Depurador GDB

Juan Jose Moreno Moll

Índice

1. Introducción

La realización de un programa en cualquier lenguaje de programación es un proceso más o menos largo que debe terminar en la escritura de un código del programa que '*funcione correctamente*'. Que un programa funcione correctamente significa que proporcione los '*resultados esperados*', es decir que su comportamiento real coincida con el comportamiento esperado. Un programa es correcto si no tiene errores de ejecución y si se cumple las especificaciones escritas en la fase de especificación.

En el proceso de escritura de un programa aparecen muchos errores que debemos ir corrigiendo hasta llegar a escribir el programa correctamente. Los tipos de errores que podemos encontrarnos son:

* Errores de sintaxis: Son errores producidos por la incumplimiento de las reglas sintácticas del lenguaje que estemos utilizando. Hasta que no se corrigen estos errores no podemos obtener un código ejecutable del programa. Este tipo de errores se corrigen mediante el **compilador**.
* Errores de ejecución: Son errores que se producen cuando se realiza una operación no válida que da lugar a la finalización anormal del programa (división por cero, abrir ficheros que no existen, acceder a zonas de memoria no permitidas, etc.).
* Errores lógicos: Son errores debidos a la traducción incorrecta de las especificaciones del programa. El programa termina normalmente pero no produce los resultados esperados.

Un **depurador** es una herramienta de programación que nos permite la detección y corrección de errores de ejecución y errores lógicos. Este tipo de errores aparecen a partir del momento en el que tenemos un código sintácticamente correcto del que obtenemos un programa ejecutable. Es al probar este programa ejecutable cuando se puede detectar la aparición de errores de ejecución y lógicos.

El depurador nos permite ejecutar el programa de tal forma que es posible ir acotando las zonas donde se están produciendo errores, observar bajo que valores o condiciones se dan éstos, etc. Esta información nos ayudará a modificar el código para corregir y evitar fallos del programa.

Hay que advertir que un depurador ayuda a solucionar problemas de incorrección puntuales de los programas, pero en ningún caso sirve para arreglar un programa mal especificado o mal concebido. Un programa mal especificado significa un tener que construir un programa del que no sabemos exactamente que debe hacer (o que no debe hacer), y jamás se podrá decir que funciona bien o mal. Y un programa mal concebido (mal diseñado o con ausencia total de diseño) no tiene solución posible.

La depuración de programas es una tarea bastante complicada, sobre todo con depuradores en línea como **gdb**, por lo que para facilitar esta tarea los programas deben estar bien especificados y diseñados y estar escritos con la máxima claridad (legibilidad de los programas). La depuración de un programa se facilita si es correcta su especificación, diseño y legibilidad y nunca a la inversa.

La mejor herramienta de depuración es evitar los errores desde el principio.

Muchos lenguajes y plataformas vienen acompañados con este tipo de herramientas. El funcionamiento de un depurador es muy semejante al funcionamiento del depurador que se explica en este documento, **gdb**. Por ejemplo el entorno de programación de Pascal TURBO-PASCAL o el de C TURBO-C vienen con este tipo de herramienta integrada dentro su interfaz gráfico. El interfaz gráfico facilita la utilización del depurador, siempre que se sepa manejar correctamente. Esto significa que la utilización sin conocimiento de causa de'f7' y/o 'f8' en los entornos anteriormente mencionados es bastante inútil. (Por ejemplo muchas personas habrán observado lo aburrido y cansado que resulta utilizar esas teclas cuando llegamos a una sección de código con 2 ó 3 bucles 'for' anidados con variables de control con valores medios).

La elección de un depurador en línea se debe a que con él no es posible hacer un uso inconsciente y mecánico, que fácilmente se da en depuradores gráficos. El conocimiento de un depurador en línea aumenta de forma muy significativa la productividad y el buen uso de depuradores gráficos.

A partir de este momento vamos a explicar algunas de las características y comandos más importantes del depurador **gdb**. Las características y comandos de otros depuradores de C o de otros lenguajes suelen ser muy semejantes. Una descripción completa de este depurador se puede encontrar en su manual.

