

Debuging

Dr. Mariano Méndez¹

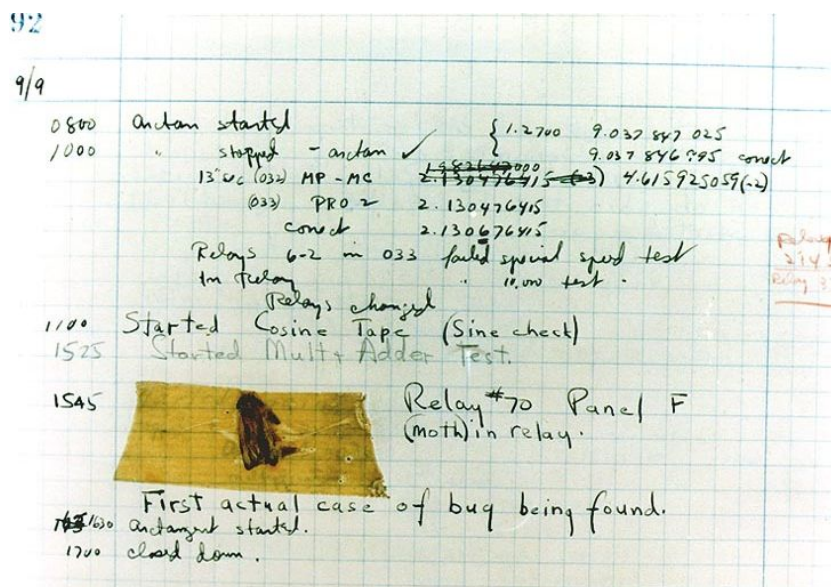
¹Facultad De Ingeniería. Universidad de Buenos Aires

23 de agosto de 2022

1. Introducción

Como dijo Edsger Wybe Dijkstra: "Si la depuración es el proceso de eliminar errores, entonces la programación debe ser el proceso de introducirlos". El termino "BUG" es comúnmente conocido en el ámbito de la Ciencia de la Computación ya que con éste se hace referencia a un error en un programa. Los errores encontrados dentro de un programa se denominan de esta manera desde la vez que el primer error fue encontrado en una computadora. Este término fue forjado por Grace Murray Hopper [?] quien en 1947 encontró un error en una computadora Mark II producido ¡por un bicho! (bug en inglés), ver Figura ??.

Figura 1: ¡El primer Bug!



Un error en el código fuente de un programa de unas 10 o 20 líneas no representa ningún tipo de problema en sí mismo. Se corrige y se continua con la ejecución del mismo. Se está en un real problema cuando la magnitud del código fuente comienza a escalar de unas pocas decenas de código a cientos de miles de líneas de código fuente. Allí encontrar y corregir un error es realmente un tema serio.

Como puede verse en la figura ??, el proceso de compilación, si se lleva a cabo manualmente, puede ser muy complejo. Esta complejidad se basa en que el compilador debe tener toda la información necesaria para poder realizar su trabajo. Cuanto menos información este recibe, en forma parámetros, mas acciones generales realiza. En este apunte se asumirá que por lo menos las siguientes opciones de compilación deberán ser especificadas :

```
$ gcc -g3 -Wall -Wextra -o test1 test1.c
```

Las opciones son:

-g3: se le indica al compilador que mantenga el maximo nivel de informacion de debug. Existen 4 niveles (0...3).
-Wall: este flag habilita los siguientes warnings: -Waddress -Warray-bounds=1 (only with -O2) -Wbool-compare -Wbool-operation -Wc++11-compat -Wc++14-compat -Wchar-subscripts -Wcomment -Wduplicate-decl-specifier -Wenum-compare -Wformat -Wint-in-bool-context -Wimplicit -Wimplicit-int -Wimplicit-function-declaratio -Winit-self -Wlogical-not-parentheses

-Wmain -Wmaybe-uninitialized -Wmemset-elt-size -Wmemset-transposed-args -Wmisleading-indentation -Wmissing-braces -Wnonnull -Wnonnull-compare -Wopenmp-simd -Wparentheses -Wpointer-sign -Wreorder -Wreturn-type -Wsequence-point -Wsizeof-pointer-memaccess -Wstrict-aliasing -Wstrict-overflow=1 -Wswitch -Wtautological-compare -Wtrigraphs -Wuninitialized -Wunknown-pragmas -Wunused-function -Wunused-label -Wunused-value -Wunused-variable -Wvolatile-register-var

Tipos de Errores Existen dos tipos básicos de errores de programación, que se identifican según en que etapa se encuentra un programa. Estas dos etapas son la **etapa de compilación** (pre-procesar,compilar,ensamblar,link-editar) ver Figura ?? y la **etapa de ejecución**, a partir de cuando el programa es cargado en memoria y se ejecuta la primera instrucción del mismo.

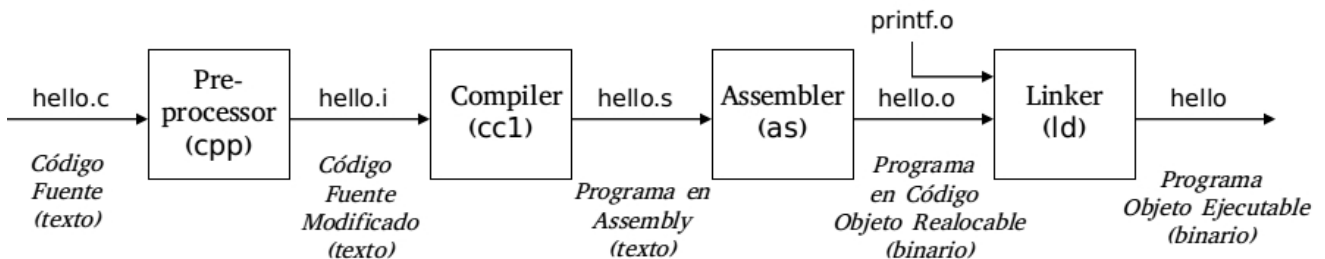


Figura 2: Proceso de Compilación

Según un error se encuentre en una u otra etapa, se denomina **error en tiempo de compilación** o **error en tiempo de ejecución**. Los errores en tiempo de compilación son fácilmente detectables ya que es el compilador quien los detecta y avisa. Por ejemplo, falta cerrar una llave , falta un punto y coma, etc. Habitualmente son errores sintácticos. *Un hecho a tener en cuenta es que durante el proceso de compilación toda referencia humana dentro de un programa es desechada en forma automática, ya que es información innecesaria para la computadora, a menos que se le diga al compilador que no elimine dicha información.*

```

1 #include <stdio.h>
2
3 int main(){
4     printf("Hola mundo!\n");
5     return 0
6 }
7

```

Figura 3: Programa con un Error de Compilación

Por el contrario un error en tiempo de ejecución, aquel que sucede mientras se esta ejecutando el programa, es mucho mas complejo de detectar y por ende de corregir. Dentro de estos tipos de errores existen a su vez dos tipos mas de errores: los **errores de ejecución** y los **errores de lógica**. La diferencia entre ambos tipos de errores de ejecución es que los primeros terminan en un final abrupto del programa, por ejemplo el clásico error de ejecución es el de la división por 0 ver Figura ??.

```

1 #include <stdio.h>
2
3 int main(){
4     int i;
5
6     for (i=0;i<9;i++)
7         printf("%d\n",1/i);
8     return 0
9 }
10

```

Figura 4: Programa con un Error de Ejecución

Mientras que los segundos, no necesariamente terminan con la **ejecución abrupta** pero el resultado que obtienen es **erróneo**. Por ejemplo el error que puede verse en la Figura ?? que es debido a no tener en cuenta la precedencia de los operadores matemáticos del lenguaje .

```
1 float promedio(float a, float b){  
2     return a + b / 2;    /* deberia ser (a + b) / 2 */  
3 }  
4
```

Figura 5: Programa con un Error de Lógica

1.1. Búsqueda Errores en el Código Fuente

Al proceso de encontrar los errores en el código fuente de un programa se lo denomina *debuggear* o *depurar* (en inglés *debugging*). De la misma forma la palabra tiene su origen en el mismo episodio. El principio de la Confirmación:

Arreglar un programa con errores es el proceso por el cual se confirman, una por una, que las cosas que el programador toma como ciertas sobre el código fuente realmente lo son. Cuando se encuentra que uno de esos supuestos no es válido, se ha encontrado una pista del lugar en el cual se encuentra un bug o error

¿Cual es la el valor agregado que le proporciona al principio de Confirmación una herramienta de debugging?. La forma más rudimentaria y menos aconsejable de realizar debugging de programas es mediante la adición de instrucciones centinelas que marcan por donde va pasando el flujo de control del programa. Lamentablemente y aunque esto parezca mentira es una de la formas utilizada por los programadores en el ámbito laboral. En el caso de C la idea es *ir poniendo printf*s de forma tal de ir viendo los valores de ciertas variables o si se llevo a cierto lugar del programa.

