Muestra el valor de la variable durante la ejecución cada vez que el programa se detiene en un punto de ruptura (ver apartado 5). 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 (0x000f1 por ejemplo).
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.

Tabla 9.2. Algunas órdenes de `gdb`.

9.5 Puntos de ruptura simples

En `gdb` se pueden añadir puntos de ruptura simple que permiten examinar qué hace el programa en un determinado lugar. Para ello se puede utilizar la orden **`break`**. Esta orden puede tomar como parámetro el nombre de una función, la dirección lógica donde parar, o un número de línea. Para continuar el programa hasta el final o hasta el próximo punto de ruptura (lo que llegue antes), se puede utilizar la orden **`continue`**.

Ejercicio 9.3. Ponga un punto de ruptura asociado a cada línea del programa fuente `mainsesion09.cpp` donde aparezca el comentario `/* break */`. Muestre información de todas las variables que se estén usando cada vez que en la depuración se detenga la ejecución. Muestre la información del contador de programa mediante **`$pc`** y el de la pila con **`$sp`**.

Una vez detenidos a causa a un punto de ruptura, podremos avanzar a la siguiente instrucción del programa con la orden **`next`** o con **`step`**. Ambas órdenes se verán en la siguiente sesión.

Los puntos de ruptura activos pueden verse con **`info breakpoints`**. Podemos eliminar un punto de ruptura con la orden **`delete`** (para mayor detalle, vea la ayuda mediante la orden `help delete`).

Ejercicio 9.4. Indique las órdenes necesarias para ver el valor de las variables `final1` y `final2` del programa generado en el ejercicio anterior en los puntos de ruptura correspondientes tras un par de iteraciones en el bucle `for`. Indique la orden para obtener el código ensamblador de la zona depurada.

Ejercicio 9.5. Considerando la depuración de los ejercicios anteriores, elimine todos los puntos de ruptura salvo el primero.

9.6 Ejecución de guiones

Para no tener que escribir las acciones en la propia interfaz de `gdb`, podemos hacer uso de los guiones de `gdb`. Un guion de este tipo es un archivo de texto con diversas líneas de órdenes. Por ejemplo, el siguiente cuadro indica un guion denominado `guion.gdb` con órdenes para la depuración de un programa denominado `ejemplo1` y, justo debajo, cómo invocar a dicho guion cuando se desee aplicar la depuración establecida en el guion al programa en cuestión:

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

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

Ejercicio 9.6. Realice las acciones del ejercicio 3 y las del ejercicio 5 en un guion y ejecútelas de nuevo mediante la opción `-x` de `gdb`. ¿Sabría decir qué hace este programa con la variable `final2`?

Ejercicio 9.7. Realice la depuración del programa ejecutable obtenido a partir del archivo fuente `ejsesion09.cpp`. Utilizando `gdb`, trate de averiguar qué sucede y por qué no funciona. Intente arreglar el programa.

9.7 Depuración avanzada de programas con `gdb`: marcos *(frames)*

Un programa contiene información acerca de las direcciones donde se van a ejecutar determinadas funciones del mismo. A esta información se le denomina la **pila de llamadas** *(call stack)*. Esta clase de pila también se conoce como una pila de ejecución, pila de control, pila de función, o pila de tiempo de ejecución, y a menudo se describe en forma abreviada como "la pila".

La pila de llamadas se divide en secciones contiguas llamadas **pila de marcos** (*stack frames*) o simplemente marcos (*frames*). Cada marco es el conjunto de datos asociados con una llamada a una función. El marco contiene los argumentos que se le da a la función, sus variables locales, y la dirección en la cual dicha función se ejecuta.

Cuando el programa comienza, la pila contiene solamente un único marco (en C/C++ el marco contiene solamente la función `main`). Cada vez que se llama a una función, se crea un nuevo marco. Cada vez que la función devuelve algo, el marco asignado a dicha función se elimina. El marco de la función que se está ejecutando actualmente se denomina **marco más interno** (*innermost frame*).

En el programa, el marco se identifica por su dirección. Un marco consta de muchos bytes, cada uno de los cuales tiene su propia dirección. Normalmente esta dirección se almacena en un registro llamado **registro puntero al marco** (*frame pointer register*) mientras se ejecuta el siguiente marco.

La utilidad `gdb` emplea marcos en la depuración de un programa. La instrucción **`info frame`** muestra información acerca del marco actual. Mientras que **`backtrace full`** nos muestra la información referente a las variables locales y el resto de información asociada al marco.

Usando los archivos de esta sesión, compile con opciones de depuración el archivo `mainsesion10.cpp` y genere su programa ejecutable llamado `ejemplo10.1`.

```
$ g++ -g –o ejemplo10.1 mainsesion10.cpp
```

Usando el archivo ejecutable obtenido tras la compilación, un ejemplo de información del marco de este ejecutable sería:

```
$ gdb ejemplo10.1
GNU gdb (Ubuntu/Linaro 7.3-0ubuntu2) 7.3-2011.08
Copyright (C) 2011 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Para las instrucciones de informe de errores, vea:
<http://bugs.launchpad.net/gdb-linaro/>...
Leyendo símbolos desde /home/usuario/ejemplo10.1...hecho.

(gdb) break cuenta
Punto de interrupción 1 at 0x80485aa: file mainsesion10.cpp, line 13.

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

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

(gdb) info frame
Stack level 0, frame at 0xbffff2b0:
 eip = 0x80485aa in cuenta (mainsesion10.cpp:13); saved eip 0x8048621
```

```
 called by frame at 0xbffff2e0
 source language c++.
 Arglist at 0xbffff2a8, args: y=0
 Locals at 0xbffff2a8, Previous frame's sp is 0xbffff2b0
 Saved registers:
  ebp at 0xbffff2a8, eip at 0xbffff2ac

(gdb)
```

Las órdenes **next** (de forma abreviada, **n**) y **step** (de forma abreviada, **s**) funcionan de forma distinta. Si se ha detenido la ejecución en una instrucción de llamada a un subprograma, la orden `step` entrará en el marco donde se encuentra ese subprograma ejecutando sus instrucciones paso a paso, mientras que la orden `next` ejecuta el subprograma como si se tratase de una instrucción simple todo él (note que no es igual situar un punto de ruptura en la invocación a una función que en una instrucción interna de la propia función).

