Práctica 7

Manual básico de uso de gdb

7.1 La lucha de los programadores con los errores

Uno de los grandes problemas al que nos enfrentamos al codificar un algoritmo son los errores. Aunque el algoritmo que estemos codificando haya sido formalmente verificado, a la hora de codificarlo en un lenguaje de programación podemos el resultado puede ser erróneo. Bien porque nos hemos equivocado al traducir alguna estructura del lenguaje algorítmico al lenguaje de pregramación usado, bien porque hayamos fallado al utilizar alguna de las características del lenguaje de programación.

El resultado de estos errores es variado:

- El programa no compila: Este es el menor de los problemas. El compilador suele dar indicaciones de qu est pasando. Pero para podre resolver el problema es necesario que lea con atención y entienda los mensajes del compilador. No se fije sólo en la línea del error. Entienda qué está diciéndole el comppilador. Tenga en cuenta que el compilador, cuando encuentra un error aplica una modificación al ficheroo que le ha pasado para intentar arreglar el problema y seguir avanzando. Esas modificaciones pueden ser correctas o no, por lo tanto, siempre son mucho más significativos los primeros errores que los últimos.

- El programa compila, pero da warnings: Nunca deje pasar un warning sin analizar qué lo está generando. Normalmente el compilador da un warning cuando compila algo que es sintácticamente correcto, pero da la impresión de ser absurdo. Normalmente, los warnings indican que el programa no hace lo que debe de hacer. Tiene que recordar que cuando se instala el compilador se hace con unas opciones por defecto. Cuando ejecutamos el compilador podemos modificar dichas opciones mediante opciones del comando. Es habitual que gcc se instale de modo que se inhiban algunos warnings, por ello le recomiendo que compile con la opción -Wall para habilitar todos los warnings. Puede que algunos le resulten absurdos, pero, hasta que no tenga claro lo que significa cada uno, no pase de ellos.

- El programa compila, pero da notas: Normalmente una nota indica que el compilador ha visto algo que sintácticamente es correcto, semánticamente no parece que afecte al correcto funcionamiento del programa, pero tiene una pinta unpoco rara. Suelen indicar problemas que afectan a eficiencia, aunque, a veces, también indican errores. Analicelas e intente eliminarlas.

- El programa compila, pero no hace lo que tiene que hacer. Me refiero a situaciones en las que el programa 'se cuelga' (posible bucle infinito), no ofrece los resultados esperados (a menudo, problema de error de codificación), o finaliza con un error y diciendo no se que de core dumped (a menudo violaciones por accesos a zonas de memoria incorrectas a través de un puntero con un valor erróneo o un acceso fuera de las posiciones reservadas para un array). En estos casos tenemos que mirar dónde está fallando. Normalmente, entre los alumnos de primeros cursos, nos encontramos con tres posibles estrategias:

  1. Empezar a cambiar líneas de código a lo loco a ver si suena la flauta. Obviamente esto no es una estrategia sino una granperdida de tiempo. Mucha gente lo hace, pero no sólo no es recomendable, es, directamente, castigable.
  2. Modificar el archivo fuente escribiendo printf() por todos los lados intentando comprobar cual es valor de las variables en diferentes zonas del programa o comprobando si se entra o sale de algunas funciones. Tampoco es una buena estrategia por muchos motivos. Para empezar, hay que recordar que printf() no escribe directamente en 'la pantalla', sino en un buffer. O se fuerza el vaciado del buffer en pantalla (con fflush(), aunque, a veces, funciona con un '\n' al final) o es posible que interpretemos muy mal lo que está pasando. Por otra parte, ensuciamos la ejecución del programa, cosa que los capullos de los profesores (curiosamente, también los clientes) suelen penalizar. Ello lleva a que después de corregir el programa tengamos que borrar todos los printf que habíamos escrito. No se puede ni imaginar cuántos alumnos han suspendido porque en ese borrado han borrado también cosas que no tenían que borrar y han entregado programas que no estaban completos. En resumen, esta tampoco es una estrategia de éxito.
  3. Utilizar un debugger para analizar el programa: Un debugger es un programa creado para hacer, de modo correcto, algo parecido a lo de los printf(). Simplemente, el debugger corre nuestro programa y nos permite ver cuánto valen las variables en puntos concretos e, incluso, interaccionar con él. Realmente, esta debería ser la única estrategia

7.2 El debugger de Gnu: comando gdb

En el mercado existen múltiples debuggers. De hecho, incluso muchos IDEs incorporan sus propios debuggers o incorporan interfaces a debuggers externos. Uno de los más populares es el debugger creado por GNU. GNU, acrónimo recursivo de GNU's not Unix (GNU no es Unix) es un grupo que crea muchas aplicaciones de software libre (puede utilizar este link para acceder).