Esta forma de realizar debug, lleva a la generación de un ciclo :

1. Agregar un print estratégicamente ubicado en el código fuente
2. Recompilar el programa
3. Analizar la nueva salida de la corrida con las marcas introducidas
4. Eliminar las marcas introducidas (printf) una vez solucionado el error
5. Recompilar todo nuevamente

Este ciclo debe repetirse por cada error encontrado.

Más allá esta decir que esta forma es altamente desaconsejada por varios motivos, alguno de ellos son:

- Agrega código fuente al programa que es inservible
- Agrega comentarios que no aportan nada pues el código debugeado así termina con el código de debugging comentado
- Da la pauta del escaso conocimiento que el programador que realizó esta tarea tenía de herramientas para tales fines
- Desvía la atención del programador

Uno de los hechos mas paradójicos es que esta comprobado que se utiliza mucho mas tiempo para hacer debug que para programar.

1.2. Herramientas

Las herramientas creadas para la búsqueda de errores en programas se denominan "Debuggers". En el caso de C el más utilizado es gdb, GNU Debugger. Cabe destacar que no es el único pero si uno de los más utilizados. Este programa no suele estar incluido en las distribuciones de linux mas comunes y debe instalárselo según corresponda a la distribución que se está manejando. El debugger es un programa nacido en el entorno de consola y por ende no debemos esperar que sea amigable en algún sentido. En la actualidad muchos Entornos Integrados de Desarrollo han incorporado a gdb a su interfaz gráfica haciendo el proceso de debugging más amigable.

2. Gdb

En primer lugar para poder hacer debugging de un programa es necesario que el compilador sepa que debe dejar la mayor cantidad posible de información, que le sirva a un humano para tal fin, como por ejemplo los nombres de las variables, las líneas de código del lenguaje de alto nivel utilizado y otras varias cosas. Gdb fue creado por Richard Stallman en el año 1986 como parte del proyecto GNU [?]. En el manual del usuario de gdb Stallman describe cuál es el propósito del programa [?]:

El propósito de un depurador como gdb es permitirte que veas que está pasando “adentro” de otro programa mientras éste se está ejecutando o que está haciendo otro programa en el momento que este falló. Gdb puede hacer cuatro tipos de cosas (más otras como soporte de aquellas) para ayudarte a encontrar bugs en el momento:

1. Iniciar tu programa, especificando cualquier cosa que sea necesaria para afectar su comportamiento.
2. Hacer que tu programa se detenga sobre ciertas condiciones específicas.
3. Examinar qué está ocurriendo, cuando el programa se detuvo.
4. Cambiar cosas en tu programa ...

Como se dijo anteriormente (??), en el proceso de compilación toda referencia a conceptos humanos son eliminados durante el mismo, esto es nombres, comentarios, código en el lenguaje de programación, etc. solo queda el código maquina necesario. Para poder realizar una depuración o debug se debe especificar al compilador que no elimine dicha información, para ello se debe utilizar una opción que el en compilador gcc es -g.

```
$ gcc -Wall -g ejemplo.c -o ejemplo.exe
```

Una vez hecho esto el compilador creará una versión ejecutable del programa que posee información para debug. Una vez que se tiene compilado el programa sin errores de compilación y la versión ejecutable del mismo se ha generado correctamente se proceda a iniciar el proceso de debugging.

```
$ gdb ejemplo.exe
```

Una vez que se haya ejecutado el comando se vera en la pantalla de la consola:

```
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 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".
Type "show configuration" for configuration details.
Para las instrucciones de informe de errores, vea:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Leyendo símbolos desde ejemplo.exe...hecho.
(gdb)
```

Con lo cual se sabrá que la ejecución del proceso de depuración o debugging ha comenzado.

2.1. Comandos Básicos

En esta Sección se describirán algunos de los comandos más utilizados del debugger.

2.1.1. help

Este comando muestra el menú de ayuda de programa. Se ejecuta (gdb) help y despliega la siguiente pantalla:

```
List of classes of commands:
```

```
aliases -- Alias de otras órdenes
breakpoints -- Para el programa en ciertos puntos
data -- Examinando datos
```

files -- Especificando y examinando archivos
 internals -- Órdenes de mantenimiento
 obscure -- Ocultar características
 running -- Corriendo el programa
 stack -- Examinar la pila
 status -- Preguntas de estado
 support -- Facilidades de soporte
 tracepoints -- Tracing of program execution without stopping the program
 user-defined -- Órdenes definidas por el usuario

Type "help" followed by a class name for a list of commands in that class.
 Type "help all" for the list of all commands.
 Type "help" followed by command name for full documentation.
 Type "apropos word" to search for commands related to "word".
 Command name abbreviations are allowed if unambiguous.

Para ejemplificar los comandos de linea de gdb se utilizara un programa C escrito en dos archivos (*main.c*, *swapper.c*).

```

1 #include<stdio.h>
2 void swap( int * a, int * b);
3
4 int main(void){
5     int i=1;
6     int j=2;
7
8     printf("i: %d, j: %d\n",i,j);
9     swap(&i,&j);
10    printf("i: %d, j: %d\n",i,j);
11
12    return 0;
13 }
14
```

main.c

```

1 void swap( int * a, int * b){
2     int c = *a;
3     *a = *b;
4     *b = c;
5 }
6
```

swapper.c

Comandos de compilación:

```

$ gcc -g3 -Wall -Wextra -c main.c swapper.c
$ gcc -o swap main.o swapper.o
$ gdb swap
```

2.1.2. **list**

Este comando permite listar (hasta 10 líneas) del código fuente del programa que está siendo depurado por gdb, puede cambiarse con `set listsize <x cantidad de líneas >`, su ejecución se realiza como se muestra a continuación, que muestra diez líneas de código:

(gdb) list

Para mostrar un rango de líneas se lo hace separando el rango entre comas. Este comando tiene muchas variantes, por ejemplo:

(gdb) list nombre_funcion

Lista el código fuente de la función especificada.

```

(gdb) list swap
1 void swap( int * a, int * b){
2     int c = *a;
```

```

3      *a = *b;
4      *b = c;
5  }
(gdb)

```

2.1.3. quit

Este comando permite la salida del ambiente de depuración y del debugger, su ejecución se realiza como se muestra a continuación:

```
(gdb) quit
```

2.2. Comandos de Ejecución

2.2.1. run

Este comando ejecuta el program como si se lo lanzara desde la consola. Para ello se tipea en la linea de comando de gdb:

```
(gdb) run
```

Este comando indicará al debugger o depurador que debe iniciar la ejecución del programa.

