

Debugging en Eclipse

Autor del video: Marco Gatti

Introducción

Autores de la guía: Joaquín Azcárate y Francisco Bravo

La idea es que terminen de leer esta guía y sean capaces de entender y afrontar cualquier error de programación que tengan. No se cubrirán errores de sintaxis o de semántica, simplemente es una descripción del entorno de Eclipse CDT y su interfaz gráfica del debugger de C, **GDB**.

Todo lo que se ve en este documento es simplemente una forma linda de correr GDB, por lo que los conocimientos son independientes del Eclipse CDT. Para comprender los comandos de GDB que Eclipse nos brinda de forma visual, los redirijo a la **documentación de GDB**.

Todas las imágenes fueron sacadas usando Eclipse IDE for C/C++ Developers en la versión que estuvo en la máquina virtual entregada por la cátedra en el 2013 y 2014. Posiblemente su Eclipse se vea un poco diferente, aunque las funcionalidades son las mismas.

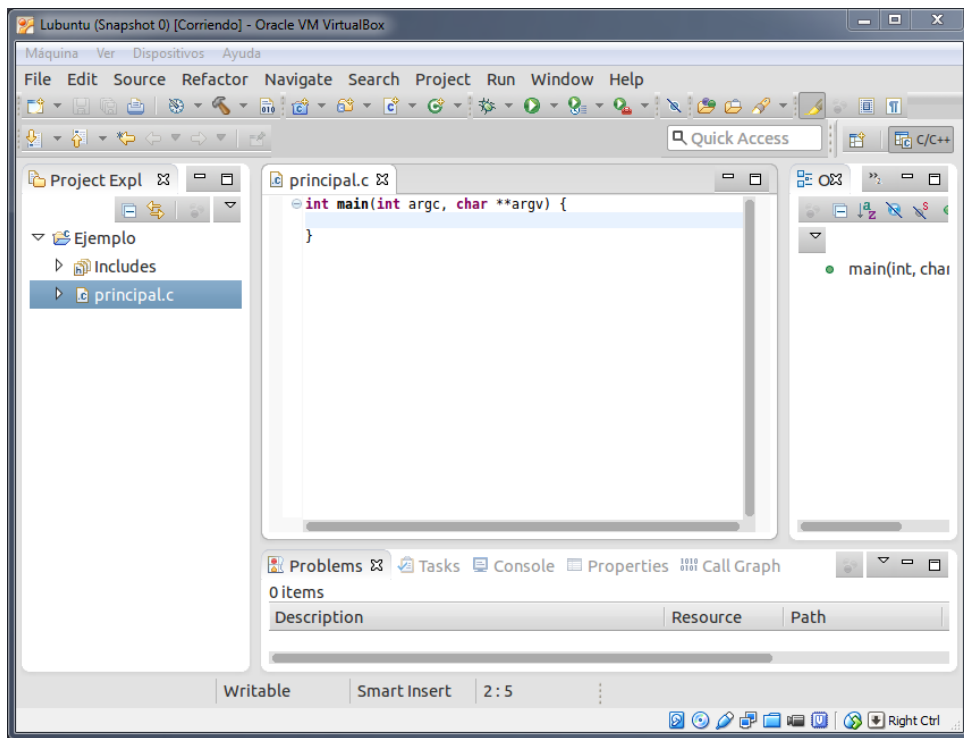
Esta guía plantea un flujo de trabajo que pretende ser interactivo, por lo que les recomiendo seguir, localmente, a la par estos conceptos, detenerse, pensar, probar, codificar...

Vale hacer uso del índice a modo de consulta.

Empezando

Si quieren debuggear, posiblemente ya cuenten con un proyecto creado, pero a modo de ejemplo vamos a crear uno nuevo.

Ahora, con un proyecto vacío, podemos hacerle `Click derecho` a éste y luego `New > Source file` (y le dan un nuevo nombre al archivo). En mi caso, mi proyecto se llama "Ejemplo" con un único archivo "principal.c":



Supongamos que quieren un programa que cree una variable entera, le asigne a esta nueva variable el número 4, y devuelva este valor. Los aliento a que piensen y programen el problema planteado antes de avanzar, así pueden seguir la guía en sus propias computadoras. Suele ser mejor aprender haciendo que solo leyendo; pero ¿que voy a saber yo? Soy un documento en internet.

Primer programa a debuggear

Una vez que tengan programado algo, pueden apretar el boton de `Debug` . Esto ejecutará su programa en modo de debug, donde pueden seguir paso a paso el flujo y los datos de su programa.