GDB, el proyecto debugger de GNU, es un debugger muy completo. Permite ver lo que ocurre 'dentro' del programa mientras se ejecuta o lo que estaba haciendo un programa cuando fallo. Existen versiones para Windows, Unix/Linux e incluso OS/X. Puede, además, usarse con programas escritos en muchos lenguajes de programación, entre ellos C. Puede bajarse la versión que le interesa desde este link. En el laboratorio virtual ya está instalado (y algunas de las cosas que le cuento posteriormente puede que no funcionen exactamente igual en otros sistemas operativos).

El problema de gdb es que funciona en modo línea. Es decir, cuando lo instale lo que tiene no es un programa de entorno gráfico, sino un nuevo comando: gdb.

7.3 Una sesión básica de *debuggeo*

Como le he indicado, `gdb` es un comando muy potente. Y, como suele suceder, no muy sencillo de usar. La documentación es extensa, comopuede comprobar en en este link. Pero no es necesario tener un conocimiento extenso para sacarle provecho. En este documento pretendo explicarle cuatro nociones básicas con las que puede empezar a disfrutar de las ventajas de este comando. Lo de profundizar más en él ya es cosa sulla[1].

Lo primero que hay que tener claro es que `gdb`, al igual que el resto de debuggers, no pueden usarse con programas 'normales'. Necesitamos compilar de una forma diferente para el debugger. Cuando compilamos para el debugger se genera un ejecutable que se puede ejecutar en modo normal, pero es mucho más grande y lento, pues incorpora todo lo necesario para que el debugger pueda realizar su labor.

La opción que debe emplear para crear un ejecutable *debuggeable* con `gcc` es `-g`. Y tiene que emplearla con todos los ficheros que pretenda debuggear. Es decir, si va a compilar para debuggear una aplicación con un programa principal y un módulo, debera ejecutar.

NOTA IMPORTANTE: Le recomiendo que, a partir de ahora, no se limite a leer. Utilice algún programa suyo (por ejemplo la práctica 4.1, para ir viendo cómo funcionan las cosas que le digo). También le recomiendo que maximice (o al menos haga bastante grande) la ventana de comandos para poder ver correctamente los efectos de los comandos.

```
gcc -g -c modulo.c
gcc -g principal.c modulo.o
```

El resultado es el programa ejecutado en el fichero `a.out`. Para comenzar a ejecutar el debugger de modo que veamos los posibles fallos de ese programa ejecutamos ahora

```
gdb -tui a.out
```

Quiero aclarar un par de cosas. En realidad la instrucción a ejecutar es `gdb a.out`. El problema es que, como le he dicho, `gdb` es un debugger en modo línea. Tiene que recordar todas las instrucciones y, a la vez tener muy claro cómo es su programa (incluso el número de cada línea. la opción `-tui` pone en marcha un interface de texto (programado con ncurses) que le facilita mucho la vida. Es recomendable que antes de lanzar este comando maximice la pantalla de la ventana de comandos pues la ventana que crea ncurses para gdb es del tamao de la ventana de comandos. En algunas versiones de sistema la ventana de ncurses se adapta a las modificaciones de tamao de la ventana de comandos, pero no en todas. Si no, es posible que no tenga espacio en la ventana que se crea para toda la información que debe contener.

[1] Para este curso no es necesario. Para otras asignaturas, ... depende. Como le he dicho, muchos IDEs incorporan sus propios debuggers, por lo que es posible que no use gdb en muchas asignaturas. Pero estoy seguro (no deja de ser una opinión) de que lo usará alguna vez tanto en la uni como en su vida profesional.

Al ejecutarse este comando, la ventana de comandos se estructura en dos partes. En la de arriba tiene el código del programa y en la de abajo una ventana de comunicación con el debugger. Es en esta parte dónde debe escribir comandos, por donde `gdb` le sacará resultados y también por donde verá todas las salidas del programa que se está debuggeando.