2.2.2. start

Este comando inicia la ejecución del programa paso a paso desde la primera linea del del mismo.

```
(gdb) start
```

2.2.3. next

Este comando permite ejecutar la siguiente linea que corresponde en la ejecución del programa. Esto puede verse en la Figura ??.

```
(gdb) next
```

```

int main(void)
{
    int i = 4;
    int j = 6;

    printf("i: %d, j: %d\n", i, j);
    swap(i, j);
    printf("i: %d, j: %d\n", i, j);
    return 0;
}

void swap(int a, int b)
{
    int c = a;
    a = b;
    b = c;
}

```

Figura 6: gdb: instrucción ejecución next

Cabe destacar que en las funciones de la biblioteca estándar de C el comando a utilizar es next, de lo contrario se comenzará a depurar dentro de estas funciones.

2.2.4. step

Este comando permite ejecutar la próxima instrucción. Se debe tener en cuenta que si esta instrucción es una función la ejecución sigue dentro de la misma. Esto puede verse en la Figura ??.

```
(gdb) step
```

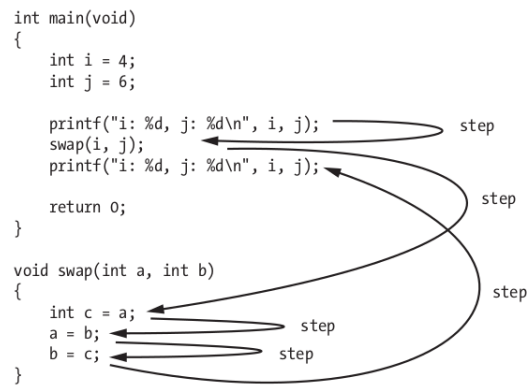


Figura 7: gdb: instrucción de ejecución step

Si la variable `step-mode` está seteada en `on` gdb se detendrá en cada instrucción de funciones que no tengan información de debug, como por ejemplo `printf` de la `libc`:

```

Debugging - Administrador de archivos /home/darthmendez/Dropbox/Academico/UBA/7540-Algoritmos... Debugging.pdf gdb /home/darthmendez/Dropbox/Academico/UBA/7540-Algor...
Archivo Editar Ver Buscar Terminal Ayuda
Starting program: /home/darthmendez/Dropbox/Academico/UBA/7540-Algoritmos Y Programacion I 2016/material/apuntes_teoricos/12-Debugging/ejemplos_debug/
ejemplo3/a.out
Temporary breakpoint 2, main () at main.c:4
4      printf(" hola mundo\n");
(gdb) s
0x00007ffff7a90130 in puts () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7a90132 in puts () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7a90134 in puts () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7a90137 in puts () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7a90138 in puts () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7a90139 in puts () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7a9013d in puts () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7ab6e10 in __strlen_sse2 () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7ab6e14 in __strlen_sse2 () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7ab6e18 in __strlen_sse2 () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7ab6e1c in __strlen_sse2 () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7ab6e20 in __strlen_sse2 () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7ab6e23 in __strlen_sse2 () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7ab6e26 in __strlen_sse2 () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7ab6e2d in __strlen_sse2 () from /usr/lib/libc.so.6
(gdb) s
0x00007ffff7ab6e34 in __strlen_sse2 () from /usr/lib/libc.so.6
(gdb) s

```

Figura 8: gdb: instrucción ejecución next

Para que no suceda eso se debe deshabilitar la opción mediante la ejecución de `set step-mode off`

2.2.5. `finish`

Esta instrucción permite finalizar la ejecución de una determinada función cuando se está dentro de la misma.

(gdb) `finish`

3. Comandos un Poco más Avanzados: Parando y Continuando

Una de las ideas principales del aporte de gdb es la posibilidad de congelar la ejecución del programa en cualquier punto de la misma y poder observar el estado del mismo, por ejemplo, cuales son los valores de determinadas variables, que porción de código está ejecutando y en qué secuencia. Para ello ha una serie de comandos que proporciona gdb que se describen a continuación. Un poco de definiciones:

- **Breakpoint:** "Un breakpoint hace que el programa se detenga si un determinado punto del mismo es alcanzado" [?]. Esto significa que un breakpoint permite detener la ejecución del programa si el mismo llega a ejecutar la línea a la que se le asigna un breakpoint.

- **Watchpoint:** Un watchpoint es otra posibilidad para detener la ejecución de un programa pero en este caso se realiza cuando una determinada expresión cambia [?]. Cuando se hace referencia a una expresión es considerada una o mas variables combinadas por un operador (i.e: i++, x+y,etc)
- **Catchpoint:** "Un catchpoint es otro tipo de breakpoint que detiene al programa cuando un cierto tipo de evento ocurre" [?].

3.1. break

Este comando permite crear breakpoints dentro de un programa a ser debugado. Permite agregar un breakpoint en una línea, el nombre de una función o en una dirección de memoria donde hay una instrucción.

```
(gdb) break <lugar>
```

El <lugar> puede ser por ejemplo el nombre de una función:

```
(gdb) break main
```

```
Breakpoint 1 at 0x6c2: file main.c, line 4.
```

```
(gdb)
```

Para crear un breakpoint en una determinada línea de un determinado archivo debe especificarse de la siguiente forma nombre_archivo:línea

```
(gdb) break main:8
```

```
Note: breakpoint 1 also set at pc 0x6c2.
```

```
Breakpoint 2 at 0x6c2: file main.c, line 4.
```

```
(gdb)
```

Cuando se lo utiliza sin ninguna referencia el comando break agrega un breakpoint en la próxima instrucción a ser ejecutada

```
(gdb) break
```

3.1.1. Breakpoints Condicionales

Gdb permite crear breakpoints si cierta condición se cumple o no; este tipo de breakpoint se llama *breakpoint condicional*. La siguiente instrucción genera un breakpoint condicional si la variable i llega a tomar en algún momento el valor 1:

```
(gdb) break main if i==1
```

```
Breakpoint 1 at 0x6c2: file main.c, line 4.
```

para verlo basta con:

```
(gdb) info break
```

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x000000000000006c2	in main at main.c:4

```
stop only if i==1
```

```
(gdb)
```

Otro ejemplo interesante puede verse en el siguiente código:

```
1 for (cont=0; cont < 7500; cont++){
2     valor= calcular_valor(cont);
3     hacer_algo(valor);
4 }
```

El código funciona bien pero empieza a fallar a llegar al valor 7000, lo mas común es querer para el programa cuando la variable cont llegue a 6999:

```
break if (i == 6999)
```

Lo mas interesante es que la condición del breakpoint puede ser cualquier condición valida en C, considerando verdadero (valor no zero) y falso (cero)

- **Operadores de igualdad, desigualdad o lógicos** (i, j, ==, !=, <, >, &&, —, etc.): break 180 if string==NULL &&
- **Operadores de corrimiento o Bitwise** (&, —, <<, >>, etc.): break test.c:34 if (x & y) == 1
- **Operadores aritmeticos** (+, -, x, /, %): break myfunc if i % (j + 3) != 0
- **Funciones propias, siempre y cuando estén linkeditadas en el programa:** break test.c:myfunc if ! es_primo(i)
- **Funciones de biblioteca, siempre y cuando estén linkeditadas en el codigo fuente:** break 44 if strlen(mystring) == 0

3.2. Watch

Permite agregar watchpoints. Recordar que un watchpoint es una variable o mas relacionadas con un operador. El watchpoint más simple se muestra a continuación

```
(gdb) watch nombreDeUnaVariable
```

Todos los breakpoints y watchpoints pueden ser eliminados ejecutando:

```
(gdb) clear
```

O si se requiere eliminar un determinado breakpoint se puede hacerlo así:

```
(gdb) clear funcion
```

Por último para consultar sobre los breakpoints se ejecuta

```
(gdb) info break
```

```
(gdb) info watch
```

Cuando se ejecuta el comando info break se listan todos los break y watchpoints seteados:

```
(gdb) info breakpoint
Num      Type      Disp Enb Address          What
1        breakpoint keep y 0x00000000000006c2 in main at main.c:4
2        breakpoint keep y 0x00000000000006c2 in main at main.c:4
3        breakpoint keep y 0x000000000000074b in swap at swapper.c:2
4        breakpoint keep y 0x000000000000074b in swap at swapper.c:2
(gdb)
```

Identifier (Num): El identificador unico del breakpoint

Type (Type): Este campo indica si el item es un breakpoint, watchpoint o catchpoint.