El depurador permite ver que está haciendo un programa por 'dentro' mientras se ejecuta. **gdb** permite hacer cuatro cosas que ayudan a detectar y corregir errores dentro del código del programa:

1) Comenzar a ejecutar un programa especificando cualquier cosa que pueda afectar a su comportamiento.
2) Hacer que el programa se pare en cualquier línea de código. Se puede indicar incluso bajo que condiciones se debe parar.
3) Examinar que ocurre cuando el programa está parado.
4) Cambiar cosa dentro del programa que está en ejecución.

2. Preparar programas ejecutables para poder utilizarlos con el depurador

Un depurador permite la ejecución controlada del programa de forma que se puede establecer una relación entre el código fuente y el código que se está ejecutando. De esta forma el depurador puede indicar a que línea del código fuente corresponde la instrucción (o instrucciones) de código ejecutable que en ese momento se está ejecutando.

Para poder hacer esto es necesario que haya una equivalencia total entre código fuente (escrito en C u otro lenguaje) y el código ejecutable del programa. Ocurre que en general un compilador cuando genera el código ejecutable de un programa realiza muchas tareas de optimización que hacen que no haya una equivalencia entre código fuente y código ejecutable. Por este motivo hay que indicar al compilador que no realice ninguna tipo de optimización para poder utilizar el código en el depurador. Cada vez que queramos depurar un programa debemos compilar el programa con la opción de no optimización. En habitual que la opción que indica al compilador que no realice tareas de optimización es '-g' (no sólo en C , sino en muchos otros lenguajes). En que en el caso del compilador '**gcc**' esta opción es también '-g'.

Todos los ficheros que componen el programa deben compilarse con esta opción. Por ejemplo en el caso de que queramos depurar un programa que utiliza un archivo fuente llamado 'vector.c' y que tiene como fichero principal a 'prog.c' debemos compilar el programa con las siguientes órdenes:

```
c:...> gcc -g vector.c
c:...> gcc -g prog.c vector.o -o prog
```

Todos los ficheros objeto que componen el programa deberían estar compilados con la opción '-g'. En el caso del gcc además es necesario indicar que el **programa ejecutable no tenga extensión (...-o prog).** De esta forma el compilador produce dos ficheros de salida: *prog* y *prog.exe*. El segundo de ellos (*prog.exe*) es el programa ejecutable habitual que podemos utilizar como siempre. El primero (*prog*) es un fichero escrito en un formato especial (COFF) y es el que el depurador lee. Así para utilizar el depurador con el programa anterior debemos utilizar la siguiente orden:

```
c:...> gdb prog
```

Como es obvio, cuando finaliza el proceso de depuración y tenemos el programa escrito correctamente, éste se debe compilar de nuevo sin la opción '-g' para permitir al compilador hacer todas aquellas tareas de optimización que pueda.

3. Utilizar GDB

Para depurar un programa se debe ejecutar el depurador al que se le pasa como parámetro el nombre del ejecutable del programa que queremos depurar. Como se ha dicho en la sección anterior el programa debe ser compilado con una opción especial.

A partir de este momento **gdb** puede leer una serie de comandos que sirven para depurar el programa. Los comandos se escriben en una línea que **gdb** toma como entrada. Los comandos realizan consultas u otras operaciones sobre el programa que se está depurando. Los comandos pueden tomar argumentos.

Como antes se ha dicho para depurar un programa debemos ejecutar en la línea de ordenes:

```
c:...> gdb prog
```

De esta forma el depurador **gdb** carga el programa. En ese momento el programa puede comenzar a depurarse. Después de ejecutar la línea anterior en pantalla aparece:

(gdb)

La línea anterior es la línea de ordenes (o de comandos) de **gdb.** En esta línea es donde se escriben los comandos que permiten la depuración del programa. Las respuesta a los comandos y la salida del programa también se presentan en está línea.