Hay que tener en cuenta un par de problemas que pueden surgir. A veces la parte de arriba se descontrola un poco. Es decir, no está viendo laparte de código que debería ver o ve zonas de código duplicadas. El problema se resuelve situando el raton en esa zona y haciendo scroll hasta que vea lo que quiere. Hay un problema, quizá incluso peor, en la parte de abajo. A veces, sobre todo cuando elprograma le pide que escriba algo, la parte de abajo queda un poco farragosa. Puede incluso que no vea usted la frase en la que su programa le pide datos. Debe siempre tener más o menos claro en que zona del programa está para poder saber exactamente qué está pasando. Si no ve el prompt de `gdb` lo más probable es que su aplicación esté esperando a que usted meta algún dato (es decir, está ejecutando, por ejemplo, una instrucción `scanf()`.

Una vez abierto `gdb` utilizaremos la parte de abajo de la ventana para darle a ordenes al debugger. A continuación le cuento las más importantes.

7.3.1 Puntos de ruptura

La idea es ejecutar el programa y ver, a la vez, el contenido de las variables. Obviamente, lo programas se ejecutan demasiado rápido para que podamos ver nada. Por eso, lo primero que tenemos que hacer es poner puntos de ruptura: Un punto de ruptura es una orden que el programa se pare justo antes de ejecutar una sentencia. Serán esos puntos en los que el programa dónde podremos ver cuánto valen las variables.

Para colocar un punto de ruptura se usa el comando `break` seguido de el nombre de un procedimiento (acción o función) del programa, o un número de línea [2].

En realidad puede poner tan sólo `b` en lugar de `break`. Si escribe, por ejemplo

```
(gdb) break 10
```

siendo la líinea 10 una línea que contenga una instrucción de código (no podemos poner breakpoints en zonas que sean comentarios o declaraciones de tipos), beremos que en la parte de arriba aparece, justo a la izquierda de la línea 10 de código una `b+`. Eso significa que ahí hay un punto de ruptura (b) habilitado (+) y que el programa no está ahora parado en él. Los puntos de ruptura se pueden deshabilitar y entonces pasaríamos a ver `b-`. Cuando el programa está parado en un breakpoint, la parte de arriba muestra `B+`.

En un programa podemos tener el número de puntos de ruptura que queramos. También existen los watchpoints que fuerzan la parada en una línea sólo si se da una condición en los valores del programa, pero su sintaxis es más compleja. Existen incluso Catchpoints, pero son aún más complejos.

Los breakpoints del programa están numerados. Podemos deshabilitar en cualquier momento cualquiera de ellos mediante : la instrucción :

```
(gdb) disable breakpoints n1, n2
```

[2]Hay otras opciones, como en todo lo que le cuento en este documento, pero estas son las más sencillas

Donde `breakpoints` podemos no escribirlo si no tenemos watchpoins ni catchpoints y n1, n2 son los números que identifican a los breakpoints que queremos deshabilitar. También podemos volver a habilitar breakpoints deshabilitados mediante:

```
(gdb) eneble breakpoints n1, n2
```

7.3.2 Ejecución del programa

Una vez tenemos puntos de ruptura estratégicamente situados podemos ejecutar el programa. Para ello usamos el comando:

```
(gdb) run
```

El programa comienza a ejecutarse hasta que encuentre un punto de ruptura o una instrucción que requiera que el usuario introduzca un dato. En el segundo caso debe introducir el dato, como se ha dicho, en la parte de abajo y pulsar enter[3].

Cuando estamos parados, la línea que está a punto de ser ejecutada está remarcada en la ventana de arriba.

Una vez el programa está detenido (salvo cuando lo está esperando una entrada por parte del usuario), si queremos continuar con la ejecución del programa, tenemos tres alternativas

1. Continuar con la ejecución:

   ```
   (gdb) continue
   ```

   Este comando le dice al debugger que siga ejecutado hasta que se alcance el siguiente punto de ruptura. Existe la opción de poner un número detrás de `continue`. Este número indica el número de breakpoints que se va a 'saltar' la ejecución hasta que vuelva a pararse (es potente, pero peligroso si no tiene claro dónde están sus puntos de ruptura). También podemos escribir `c` en lugar de `continue`.