Disposition (Disp): cada breakpoint tiene una disposicion que indica que le sucedera la proxima vez que este cause a gdb que pausee la ejecucion del programa. existen tres posibles

keep: el breakpoint permanece sin cambios hasta la próxima vez que sea alcanzado. es por defecto.

del: el breakpoint sera eliminado la próxima vez que este sea alcanzado.

dis: el breakpoint seraa deshabilitado la proxima vez que sea alcanzado.

Enable Status (Enb): indica si el breakpoint esta habilitado o deshabilitado.

Address (Address): es el lugar de la memoria en el cual esta seteado el breakpoint

Location (What): cada breakpoint esta alojado en algún lugar del codigo fuente , este campo indica ese lugar.

3.2.1. Inspeccionando valores de las variables

Una de las cosas mas importantes en el proceso de debbuging o depuración es saber que valor tiene una o un grupo de variables en un determinado momento de la ejecución. Para ello se utiliza básicamente el comando `print`, a continuación se utilizara un fragmento de código fuente escrito en lenguaje C:

```
1 #include <stdio.h>
2
3 int main(void){
4     int i=1;
5     int j=2;
6     int vector[15];
7
8     printf("i: %d, j: %d\n",i,j);
9     printf("i: %d, j: %d\n",i,j);
10    for (i=0; i<10;i++)
11        vector[i]=0;
12
13    return 0;
14 }
15
```

main.c

Se procede a compilar el programa y a llamarlo usando gdb:

```
(darthmendez@darthMendez ~> gdb main.exe
GNU gdb (GDB) 8.0.1
Copyright (C) 2017 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-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from main.exe...done.
(gdb) start
Temporary breakpoint 1 at 0x6c2: file main.c, line 3.
Starting program: /home/main.exe
```

```
Temporary breakpoint 1, main () at main.c:3
3 int main(void){
(gdb) print i
$1 = -7514
(gdb)
```

Si se quisiera ver el contenido del vector se puede ejecutar el siguiente comando :

```
(gdb) print vector
$2 = {1, 0, -139716939, 32767, 1, 0, 1431652253, 21845, 0, 0, 0, 0, 1431652176, 21845, 1431651760}
(gdb)
```

Como puede verse el contenido del vector apenas iniciado el programa es basura, al igual que el de las variables.

Si se desea que una lista de variables o expresiones sea visualizada cada vez que se ejecuta un comando se utiliza para ello *display*:

```
(gdb) display i
1: i = -7514
(gdb) display j
2: j = 32767
(gdb) display
1: i = -7514
2: j = 32767
(gdb) display vector
3: vector = {1, 0, -139716939, 32767, 1, 0, 1431652253, 21845, 0, 0, 0, 0, 1431652176, 21845, 1431651760}
(gdb) display
1: i = -7514
2: j = 32767
3: vector = {1, 0, -139716939, 32767, 1, 0, 1431652253, 21845, 0, 0, 0, 0, 1431652176, 21845, 1431651760}
(gdb)
```

Algunos comandos útiles para visualización:

variables locales para **visualizar únicamente las variables locales** se ejecuta el comando *info locals*

```
(gdb) info locals
```

funciones para **visualizar las funciones de un programa** se ejecuta el comando *info functions*

```
(gdb) info functions regexp
```

tipos de datos para **saber el tipo de dato en el cual se definió una variable** se ejecuta el comando **ptype**

```
(gdb) ptype vector
type = int [15]
```

tipo de una expresión el comando **whatis** devuelve el tipo de una expresión tras ser evaluada.

```
(gdb) whatis i=1
type = int
```

3.2.2. **bt**

Este comando **imprime el stack de las llamadas a las funciones y muestra cual es la que se está ejecutando actualmente.**

```
(gdb) bt
terceraFuncion <= current location
otraFuncion
unaFuncion
main
```

Estos son los comandos mínimos que son necesario saber para poder realizar una depuración de un programa. A continuación se realizará un ejemplo.

4. Un Ejemplo Sencillo

El ejemplo seleccionado es :

```
1 #include <stdio.h>
2
3 int main(){
4     int numero;
5     int contador=1;
6     int suma;
7
8     printf("Ingrese el valor de un numero entero\n");
9     scanf("%d",&numero);
10    while(contador<=numero){
11        suma=suma+contador;
12        contador++;
13    }
14    printf("El valor de la suma es:%d\n",suma);
15    return 0;
16 }
```

Se compila utilizando las siguientes opciones, donde **—g—** indica dejar la información para la depuración y **—Wall—** pone al compilador en la opción pedante:

```
$ gcc -Wall -g ejemplo.c -o ejemplo.exe
```

Un ejercicio interesante para ver es el de hacer la prueba de compilar el mismo programa con y sin la opción de la información de depuración:

```
$ gcc -Wall ejemplo.c -o ejemplo
$ gcc -Wall -g ejemplo.c -o ejemplo.exe
```

luego se ejecuta el comando **ls** con la opción **-l** de la siguiente forma:

```
$ ls -l ejemplo*
-rwxrwxr-x 1 mariano mariano 8776 sep 18 18:06 ejemplo
-rw-rw-r-- 1 mariano mariano 271 sep 16 13:11 ejemplo.c
-rwxrwxr-x 1 mariano mariano 9960 sep 18 18:07 ejemplo.exe
```

puede verse que **ejemplo.exe**, con lo cual se puede validar que la información de depuración ocupa espacio en el archivo ejecutable.

Se ejecuta el programa y el resultado que arrojó la ejecución no está por lejos al valor esperado, por ende se inicia una depuración:

```
$ gdb ejemplo.exe
```

Como el programa tiene solo 16 líneas las listamos de la siguiente forma:

```
(gdb) list 1,16
1 #include<stdio.h>
2
3 int main(){
4 int numero;
5 int contador=1;
6 int suma;
7
8 printf("Ingrese el valor de un número entero\n");
9 scanf("%d",&numero);
10 while(contador<=numero){
11 suma=suma+contador;
12 contador++;
13 }
14 printf("El valor de la suma es:%d\n",suma);
15 return 0;
16 }(gdb)
```

Se ejecuta el programa desde el debugger:

```
Starting program: /home/mariano/Dropbox/Academico/UBA/7540-Algoritmos Y Programación I 2016/teoria/12-D
Ingrese el valor de un número entero
```

```
3
```

```
El valor de la suma es:32773
```

```
[Inferior 1 (process 26928) exited normally]
```

```
(gdb) run
```

```
Starting program: /home/mariano/Dropbox/Academico/UBA/7540-Algoritmos Y Programación I 2016/teoria/12-D
Ingrese el valor de un número entero
```

```
7
```

```
El valor de la suma es:32795
```

```
[Inferior 1 (process 26929) exited normally]
```

```
(gdb)
```

```
(gdb) run
```

```
Starting program: /home/mariano/Dropbox/Academico/UBA/7540-Algoritmos Y Programación I 2016/teoria/12-D
Ingrese el valor de un número entero
```

```
3
```

```
El valor de la suma es:32773
```

```
[Inferior 1 (process 26928) exited normally]
```

El resultado obtenido de la suma de los 3 primeros números es 32773. Este valor dista por lejos de lo que debería ser, un 6. Volvemos a hacer otra prueba, utilizando el comando `—run—` y obtenemos otro valor no esperado

```
(gdb) run
```

```
Starting program: /home/ejemplo.exe
Ingrese el valor de un número entero
```

```
7
```

```
El valor de la suma es:32795
```

```
[Inferior 1 (process 26929) exited normally]
```

```
(gdb)
```