Para explicar los comandos más habituales de **gdb** vamos a utilizar un programa llamado 'prog' para ilustrar los ejemplos. Todos los ejemplos estarán escritos en cursiva. Aquellos comandos, entradas, etc que el usuario debe escribir se resaltarán en negrrita. Este programa está compuesto por tres ficheros: *prog.c, libes.c, es.h*. Este programa lee una cadena de caracteres y la muestra por pantalla. El código del programa es el siguiente:

```
/* ********************** PROG.C ************** */
/* Ejemplo de la funcion 'lee_cadena' de la libreria "libes.c" */

#include <stdio.h>
#include "es.h"

#define TAM 10


main()
{
  char cad[TAM];
  int x, y, z;

  puts("Programa de prueba para lectura de una cadena.");
  printf("introduce una cadena: ");
  fflush(stdout);
  lee_cadena(cad, TAM);
  printf("cadena: %s\n", cad);

  printf("Escribir valor x: ");
  scanf("%d", &x);
  printf("Escribir valor y: ");
  scanf("%d", &y);
  z = x + y;
  printf("Los valor son x = %d,  y = %d,  x + y = %d\n", x, y, z);

  return 0;

}
```

```
/**********************LIBES.C ***************************** */
/* Libreria de funciones sobre entrada-salida en C. Incluye funciones de
sobre ficheros, lectura de  consola, etc */
/* Fichero de cabecera "es.h" */

#include <stdio.h>
#include <conio.h>
#include "es.h"

/* ************** lee_cadena(char *cad,int n) ******************* */
/* Proposito: Lee una cadena de consola. Lee caracteres hasta que se pulsa
el retorno de carro o se han pulsado 'n' caracteres. Dentro de los caracteres
pulsados se incluye el retorno de carro que se sustituye por un '\0' (final de
cadena en C). Esta funcion tiene eco en pantalla.
Comentarios: a) Seguramente solo sirve para compiladores que tienen la
funcion 'getche()' cuyo prototipo esta en conio.h (por ejemplo TurboC).
             b) La cadena cad debe tener memoria reservada.

  Entradas:  cad: Cadena en la que se colocaran los caracteres leidos por
             consola.
             n: numero maximo de caracteres que se permite leer de
consola.
  Salida:    cad: Sobre 'cad' se colocan los caracteres leidos por consola'
  Devuelve: puntero a la cadena 'cad'.   */

char *lee_cadena(char *cad, int n)
{
  char c;
  int i=0;

  do{
     c=getche();
     cad[i]=c;
     i++;
  }while ((c!='\r')  && (i<(n-1)) );
  cad[i]='\0';
  puts("");
  return cad;
}
/* ********************** ES.H ******************************** */
/* Fichero de cabecera de la libreria "libes.c" */
#ifndef _ES_
  #define _ES_

  char *lee_cadena(char *cad, int n);
#endif
```

3.1 Tareas principales de un depurador

Las tareas principales de un depurador son:

1) Ejecutar un programa paso a paso (línea a línea) ,o ejecutar un sección del programa
2) Colocar puntos de ruptura o 'breakpoints' en una línea del programa. El depurador detiene la ejecución del programa cuando llega una línea de código con un punto de ruptura.
3) Consultar el valor de cualquier variable o expresión.
4) Mostrar valores de variables o expresiones cada vez que se llega a un punto de ruptura o cuando estamos haciendo una ejecución línea a línea.
5) Consultar pila de llamadas de las funciones.
6) Indica cuando se produce un error de ejecución el estado de la pila de llamadas a función, los valores de las variables, etc.

3.2. Lanzar a GDB

Para comenzar la depuración podemos ejecutar las siguientes órdenes desde la línea de órdenes de la shell:

c:\..> *gdb* programa

En este caso programa es el nombre del ejecutable (compilado de forma conveniente) del programa que queremos depurar.

También es posible lanzar a **gdb** para obtener información sobre un error de ejecución de un programa. En algunos sistemas opertivos (UNIX) cuando ejecutamos un programa y se produce error de ejecución, el sistema crea un fichero llamado generalmete '**core'** que es una imagen del código del programa en el momento que se ha producido el error. El depurador puede leer este fichero y darnos información del programa en el momento en el que se produce el error (valores de las variables, línea donde se produce error, pila de llamadas de fucniones, etc). Si al ejecutar un programa se produce un error de ejecución y el sistema produce un 'core', entonces la forma de lanzar el depurador tener información sobre el error es:

c:\..> *gdb* programa *core*

En ambos casos **gdb** carga el programa que está dispuesto para empezar a ejecutar, para consultar valores de variables, etc. Aparecerá en la pantalla la siguiente línea:

(gdb)

A partir de este momento podemos empezar a escribir comandos de **gdb**.

En el caso de nuestro programa 'prog' debemos realizar las siguientes operaciones:

1) Compilar los ficheros de forma adecuada:

c:\...> ***gcc -c -g libes.c***
c:\...> ***gcc -g prog.c libes.o -o prog***

Después de realizar esto deben aparecer en el directorio los ficheros: 'libes.o', 'prog.exe' y 'prog'.