2. Ejecutar la instrucción dónde estamos parados

   ```
   (gdb) next
   ```

   Este comando le dice al debugger que ejecute la sentencia en la que está parado y se pare antes de ejecutar la siguiente. Existe la opción de poner un número detrás de `next`. Este número indica el número de sentencias que queremos ejecutar. También podemos escribir `n` en lugar de `next`.

3. Ejecutar la instrucción dónde estamos parados, pero entranso en el código de procedimientos

   ```
   (gdb) step
   ```

[3]Esto complica mucho el debuggeo de programas que usen ncurses

Este comando le dice al debugger que ejecute la sentencia en la que está parado y se pare antes de ejecutar la siguiente. Si la sentencia en la que estamos parados incluye la llamada a un procedimiento, se considera que la siguiente sentencia es la primera del procedimiento. Es decir, a diferencia de `next`, `step` nos permite pasar a analizar el código de acciones y funciones. Existe la opción de poner un número detrás de `step`. Este número indica el número de sentencias que queremos ejecutar. También podemos escribir `s` en lugar de `step`.

Un dato interesante es que si pulsa enter sin darle ninguna orden a `gdb` él siempre decide que lo que uste desea es ejecutar la última orden.

Otro dato interesante es que si ejecuta `run` cuando ya se ha iniciado la ejecución del programa, vuelve (una vez le confirma a `gdb` que pretende hacer semejante locura) simplemente reinicia la ejecución.

7.3.3 Acceso a los datos

Como ya hemos comentado, cuando estemos parados (en un breakpoint o depues de ejecutar `next`, `continue` o `step` tenemos acceso a todos los valores accesibles en el entorno en el que se está ejecutando. Es decir, constantes y variables globales, variables locales y parámetros reales. Para ver su contenido podemos ejecutar el comando de `gdb`:

`(gdb) print` *expresion*

La expresión es cualquier expresión correctamente escrita en el mismo lenguaje que el lenguaje fuente. Es decir, el nombre de una variable, lo apuntado por un puntero o un acceso complejo a variables válidas como a[2*p.t*].

A veces nos interesa mucho más seguir la evolución de una o varias variables según va evolucionando el programa. Ello nos obligaría a estar escribiendo múltiples veces la instrucción `print`. Para evitarlo, podemos emplear el comando

`(gdb) display` *expresion*

Una vez ejecutemos esto, en cada punto de parada veremos en la parte de abajo el valor de la expresión. Si hacemos display de muchas cosas, cada una de ellas está identificada por un número. Podemos eliminar los displays haciendo:

`(gdb) undisplay` *numero*

o bien deshabilitar los display con `disable display numero` y volver a habilitarlos con `enable display numero`.

Debe quedar claro que si entramos en un procedimiento, bien usando `step` o bien porque llegamos a un breakpoint que está dentro de una función, los valores que se ven afectados por el display son los que hagan referencia a dicha función. Es decir, suponga que está ejecutando un programa que tiene una variable local a y que ha hecho display a. Supongamos que ejecuta un `step` con lp que accede al código de otra función. Bien, pues como esa a no es una variable de esa función, el display no está definido dentro de la función. No se lie. Es posible que esa función tenga otra variable llamada a. Si también se hecho desplay de esa otra variable, veremos ahora su valor, y no el de la externa.

7.3.4 Un poco de organización

Y con esto ya puede usted debuggear.

Bueno, con esto y un poco de lógica. No conviene que se limite a poner un breakpoint en la primera instrucción del programa y vaya haciendo `step`hasta el final tras haber hecho un `display` de todas las variables. Lo lógico es que piense dónde puede estar el problema y coloque breakpoints antes de ese punto para ver que los valores que llegan allí son los esperados y después hacer `next`. Si tras la ejecución de la sentencia los valores que comprueba están bien, lo lógico es que el contenido de la sentencia sea correcta. Si no, empiece de nuevo y ahora entre con `step` a esa función. Y así hasta encontrar el problema.

Que conste, debuggear es un arte que sólo se aprende con la práctica, así que le recomiendo que practique.

Ah, para cerrar el debugger basta con ejecutar `quit`, aunque basta con poner `q`