Debido a que los valores no están dentro de lo esperado vamos a iniciar una depuración paso a paso, en el cual se crea un breakpoint inicial en la función main, línea 3:

```
(gdb) start
```

```
Punto de interrupción temporal 1 at 0x40064e: file ejemplo.c, line 3.
```

```
Starting program: /home/ejemplo.exe
```

```
Temporary breakpoint 1, main () at ejemplo.c:3
```

```
3 int main(){
```

Se agregarán dos watchpoints

```
(gdb) watch contador
Hardware watchpoint 2: contador
(gdb) watch suma
Hardware watchpoint 3: suma
(gdb) info watch
```

Num	Type	Disp	Enb	Address	What
2	hw watchpoint	keep	y		contador
3	hw watchpoint	keep	y		suma

Se inicia a ejecutar el programa paso a paso, hay que tener en cuenta que la línea que está mostrando gdb es la línea que se ejecutará si se vuelve a ingresar el comando `—next—`, en este caso es la línea 5:

```
(gdb) next
5 int contador=1;
```

Una vez que se ejecute la línea 5 ingresando un nuevo comando `next`, el valor de la variable `contador` debería cambiar, y dado que se ha definido un watchpoint se debería mostrar dicho cambio:

```
(gdb) next
```

```
Hardware watchpoint 2: contador
```

```
Old value = -9072
New value = 1
main () at ejemplo.c:8
8 printf("Ingrese el valor de un número entero\n");
```

Se continua ejecutando paso a paso:

```
(gdb) next
Ingrese el valor de un número entero
9 scanf("%d",&numero);
(gdb) next
3
10 while(contador<=numero){
(gdb) next
11 suma=suma+contador;
```

Y en la próxima acción de programa a ejecutar el valor de `suma` debería cambiar

```
(gdb) next
```

```
Hardware watchpoint 3: suma
```

```
Old value = 32767
New value = 32768
main () at ejemplo.c:12
12 contador++;
```

Ya aquí se puede decir que el bug ha sido encontrado...La variable `suma` no ha sido inicializada con ningún valor, por ende tiene el contenido que las celdas de memoria que dicha variable ocupa tenían al iniciarse la computadora ese valor es `—Old value = 32767—`. De esta forma la suma que se está realizando parte de cualquier valor y no del neutro para la suma que es 0. De todas formas y como ejemplo se prosigue con la depuración:

```
(gdb) next
```

```
Hardware watchpoint 2: contador
```

```
Old value = 1
New value = 2
main () at ejemplo.c:10
10 while(contador<=numero){
(gdb) next
11 suma=suma+contador;
(gdb) next
```

Hardware watchpoint 3: suma

```
Old value = 32768
New value = 32770
main () at ejemplo.c:12
12 contador++;
(gdb) next
```

Hardware watchpoint 2: contador

```
Old value = 2
New value = 3
main () at ejemplo.c:10
10 while(contador<=numero){
(gdb) next
11 suma=suma+contador;
(gdb) next
```

Hardware watchpoint 3: suma

```
Old value = 32770
New value = 32773
main () at ejemplo.c:12
12 contador++;
(gdb) next
```

Hardware watchpoint 2: contador

```
Old value = 3
New value = 4
main () at ejemplo.c:10
10 while(contador<=numero){
(gdb) next
14 printf("El valor de la suma es:%d\n",suma);
(gdb) next
El valor de la suma es:32773
15 return 0;
(gdb) next
16 }(gdb)
```

Watchpoint 2 deleted because the program has left the block in which its expression is valid.

```
(gdb) next
[Inferior 1 (process 26895) exited normally]
(gdb)
```

Finalmente el programa termina exitosamente. Por ello el cambio que debe realizarse es la inicialización de la variable suma.

```
1 #include<stdio.h>
2 int main(){
3     int numero;
4     int contador=1;
5     int suma=0;
6
7     printf("Ingrese el valor de un numero entero\n");
8     scanf("%d",&numero);
9     while(contador<=numero){
10         suma=suma+contador;
11         contador++;
12     }
13     printf("El valor de la suma es:%d\n",suma);
14     return 0;
15 }
```

Se vuelve a compilar con la corrección del bug y se vuelve a ejecutar:

```
$gcc -Wall -g ejemplo.c -o ejemploBien.exe
$./ejemploBien.exe
Ingrese el valor de un número entero
3
El valor de la suma es:6
```

Finalmente se obtiene el resultado deseado. Con lo cual no quiere decir que el programa está libre de errores sino que para el valor de la variable numero=3, el resultado es correcto. Para mejorar la detección de errores se utilizará una técnica llamada Testing.

Nota del autor: parte de este apunte, que solo pretende ser una ayuda memoria para la utilización de gdb esta basado en el libro [?] **The art of debugging with GDB, DDD, and Eclipse** de Matloff, Norman S y Salzman, Peter Jay el cual recomendamos como bibliografía para profundizar algunos temas.

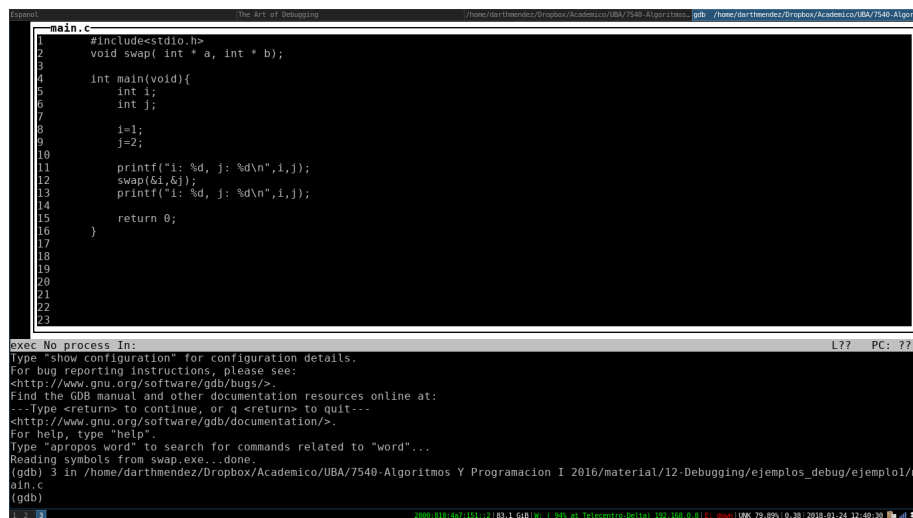
5. Debug con Gdb TUI

Si bien parecería ser que no existe una alternativa gráfica para depurar, **gdb trae un entorno mas amigable llamado TUI (Text User Interface), que viene por defecto con gdb.** Para activar dicho entorno **se activa presionando <ctrl>+ X + A** (en ese orden preciso orden) desde dentro de gdb. O directamente **gdb programa.exe tui**. Comandos importantes a tener en cuenta en modo TUI.

5.1. Comando: layout

Este comando **permite setear el diseño de las ventanas.** En cada diseño la ventana de comandos siempre se visualiza, ademas de incorporar otra. **La sintaxis es layout + opción:**

rc: **Muestra la ventana del código fuente y la ventana de comandos.**



```

main.c
1  #include<stdio.h>
2  void swap( int * a, int * b);
3
4  int main(void){
5      int i;
6      int j;
7
8      i=1;
9      j=2;
10
11     printf("i: %d, j: %d\n",i,j);
12     swap(&i,&j);
13     printf("i: %d, j: %d\n",i,j);
14
15     return 0;
16 }
17
18
19
20
21
22
23

exec: No process in:
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
--Type <return> to continue, or q <return> to quit--
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from swap.exe...done.
(gdb) 3 in /home/darthmendez/Dropbox/Academico/UBA/7540-Algoritmos Y Programacion I 2016/material/12-Debugging/ejemplos_debug/ejemplo1/main.c
(gdb)
  