2) Lanzar el programa con **gdb** para comenzar la depuración:

c:\...> ***gdb prog***

El depurador carga el programa y lo indica presentando en pantalla la línea de comandos del depurador:

(gdb)

4. Comenzar y finalizar la ejecución de un programa dentro de GDB

Para ejecutar el programa dentro de **gdb** tenemos que usar el comando, *run.* El comando tiene una abreviación *r*. Las dos formas de comanzar la ejecución son:

(gdb) ***run***
(gdb) ***r***

La ejecución no se detiene hasta que el programa encuentra una entrada de datos o un 'breakpoint'. La salida del programa se muestra según se va ejecutando el programa. Cuando el programa necesita una entrada de datos debemos suministrársela para que el programa continúe su ejecución.

Si el programa necesita argumentos se introducen como argumentos del comando *run*:

(gdb) ***run lista_argumentos***

Para volver a comenzar la depuración del programa simplemente hay que volver a ejecutar e comando *run*. Todos los 'breakpoints', 'whatchpoints', etc. colocados en la primera depuración se conserva.

Es posible en cualquier comento consultar los argumentos y las variables de entorno del programa:

(gdb) ***show args***
(gdb) ***show env***

Si queremos salir del depurador tenemos el comando *quit*, y su abreviación *q*:

(gdb) ***quit***
(gbd) ***q***

En nuestro ejemplo podemos probar el comando *run*:

(gdb) ***run***
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.12 (go32), Copyright 1994 Free Software Foundation, Inc...
Starting program: c:/users/jmoreno/prog_ii/lib/ej1
Programa de prueba para lectura de una cadena.
introduce una cadena: ***esto_cad***
cadena: esto_cad
Escribir valor x: 1
Escribir valor y: 3
Los valores son x = 1, y = 3, x + y = 4

Program exited normally.
go32_close called
(gdb) ***q***

5. Parar y continuar una ejecución.

La tarea del depurador no es ejecutar un programa como puede hacerse desde la línea de órdenes. Su principal tarea el poder detener la ejecución del programa antes de que ésta finalice y poder investigar que ocurre en cualquier punto del programa. Hay dos formas de hacer que el programa se detenga antes de finalizar: los '*breakpoints*' y los '*watchpoints*'.

5.1. Breakpoints

Un *breakpoints* permite que el programa se detenga en cualquier línea del programa. Un *breakpoint* puede tener condiciones que indican al depurador si debe parar o no. Generalmente cuando colocamos un breakpoint indicamos en que la línea del programa debemos pararnos. Es normal que un *breakpoint* se coloque inmediatamente antes de los puntos del programa en los que creemos que tenemos errores o problemas.

El comando para colocar un *breakpoint* es *break* (o abreviado *b*): Este comando tiene argumentos que indican donde se colocan los *breakpoints*. Algunas formas de colocar un *breakpoint* son:

break: Coloca un *breakpoint* en la línea siguiente a la que estamos en ese momento.
break línea: Coloca un *breakpoint* en la línea indicada de código del programa.
break +num:
break -num: Coloca un *breakpoint* en algún número de líneas antes (o después) de la línea en la que estamos parados actualmente.
break nombre_función: Coloca un *breakpoint* al comienzo de la función indicada.
en la línea indicada de código del programa.

break fichero:línea: Coloca un *breakpoint* en la línea indicada de código del fichero ***fichero*** del programa. Esta forma de utilizar comandos se utiliza cuando el programa está formado por más de un fichero.

break fichero:función: Coloca un *breakpoint* al comienzo de la función del fichero indicado. Esta forma de utilizar comandos se utiliza cuando el programa está formado por más de un fichero.

Otros comandos relacionados con *breakpoints* son:

info break: Indica todos los *breakpoints* colocados en el programa. Los *breakpoints* están numerados.

clear: Borra el *breakpoint* colocado en la instrucción siguiente.

clear [fichero]:línea: Borra el *breakpoint* colocado en la línea (del fichero) indicada.

clear [fichero]:nombre_función: Borra el *breakpoint* colocado en la función (del fichero) indicada.

delete: Borra todos los *breakpoints*.

delete n: Borra el *breakpoint* ***n***.

disable n: Desactiva el *breakpoint* ***n***. El programa no se detendrá en este punto hasta que volvamos a habilitar otra vez el *breakpoint*.

ignore n m: Desactiva el *breakpoint* ***n m*** veces.

enable n: Habilita el *breakpoint* ***n***.