Ejercicio 9.8. Compile el programa `mainsesion10.cpp` y genere un ejecutable con el nombre `ejemplo10.1`. Ejecute `gdb` con dicho ejemplo y realice una ejecución depurada mediante la orden `run`. Añada un punto de ruptura *(breakpoint)* en la línea donde se invoca a la función **cuenta** (se puede realizar tal y como se muestra en el ejemplo anterior o mediante el número de línea donde aparezca la llamada a esa función). Realice 10 pasos de ejecución con `step` y otros 10 con `next`. Comente las diferencias.

Con **down** o **up** podemos elegir subir o bajar en la pila de marcos, de tal forma que podemos ir a la función más interna o subir a donde se hizo la última llamada a dicha función.

Ejercicio 9.9. Depure el programa del ejercicio 1. Introduzca un punto de ruptura (*breakpoint*) dentro de la función **cuenta**. Usando la orden `info frame`, muestre la información del marco actual y del marco superior; vuelva al marco inicial y compruebe si ha cambiado algo.

9.8 Puntos de Ruptura Condicionales

La utilidad `gdb` permite añadir comprobaciones en los puntos de ruptura, de tal forma que sólo habilita el punto de ruptura si se cumple la condición. Para ello utiliza la sintaxis del lenguaje depurado, así, si el programa estaba compilado para C++, la sintaxis utilizada será la de C++.

Un ejemplo sería el siguiente:

```
(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
```

Ejercicio 9.10. Ponga un punto de ruptura en la línea 30 del programa (función **multiplica**) de tal forma que el programa se detenga cuando la variable `final` tenga como valor 8. Compruebe si se detiene o no y explique por qué.

9.9 Cambio de valores en variables

Una de las grandes ventajas en `gdb` es que se permite cambiar el valor de una variable mientras se está depurando un programa. Para ello puede ser útil la orden **`set`**, cuya sintaxis es:

`set variable` `variable=valor`

Un ejemplo sería el siguiente usando el mismo ejecutable anterior:

```
(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)
```

Ejercicio 9.11. Pruebe el ejemplo anterior, ejecute después un `continue` y muestre el valor de la variable `tmp`. Todo haría indicar que el valor debiera ser 12 y sin embargo no es así, explique por qué.

9.10 Depurar programas que se están ejecutando

La utilidad `gdb` permite depurar programas que ya se encuentran ejecutándose en el sistema operativo (*daemons*). Para ello, nada más ejecutar `gdb` podemos realizar la depuración de un proceso que ya se encuentre en ejecución. Para ello, dicho proceso debe tratarse de algún programa con un tiempo de ejecución bastante largo o que disponga de alguna instrucción que permita su detención como puede ser la de una sentencia para introducir datos por teclado. Una vez dentro de la utilidad `gdb`, la orden sería la siguiente:

`attach` `PID`

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

Para probar esta posibilidad de depuración emplearemos el archivo fuente de C++ denominado `ejsesion10.cpp` cuya función es la de realizar una suma de una serie de valores dispuestos en un vector junto con otro valor introducido desde teclado. Según el requisito indicado anteriormente, este programa dispone de una sentencia para introducir un dato desde teclado que será sumado al resto de valores del vector.

Un ejemplo sería el siguiente:

```
$ g++ -g ejsesion10.cpp -o ej1
```

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

```
$ gdb
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
```

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

Ejercicio 9.12. Busque cualquier programa escrito en C++ que cumpla los requisitos para poderlo depurar utilizando la orden `attach`. Compílelo usando el flag de depuración, ejecútelo en una Shell en segundo plano y, en otra Shell, ejecute el depurador con el programa que se está ejecutando en estos momentos en la shell anterior. Utilice las órdenes de `gdb` para hacer que el programa que se está ejecutando se detenga en algún lugar y posteriormente se pueda continuar su ejecución. Escriba todos los pasos que haya realizado.

Ejercicio 9.13. Utilizando las órdenes de depuración de `gdb`, corrija el error del programa `ecuacionSegundoGrado.cpp`. Escriba todos los pasos que haya realizado. Pruebe a depurarlo usando `attach`.

9.11 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`.

```
(gdb) shell cat ej1.cpp
#include <iostream>

using namespace std;

/*
 Este programa trata de sumar una lista de numeros.
 La lista de numeros aparece en la variable "vector" y el resultado
 se almacena en la variable "final".
*/
... ...
... ...
```

Otra posibilidad que nos permite la utilidad `gdb` es la de, sin abandonar la ejecución del depurador, poder introducir todas las órdenes 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`.

```
(gdb) shell
$ cat ej1.cpp
#include <iostream>

using namespace std;

/*
 Este programa trata de sumar una lista de numeros.
 La lista de numeros aparece en la variable "vector" y el resultado
 se almacena en la variable "final".
*/
... ...
... ...
$ exit
(gdb)
```