```

asm: **Muestra la ventana del código assembler y la ventana de comandos.**

```

0x555555546d1 <main+23>    movl    $0x1,-0x10(%rbp)
0x555555546d8 <main+30>    movl    $0x2,-0xc(%rbp)
0x555555546df <main+37>    mov     -0xc(%rbp),%edx
0x555555546e2 <main+40>    mov     -0x10(%rbp),%eax
0x555555546e5 <main+43>    mov     %eax,%esi
0x555555546e7 <main+45>    lea     0x106(%rip),%rdi    # 0x555555547f4
0x555555546ee <main+52>    mov     $0x0,%eax
0x555555546f3 <main+57>    callq   0x555555545a0 <printf@plt>
0x555555546f8 <main+62>    lea     -0xc(%rbp),%rdx
0x555555546fc <main+66>    lea     -0x10(%rbp),%rax
0x55555554700 <main+70>    mov     %rdx,%rsi
0x55555554703 <main+73>    mov     %rax,%rdi
0x55555554706 <main+76>    callq   0x5555555473f <swap>
0x5555555470b <main+81>    mov     -0xc(%rbp),%edx
0x5555555470e <main+84>    mov     -0x10(%rbp),%eax
0x55555554711 <main+87>    mov     %eax,%esi
0x55555554713 <main+89>    lea     0xda(%rip),%rdi    # 0x555555547f4
0x5555555471a <main+96>    mov     $0x0,%eax
0x5555555471f <main+101>   callq   0x555555545a0 <printf@plt>
0x55555554724 <main+106>   mov     $0x0,%eax
0x55555554729 <main+111>   mov     -0x8(%rbp),%rcx
0x5555555472d <main+115>   xor     %fs,%rcx
0x55555554736 <main+124>   je      0x5555555473d <main+131>

native process 24470 In: main                                L8      PC: 0x555555546d1
(gdb) layout src
(gdb) layout split
(gdb) focus asm
Focus set to asm window.
(gdb) focus src
Focus set to src window.
(gdb) layout src
Undefined command: "layuot". Try "help".
(gdb) layout src
(gdb) layout asm
(gdb) layout asm

```

split: Muestra la ventana del código fuente, del código assembly y la ventana de comandos.

```

main.c
8      i=1;
9      j=2;
10
11     printf("i: %d, j: %d\n",i,j);
12     swap(&i,&j);
13     printf("i: %d, j: %d\n",i,j);
14
15     return 0;
16 }
17
18

0x555555546d1 <main+23>    movl    $0x1,-0x10(%rbp)
0x555555546d8 <main+30>    movl    $0x2,-0xc(%rbp)
0x555555546df <main+37>    mov     -0xc(%rbp),%edx
0x555555546e2 <main+40>    mov     -0x10(%rbp),%eax
0x555555546e5 <main+43>    mov     %eax,%esi
0x555555546e7 <main+45>    lea     0x106(%rip),%rdi    # 0x555555547f4
0x555555546ee <main+52>    mov     $0x0,%eax
0x555555546f3 <main+57>    callq   0x555555545a0 <printf@plt>
0x555555546f8 <main+62>    lea     -0xc(%rbp),%rdx
0x555555546fc <main+66>    lea     -0x10(%rbp),%rax
0x55555554700 <main+70>    mov     %rdx,%rsi

native process 24470 In: main                                L8      PC: 0x555555546d1
(gdb) layout src
(gdb) layout split
(gdb) layout split
(gdb) focus asm
Focus set to asm window.
(gdb) focus src
Focus set to src window.
(gdb) layout src
Undefined command: "layuot". Try "help".
(gdb) layout src
(gdb) layout asm
(gdb) layout split
(gdb)

```

reg: Si se está en el diseño de src, esta opción muestra además los registros y la ventana de comando. Si se está en el diseño asm o split, esta opción muestra además los registros y la ventana de comando.

next: Muestra el próximo diseño de pantallas.

prev: Muestra el diseño previo de pantallas.

5.2. Comando: focus

Este comando permite desplazarse entre las ventanas de TUI para darle foco en la que se desea trabajar. Se utiliza siguiendo los siguientes parámetros:

src: permite hacer scrolling en la ventana del código fuente.

asm: permite hacer scrolling en la ventana del código assembly.

regs: permite hacer scrolling en la ventana de registros.

cmd: permite hacer scrolling en la ventana de comandos.

prev: permite hacer scrolling en la ventana precedente.

next: permite hacer scrolling en la ventana siguiente.

5.3. Comando: tui reg

Comando encargado de **cambiar los registros que son mostrados en la ventana de registros**. Si la ventana de registros no esta siendo mostrada este comando hace que se muestre, ademas se le pueden agregar las siguientes opciones:

general: se muestran los registros de proposito general de la cpu.

```

Register group: general
rax      0x0      0
rcx      0x0      0
rsi      0x7fffffe3b8 140737488348088
rbp      0x7fffffe2d0 0x7fffffe2d0
r8       0x55555554e0 93824992233440
r10      0x2       2
r12      0x555555545b0 93824992232880
r14      0x0      0
rip      0x555555546d1 0x555555546d1 <main+23>
cs       0x33     51
fs       0x0      0
rbx      0x0      0
rdx      0x7fffffe3c8 140737488348104
rdi      0x1      1
rsp      0x7fffffe2c0 0x7fffffe2c0
r9       0x7ffff7de79b0 140737351940528
r11      0x3      3
r13      0x7fffffe3b0 140737488348080
r15      0x0      0
eflags   0x246     [ PF ZF IF ]
ss       0x2b     43
es       0x0      0

8      i=1;
9      j=2;
10
11      printf("i: %d, j: %d\n",i,j);
12      swap(&i,&j);
13      printf("i: %d, j: %d\n",i,j);
14
15      return 0;
16
17
18

Inferior 1 [process 24470] will be killed.
Quit anyway? (y or n) n
Not confirmed.
(gdb) focus reg
Focus set to regs window.
(gdb) layout src
(gdb) tui reg all
(gdb) reg general
Undefined command: "reg". Try "help".
(gdb) tui reg general
(gdb)

```

float: se muestran los registros de punto flotante de la cpu.

```

Register group: float
st0      0 (raw 0x0000000000000000)
st2      0 (raw 0x0000000000000000)
st4      0 (raw 0x0000000000000000)
st6      0 (raw 0x0000000000000000)
fctrl    0x37f    895
ftag     0xffff   65535
fioff    0x0      0
fofff    0x0      0
st1      0 (raw 0x0000000000000000)
st3      0 (raw 0x0000000000000000)
st5      0 (raw 0x0000000000000000)
st7      0 (raw 0x0000000000000000)
fstat    0x0      0
fiseg    0x0      0
foseg    0x0      0
fop      0x0      0

8      i=1;
9      j=2;
10
11      printf("i: %d, j: %d\n",i,j);
12      swap(&i,&j);
13      printf("i: %d, j: %d\n",i,j);
14
15      return 0;
16
17
18

Quit anyway? (y or n) n
Not confirmed.
(gdb) focus reg
Focus set to regs window.
(gdb) layout src
(gdb) tui reg all
(gdb) reg general
Undefined command: "reg". Try "help".
(gdb) tui reg float
(gdb)

```

system: se muestran los registros del sistema.

```

Register group: system
fs_base 0x7ffffcd4c0 140737353929920 gs_base 0x0 0
orig_rax 0xffffffffffffffff -1

main.c
6 int i;
7 int j;
8 i=1;
9 j=2;
10 printf("i: %d, j: %d\n",i,j);
11 swap(&i,&j);
12 printf("i: %d, j: %d\n",i,j);
13 return 0;

Not confirmed.
(gdb) focus reg
Focus set to regs window.
(gdb) layout src
(gdb) tui reg all
(gdb) reg general
Undefined command: "reg". Try "help".
(gdb) tui reg general
(gdb) tui reg float
(gdb) tui reg vector
(gdb) tui reg system
(gdb)