5.2. Watchpoints

Un *watchpoints* permite parar la ejecución del programa cuando una expresión o variable cambia. No es necesario indicar un lugar determinado del programa donde colocar el *watchpoint*.

Los *watchpoint* hacen que los programas se ejecuten mucho más lentamente pero permiten encontrar errores que no sabemos donde se producen. La situación más habitual en la que se usa este comando es cuando tenemos alguna variable que no debía cambiar de valor, que cambia a un valor no deseado, en un momento no deseado, etc. Entonces lo que hacemos es asignar un *watchpoint* a esa expresión o variable y ejecutamos el programa. El programa se detendrá en el momento en que la expresión cambia de valor. Así podremos ver la instrucción o el punto de programa en el cambia la expresión.

El comando para colocar un *whatchpoint* es *watch*. Este comando tiene argumentos que indican que expresión queremos controlar:

watch expresión: Coloca un *watchpoint* en una expresión

Otros comandos relacionados con *breakpoints* son:

info watch: Indica todos los *watchpoints* colocados en el programa. Los *watchpoints* están números.

Los comandos *clear*, *enable* y *disable* también se pueden utilizar con *watchpoints*.

5.3. Continuar la ejecución y ejecución paso a paso

Una vez que se ha detenido el programa en *breakpoint* o un *watchpoint* es necesario que el programa continue la ejecución. Hay dos formas de continuar la ejecución del programa. Una de ellas es continuar hasta que encuentre otro *breakpoint* o *watchpoint.* Otra es continuar la ejecución instrucción a instrucción.

El comando para continuar la ejecución hasta el siguiente *breakpoint* o *watchpoint* es *continue* (o abreviado *c*):

(gdb) ***continue***

Otra posibilidad es continuar la ejecución instrucción a instrucción. Esta es una forma de seguir el programa de forma muy precisa y se utiliza para hacer la traza del programa es trozos del programa especialmente difíciles. Si ejecutamos el programa de está forma se muestra cada vez la instrucción que se acaba de ejecutar. Hay dos formas de ejecutar el programa instrucción a instrucción:

- ***step***: Abreviatura ***s*** . Ejecuta la siguiente instrucción. Si encuentra una llamada a una función la ejecución entra dentro del código de la función. Una llamada a función no se considera una instrucción.
- ***next***: Abreviatura ***n*** . Ejecuta la siguiente instrucción. Si encuentra una llamada a una función la ejecución no entra dentro del código de la función. Una llamada a función se considera una única instrucción.

5.4. Ejemplos de los comandos break. info, delete, continue, next, step

Ejemplo de utilización de los comandos *break, info, delete* y *continue*. En este ejemplo vamos a colocar dos puntos de ruptura (*breakpoint*), a informarnos sobre ellos (*info*), a eliminar un punto de ruptura (*delete*) y a permitir que el programa continúe su ejecución después de una parada en un punto de ruptura (*continue*):

```
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (go32), Copyright 1996 Free Software Foundation, Inc...
(gdb) b 15
Breakpoint 1 at 0x15ef: file ej1.c, line 15.

(gdb) b  lee_cadena
Breakpoint 2 at 0x16a6: file libes.c, line 31.

(gdb) r
Starting program: c:/users/jmoreno/prog_ii/lib/ej1
```

```
Breakpoint 1, main () at ej1.c:15
15      puts("Programa de prueba para lectura de una cadena.");

(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x000015ef in main at ej1.c:15
       breakpoint already hit 1 time
2   breakpoint     keep y   0x000016a6 in lee_cadena at libes.c:31

(gdb) d 1

(gdb) info break
Num Type           Disp Enb Address    What
2   breakpoint     keep y   0x000016a6 in lee_cadena at libes.c:31

(gdb)  c
Continuing.
Programa de prueba para lectura de una cadena.
introduce una cadena:
Breakpoint 2, lee_cadena (cad=0x4da8c "é1", n=10) at libes.c:31
31      int i=0;
(gdb) c
Continuing.

cadena: la_cadena
Escribir valor x: 1
Escribir valor y: 3
Los valor son x = 1,  y = 3,  x + y = 4

Program exited normally.
0x10 in ?? ()
(gdb) q
```

Ejemplo de utilización de los comandos *break* y *next*. En este ejemplo vamos a colocar un punto de ruptura antes de la llamada a la función 'lee_cadena()'. Veremos que una ejecución paso a paso con *next* no entra en el código de la función.

```
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (go32), Copyright 1996 Free Software Foundation, Inc...
(gdb) b 16
Breakpoint 1 at 0x15fc: file ej1.c, line 16.

(gdb) r
Starting program: c:/users/jmoreno/prog_ii/lib/ej1
```

```
Breakpoint 1, main () at ej1.c:16
16      printf("introduce una cadena: ");

(gdb) n
17      fflush(stdout);

(gdb) n
Programa de prueba para lectura de una cadena.
introduce una cadena:
18      lee_cadena(cad, TAM);
la_cadena

(gdb) n
19      printf("cadena: %s\n", cad);
(gdb) c
Continuing.

cadena: la_cadena
Escribir valor x: 1
Escribir valor y: 3
Los valor son x = 1,  y = 3,  x + y = 4

Program exited normally.
0x10 in ?? ()
(gdb) q
```

Ejemplo de utilización de los comandos *break* y *step*. En este ejemplo vamos a colocar un punto de ruptura antes de la llamada a la función 'lee_cadena()'. Veremos que una ejecución paso a paso con *step* entra en el código de la función.

```
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (go32), Copyright 1996 Free Software Foundation, Inc...
(gdb) b 16
Breakpoint 1 at 0x15fc: file ej1.c, line 16.

(gdb) r
Starting program: c:/users/jmoreno/prog_ii/lib/ej1

Breakpoint 1, main () at ej1.c:16
16      printf("introduce una cadena: ");

(gdb) s
17      fflush(stdout);
```

```
(gdb) s
Programa de prueba para lectura de una cadena.
introduce una cadena:
18      lee_cadena(cad, TAM);

(gdb) s
lee_cadena (cad=0x4da8c "é1", n=10) at libes.c:31
31      int i=0;

(gdb) s
34          c=getche();

(gdb) c
Continuing.

cadena: la_cad
Escribir valor x: 1
Escribir valor y: 2
Los valor son x = 1,  y = 2,  x + y = 3

Program exited normally.
0x10 in ?? ()
(gdb) q
```

6. Consultar valores de los datos y expresiones del programa

Uno de los cometidos más importantes del depurador es la inspección de cualquier valor de una variable o expresión del programa. De esta forma podemos comprobar si el programa opera con los datos de la forma deseada. Tenemos dos comandos que cumplen esta función *print* y *display.*

6.1. Consultar datos y expresiones

La forma habitual de consultar el valor de un dato o de una expresión es con el comando *print* (o abreviado *p*). Este comando devuelve el valor de cualquier expresión o variable que se especifique en el formato elegido. La forma de usar el comando es:

print exp:

print /f exp: Muestra el valor de la expresión *exp* (en lenguaje fuente) en el formato adecuado al tipo de la expresión *exp*. Para elegir un formato distinto al habitual se puede indicar con *'/f'*, donde *f* es una letra que indica el formato.

Los formatos posibles son:

- x: Escribe la variable o expresión en forma hexadecimal.
- d: Escribe la variable o expresión en forma decimal con signo.
- u: Escribe la variable o expresión en forma decimal sin signo.
- o: Escribe la variable o expresión en forma octal.
- t: Escribe la variable o expresión en binario.
- a: Escribe la variable o expresión como si fuera una dirección. La dirección la escribe con dos números hexadecimales, el segmento y el desplazamiento.
- c: Escribe la variable o expresión como un carácter.
- f: Escribe la variable o expresión como un número en coma flotante.