```

vector: se muestran los registros de vector.

```

Register group: vector
xmm0 {v8_float = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_double = {0x0, 0x0, 0x0, 0x0}, v32_int8 = {0x0 <repeats 32 times>}}
ymm1 {v8_float = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_double = {0x0, 0x0, 0x0, 0x0}, v32_int8 = {0x2f <repeats 16 times>}}
ymm2 {v8_float = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_double = {0x0, 0x0, 0x0, 0x0}, v32_int8 = {0x0 <repeats 32 times>}}
ymm3 {v8_float = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_double = {0x0, 0x0, 0x0, 0x0}, v32_int8 = {0x0, 0xff, 0x0 <repeats 30 times>}}
ymm4 {v8_float = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_double = {0x0, 0x0, 0x0, 0x0}, v32_int8 = {0x0 <repeats 32 times>}}
ymm5 {v8_float = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_double = {0x0, 0x0, 0x0, 0x0}, v32_int8 = {0x0 <repeats 13 times>}}
ymm6 {v8_float = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_double = {0x0, 0x0, 0x0, 0x0}, v32_int8 = {0x0 <repeats 32 times>}}
ymm7 {v8_float = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_double = {0x0, 0x0, 0x0, 0x0}, v32_int8 = {0x0 <repeats 32 times>}}
ymm8 {v8_float = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_double = {0x0, 0x0, 0x0, 0x0}, v32_int8 = {0x0 <repeats 32 times>}}
ymm9 {v8_float = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, v4_double = {0x0, 0x0, 0x0, 0x0}, v32_int8 = {0x0 <repeats 32 times>}}

8 i=1;
9 j=2;
10
11 printf("i: %d, j: %d\n",i,j);
12 swap(&i,&j);
13 printf("i: %d, j: %d\n",i,j);
14
15 return 0;
16 }
17
18

Quit anyway? (y or n) n
Not confirmed.
(gdb) focus reg
Focus set to regs window.
(gdb) layout src
(gdb) tui reg all
(gdb) reg general
Undefined command: "reg". Try "help".
(gdb) tui reg general
(gdb) tui reg float
(gdb) tui reg vector
(gdb)

```

all: muestra todos los registros de la cpu.

next: permite hacer scrolling en la ventana siguiente.

Nota: para ampliar el tamaño de las ventanas se debe utilizar el comando **winheight**, adicionando el nombre de la ventana (src, cmd,asm y regs) y un numero entero que indique el tamaño de ampliación de la ventana ; si este es positivo crece el tamaño de la ventana y si es negativo decrece.

```

winheight src +10
winheight name -10

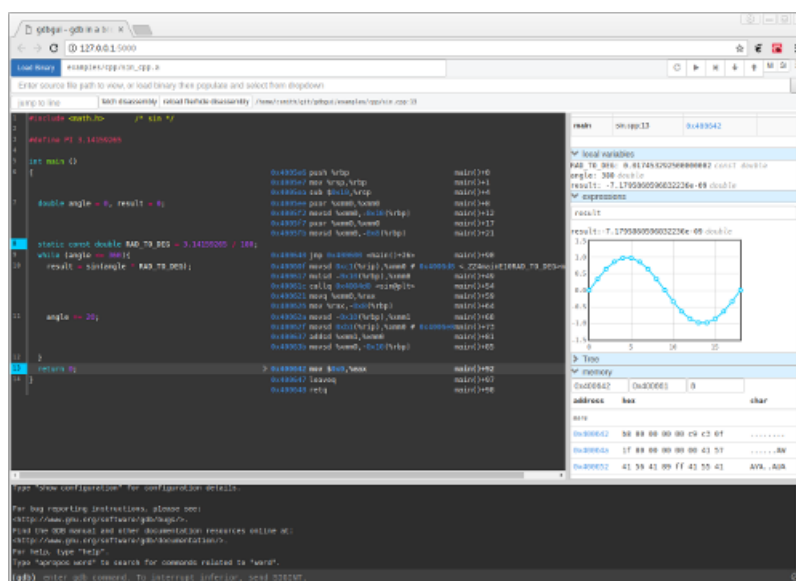
```

6. Debug con un Front-End de Gdb para un Navegador: gdbgui

Gdbgui es un front-end realizado para **debugear o depurar programas utilizando gdb pero a través de un navegador**. El mismo puede ser descargado de <https://gdbgui.com/>. ¿Cuales son las implicancias de usar esta herramienta? Se pasa de esto :

```
>>> gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
>>>
```

a esto:



Esta aplicación web es un cascaron que embellece gráficamente a gdb. Ojo! no es otro debugger sino que **es una mascara gráfica**, por decirlo de algún modo para gdb. Para poder ser instalado se deben cumplir algunos requisitos:

Sistema Operativo: Ubuntu 14.04+, macOS, Windows

gdb: 7.x or later

Lenguajes: C, C++, go, rust

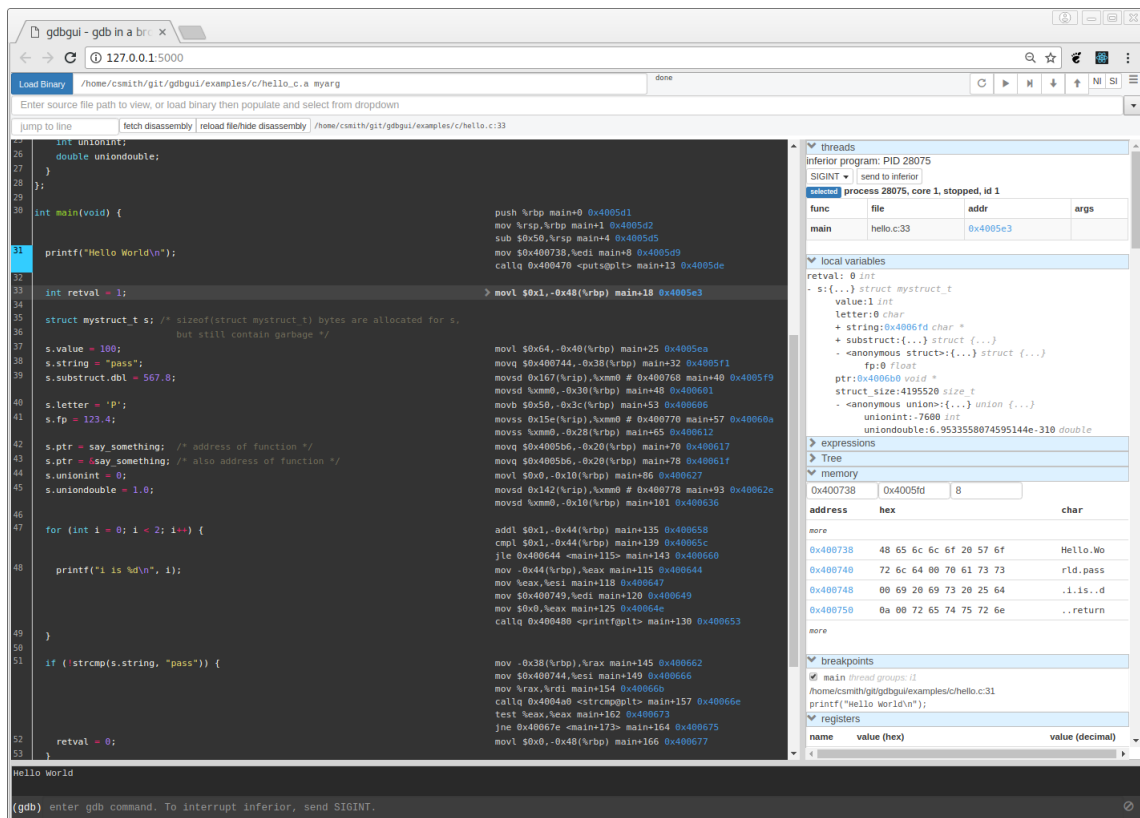
Browsers: Chrome, Firefox. JavaScript y las cookies tienen que estar habilitadas.