Un ejemplo de como se utiliza este comando puede ser:

```
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (go32), Copyright 1996 Free Software Foundation, Inc...

(gdb) b 21
Breakpoint 1 at 0x1635: file ej1.c, line 21.

(gdb)  b   25
Breakpoint 2 at 0x1671: file ej1.c, line 25.

(gdb) r
Starting program: c:/users/jmoreno/prog_ii/lib/ej1
Programa de prueba para lectura de una cadena.
introduce una cadena:
Breakpoint 1, main () at ej1.c:21
21      printf("Escribir valor x: ");

(gdb) p  cad
$1 = "pepe\r\000\000\000¤Ú"

(gdb) p /x cad
$2 = {0x70, 0x65, 0x70, 0x65, 0xd, 0x0, 0x0, 0x0, 0xa4, 0xda}

(gdb) p /c cad
$3 = {112 'p', 101 'e', 112 'p', 101 'e', 13 '\r', 0 '\000', 0 '\000',
  0 '\000', -92 '¤', -38 'Ú'}

(gdb) p /d cad
$4 = {112, 101, 112, 101, 13, 0, 0, 0, -92, -38}

(gdb) c
Continuing.
```

cadena: pepe
Escribir valor x: 1
Escribir valor y: 3

Breakpoint 2, main () at ej1.c:25
25 z = x + y;

(gdb) p x
$5 = 1

(gdb) p /x x
$6 = 0x1

(gdb) p (x-y)
$7 = -2

(gdb) p /u (x-y)
$8 = 4294967294

*(gdb) p (2*x/y)*
$9 = 0

*(gdb) p (2*x/(float)y)*
$10 = 0.66666666666666663
(gdb) c
Continuing.

Los valor son x = 1, y = 3, x + y = 4

Program exited normally.
0x10 in ?? ()
(gdb) q

6.2. Consulta de variables con mismo nombre y distinto alcance

Un problema que aparece cuando utilizamos el depurador es que aparezcan variables con el mismo nombre en contextos distintos (con alcance distinto). Si tenemos varias variables con el mismo nombre en funciones o incluso en ficheros distintos y queremos consultar sus valores es posible que nos veamos obligados a indicar el nombre de la función o del fichero donde está la variable que queremos consultar. La forma de indicar el fichero o la función en la que está la variable que queremos consultar es:

print funcion::variable
print fichero::variable

6.3. Vectores creados de forma dinámica

Otro problema que tenemos es consultar el contenido de un vector que se ha creado con funciones de memoria dinámica. Por ejemplo si hemos creado un vector en el programa de la forma siguiente:

*vector = (int *) malloc(tam * sizeof(int));*

podemos consultar su contenido con :

print vector@tam

6.4. Mostrar valores de expresiones y variables de forma automática

Es posible que queramos conocer el valor de una variable muy frecuentemente (generalmente para saber en que momento cambia). Para ello podemos utilizar el comando *display*. Este comando muestra el valor de la expresión especificada cada vez que el programa se detiene. La forma de usar el comando es:

display exp:
display /f exp: Muestra el valor de la expresión *exp* (en lenguaje fuente) en el formato adecuado al tipo de la expresión *exp*, cada vez que el programa se detiene. Para elegir un formato distinto al habitual se puede indicar con *'/f'*, donde *f* es una letra que indica el formato.

Asociado a este comando se pueden utilizar los siguientes: *delete display num, disable display num, enable display num, info display.* El significado de estos comandos es análogo a los vistos anteriormente.

7. Consultar la pila

Otro tipo de información que es importante conocer cuando estamos depurando el programa es la pila de llamadas de las funciones. En la pila de llamadas de la función podemos ver como se llaman unas a otras las funciones, el orden de llamada de las funciones y los valores con los que se llaman a las funciones. Esta información es muy útil cuando el programa tiene un error de ejecución porque nos indica que función ha fallado, valores tenían sus parámetros y que función o funciones llamaban a la función que ha fallado. El comando que se utiliza para conocer el estado de la pila de llamadas de funciones es *backtrace* (o abreviado *bt*).

bracktrace: Muestra la pila de llamadas de las funciones del programa.
backtrace n: Muestra los *n* primeras llamadas de la pila de llamadas de las funciones del programa.