Versiones de Python: 2.7, 3.4, 3.5, 3.6, pypy

una vez que se cumplen estos requisitos, se descarga el programa de la url, y se lanza con solo ejecutar en la terminal (en linux):

```
$ gdbgui
```

Se podrá disfrutar de la comodidad del entorno grafico y ademas de la consola de gdb también.



En esta vista de gdbgui uno de los paneles más importantes es el que muestra la Figura ?? allí se ven los valores de las variables locales, los breakpoints, los watchpoints, registros, etc. Toda la información que se necesita para la depuración está en esa ventana

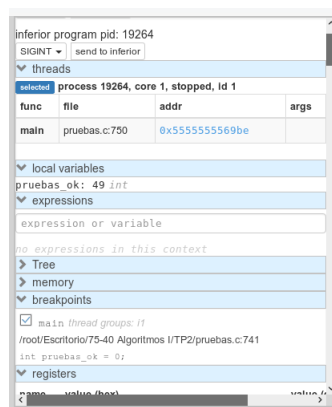


Figura 9: Panel lateral derecho gdbgui

En la parte inferior sigue viéndose la consola de gdb la cual puede ser manejada por los comandos habituales:

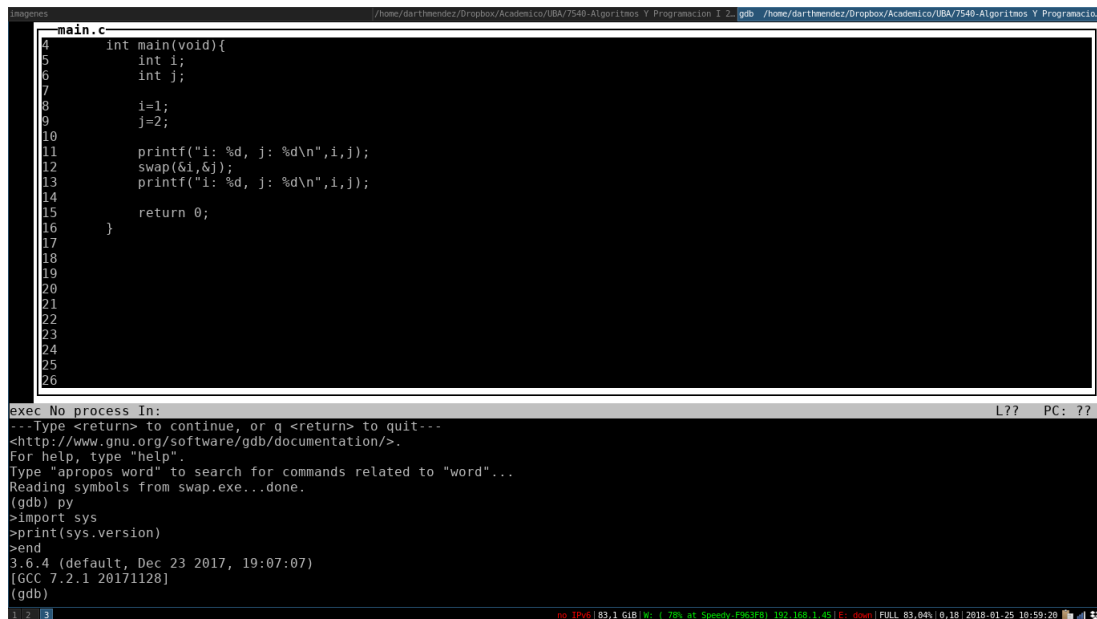


Figura 10: Panel Inferior gdbgui

7. Delikatessen: Consola Python Dentro de Gdb

Aunque muchos no lo sepan gdb tiene un **interprete python embebido**. Para saber que versión es, se puede ejecutar los siguientes comandos python:

```
(gdb) py
>import sys
>print(sys.version)
>end
3.6.4 (default, Dec 23 2017, 19:07:07)
[GCC 7.2.1 20171128]
(gdb)
```



```
main.c
1  int main(void){
2      int i;
3      int j;
4
5      i=1;
6      j=2;
7
8      printf("i: %d, j: %d\n",i,j);
9      swap(&i,&j);
10     printf("i: %d, j: %d\n",i,j);
11
12     return 0;
13 }
14
15
16
17
18
19
20
21
22
23
24
25
26

exec No process in:
...Type <return> to continue, or q <return> to quit...
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from swap.exe...done.
(gdb) py
>import sys
>print(sys.version)
>end
3.6.4 (default, Dec 23 2017, 19:07:07)
[GCC 7.2.1 20171128]
(gdb)
```

De esta forma **se puede ejecutar casi cualquier cosa en python, incluso interactuando con gdb.**

Gdb Cheat Sheet

By Mariano Mendez based on a template of Michelle Cristina de Sousa Baltazar

Running processes

(gdb) run	
(gdb) r	Launch a process.
(gdb) run <args>	
(gdb) r <args>	Launch a process with arguments <args>.
% gdb -args a.out 1 2 3	
(gdb) run	
...	
...	
(gdb) run	Launch a process for with arguments a.out 1 2 3 without having to supply the args every time.
(gdb) set args 1 2 3	
(gdb) run	
...	
(gdb) run	
...	Launch a process for with arguments a.out 1 2 3 without having to supply the args every time.
(gdb) show args	Show the arguments that will be or were passed to the program when run.
(gdb) attach 123	Attach to a process with process ID 123.
(gdb) attach a.out	Attach to a process with process named a.out .

Executing

(gdb) step	
(gdb) s	Do a source level single step in the currently selected thread.
(gdb) next	
(gdb) n	Do a source level single step over in the currently selected thread.
(gdb) stepi	
(gdb) si	Do an instruction level single step in the currently selected thread.
(gdb) nexti	
(gdb) ni	Do an instruction level single step over in the currently selected thread.

More Executing

(gdb) finish	Step out of the currently selected frame.
(gdb) return <exp.>	Return immediately from the currently selected frame, with an optional return value.
(gdb) until 12	Run until we hit line 12 or control leaves the current function.

Breakpoints

A breakpoint is a point in a program at which operation may be interrupted during debugging so that the state of the program at that point can be investigated.

(gdb) break main	Set a breakpoint at all functions named main.
(gdb) break 12	Set a breakpoint in current file at line 12.
(gdb) break test.c:12	Set a breakpoint in file test.c at line 12.
(gdb) clear	Delete all breakpoints.
(gdb) info breakpoints	List all breakpoints.
(gdb) disable break#	Disable the breakpoint <break#>.
(gdb) enable break#	Enable the breakpoint break#.

Watchpoints

A debugging mechanism whereby execution is suspended every time a specified memory location is modified; or, any of various similar such mechanisms.

(gdb) watch main	
(gdb) watch 12	
(gdb) watch test.c:12	Set a watchpoint, same as breakpoints.
(gdb) clear	Delete all watchpoints.
(gdb) info watch	List all watchpoints.
(gdb) disable watch#	
(gdb) enable watch#	Disable / enable the watchpoint watch#.

Listing Source Code

List specified function or line. With no argument, lists ten more lines after or around previous listing. "list -" lists the ten lines before a previous ten-line listing. One argument specifies a line, and ten lines are listed around that line. Two arguments with comma between specify starting and ending lines to list.

(gdb) list	With no argument, list ten more lines after or around previous listing.
(gdb) list 10	
(gdb) list hello.c:10	
(gdb) list hello.c:main	
(gdb) list address	Print informations about the break- and watchpoints.

General Info

(gdb) info args	Print the arguments to the function of the current stack frame.
(gdb) info breakpoints	Print informations about the break- and watchpoints.
(gdb) info display	Print informations about the displays.
(gdb) info locals	Print the local variables in the currently selected stack frame.
(gdb) info signals	List all signals and how they are currently handled.
(gdb) info threads	List all threads.
(gdb) info sharedlibrary	List loaded shared libraries.
(gdb) whatis var_name	Print type of named variable.
(gdb) info registers	Show the general purpose registers for the current thread.
(gdb) info all-registers	Show all registers in all register sets for the current thread.