En el ejemplo vamos a colocar 2 puntos de ruptura uno en la función 'main()' y otro en la función 'lee_cadena()'. De esta forma podemos consultar la pila de llamadas de las funciones cuando el punto de ejecución está en 'main()' y cuando el punto de ejecución está en la función 'lee_cadena()'. En la primera consulta podemos ver que sólo tenemos en la pila de llamadas a 'main()'. En la segunda consulta vemos que en la pila de llamadas está 'lee_cadena()' y 'main()' por este orden. En la pila también podemos ver los valores que reciben los parámetros de las funciones. Esto significa que la función que se está ejecutando en la segunda consulta es 'lee_cadena()' y que está función ha sido llamada desde la función 'main()'.

```
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (go32), Copyright 1996 Free Software Foundation, Inc...

(gdb) b 14
Breakpoint 1 at 0x15ef: file ej1.c, line 14.

(gdb) b  libes.c:34
Breakpoint 2 at 0x16ad: file libes.c, line 34.

(gdb) r
Starting program: c:/users/jmoreno/prog_ii/lib/ej1

Breakpoint 1, main () at ej1.c:15
15      puts("Programa de prueba para lectura de una cadena.");

(gdb) bt
#0  main () at ej1.c:15
#1  0x1c33 in __crt1_startup ()

(gdb) c
Continuing.
Programa de prueba para lectura de una cadena.
introduce una cadena:
Breakpoint 2, lee_cadena (cad=0x4da8c "é1", n=10) at libes.c:34
34          c=getche();

(gdb) bt
#0  lee_cadena (cad=0x4da8c "é1", n=10) at libes.c:34
#1  0x1621 in main () at ej1.c:18
#2  0x1c33 in __crt1_startup ()

(gdb) delete 2

(gdb) c
```

Continuing.
cadena

cadena: cadena
Escribir valor x: 1
Escribir valor y: 2
Los valor son x = 1, y = 2, x + y = 3

Program exited normally.
0x10 in ?? ()
(gdb) q

8. Alterar la ejecución del programa

Es posible alterar el valor de una variable utilizando el comando *print*. La forma de hacerlo es indicando el nuevo valor de la variable:

print x=4

Este comando cambia el valor de la variable *x* y la fuerza a tener el valor 4.

9. Listar el programa

El depurador muestra la última línea de código que se ha ejecutado cada vez que el programa se detiene. Pero muchas veces necesitamos ver varias líneas de código fuente, anteriores o posteriores a la línea actual, e incluso de otras partes de programa como funciones o fragmentos de código muy alejados de la línea que se ejecuta en ese momento. El depurador puede mostrar líneas del código fuente del programa el comando ***list***. Las formas más habituales de utilizar ***list***:

list num_línea: muestra varias líneas de código fuente centradas alrededor de la línea de código *num_línea*.
list nombre_función: muestra varias líneas centradas alrededor de la primera línea de código de la función *nombre_función.*
list fich:num_línea: muestra varias líneas de código fuente centradas alrededor de la línea de código *num_línea* del fichero *fich*.
list fich:nombre_función: muestra varias líneas centradas alrededor de la primera línea de código de la función *nombre_función* del fichero *fich.*

list: muestra las líneas centradas alrededor de la última línea mostrada.
list +: muestra varias líneas después de la última línea mostrada.
list +: muestra varias líneas antes de la última línea mostrada.

Ejemplo del uso de este comando en el programa que venimos utilizando:

```
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (go32), Copyright 1996 Free Software Foundation, Inc...
(gdb) b 15
Breakpoint 1 at 0x15ef: file ej1.c, line 15.
(gdb) r
Starting program: c:/users/jmoreno/prog_ii/lib/ej1

Breakpoint 1, main () at ej1.c:15
15      puts("Programa de prueba para lectura de una cadena.");

(gdb)  list
10    main()
11    {
12      char cad[TAM];
13      int x, y, z;
14
15      puts("Programa de prueba para lectura de una cadena.");
16      printf("introduce una cadena: ");
17      fflush(stdout);
18      lee_cadena(cad, TAM);
19      printf("cadena: %s\n", cad);

(gdb) l -
1     /* ********************** EJ1.C ************** */
2     /* Ejemplo de la funcion 'lee_cadena' de la libreria "libes.c" */
3
4     #include <stdio.h>
5     #include "es.h"
6
7     #define TAM 10
8
9
(gdb) l +
10    main()
11    {
12      char cad[TAM];
13      int x, y, z;
14
15      puts("Programa de prueba para lectura de una cadena.");
16      printf("introduce una cadena: ");
17      fflush(stdout);
18      lee_cadena(cad, TAM);
19      printf("cadena: %s\n", cad);
```

(gdb) c
Continuing.
Programa de prueba para lectura de una cadena.
introduce una cadena:
cadena: dddf
Escribir valor x: 1
Escribir valor y: 3
Los valor son x = 1, y = 3, x + y = 4

Program exited normally.
0x10 in ?? ()

(gdb) q