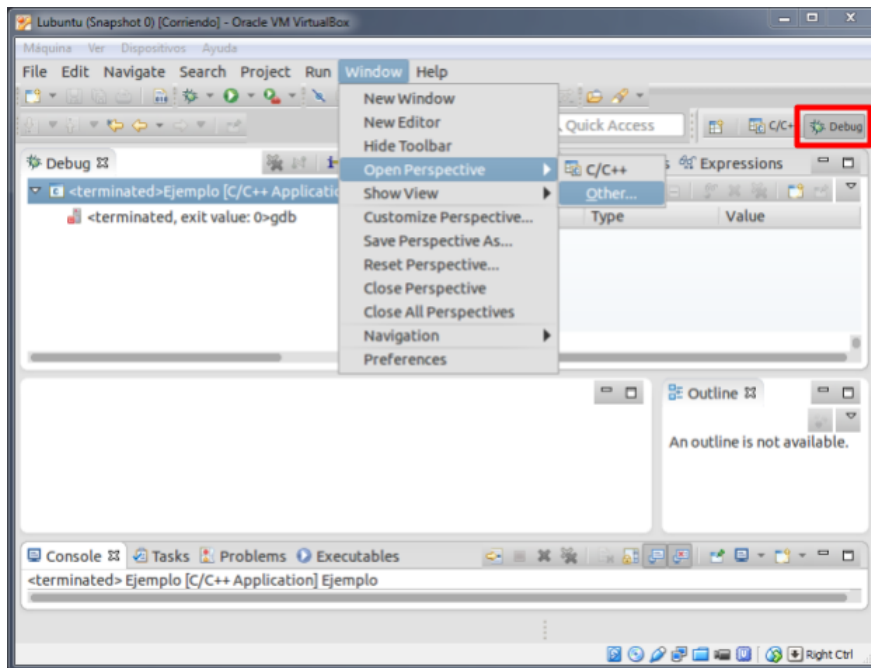


Si es la primera vez que apretan el botón de `Debug` , les preguntará si quieren cambiar de perspectiva^[1]. Te recomiendo decirle que sí y pedirle que recuerde la decisión para que ese cartel no te vuelva a aparecer.

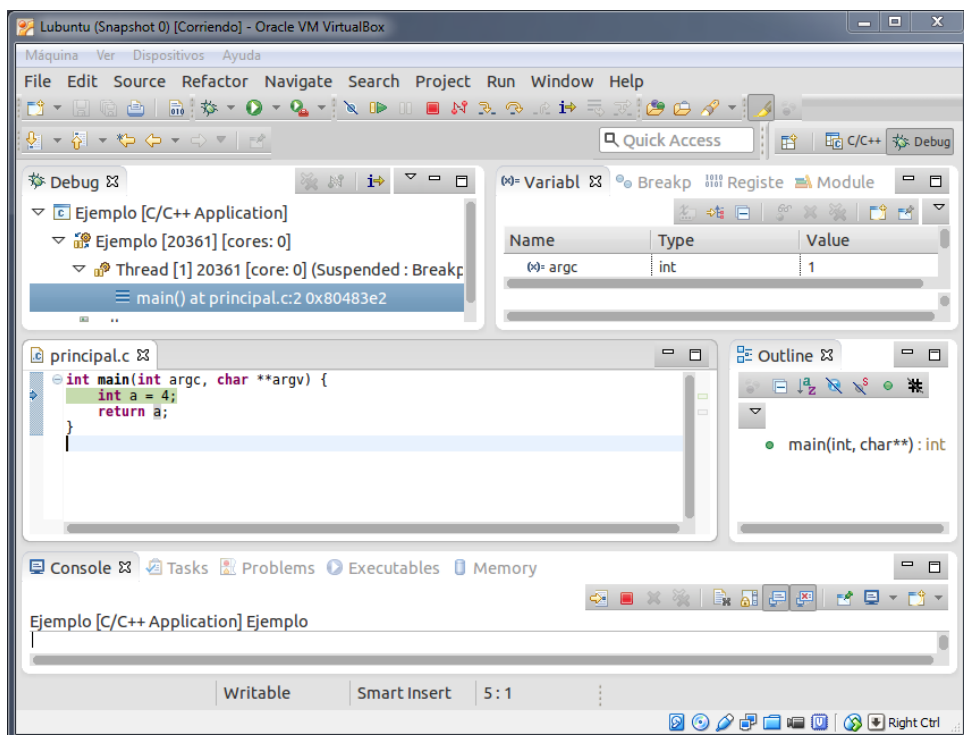
De apretar no, quedarán en la perspectiva de programación, pero no se preocupen, cambiar a la perspectiva de Debug también pueden ir a `Window > Open Perspective > Other... > Debug` .

De ahora en adelante les aparecerá la perspectiva de Debug a la izquierda, junto con la perspectiva de programación, y pueden ir intercambiándolas a medida que necesiten

programar / debuggear.



Entendiendo la perspectiva de Debug



En esta perspectiva tienen 5 marcos diferenciados:

1. Vista **Debug** , donde les muestra qué se está ejecutando en este momento y por qué no avanza su programa. En esta imagen se ve que está ejecutando:

- el proyecto "Ejemplo [C/C++ Application]"
- el ejecutable "Ejemplo" con PID 20361 en el núcleo 0

- en un hilo con TID 20361
- suspendido por un breakpoint
- con el stack con una sola función `main()` .

2. Un marco con las vistas de `Variables` , `Breakpoints` , y otras dos:

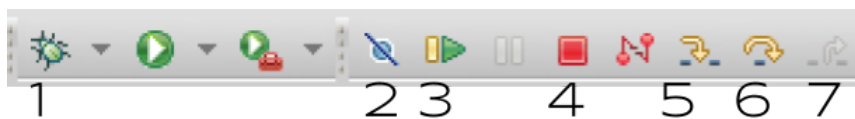
- Vista `Variables` muestra todas las variables locales en el stack actual.
- Vista `Breakpoints` muestra los breakpoints que tenemos asignados.

3. Vista de archivos, particularmente solo vemos nuestro único archivo "principal.c" y está pintada la línea a *ejecutar* (es decir, todavía no se ejecutó `int a = 4;`).

4. Vista de `Outline` , es una forma de acceder rápidamente a las funciones que se encuentran en la vista de archivos.

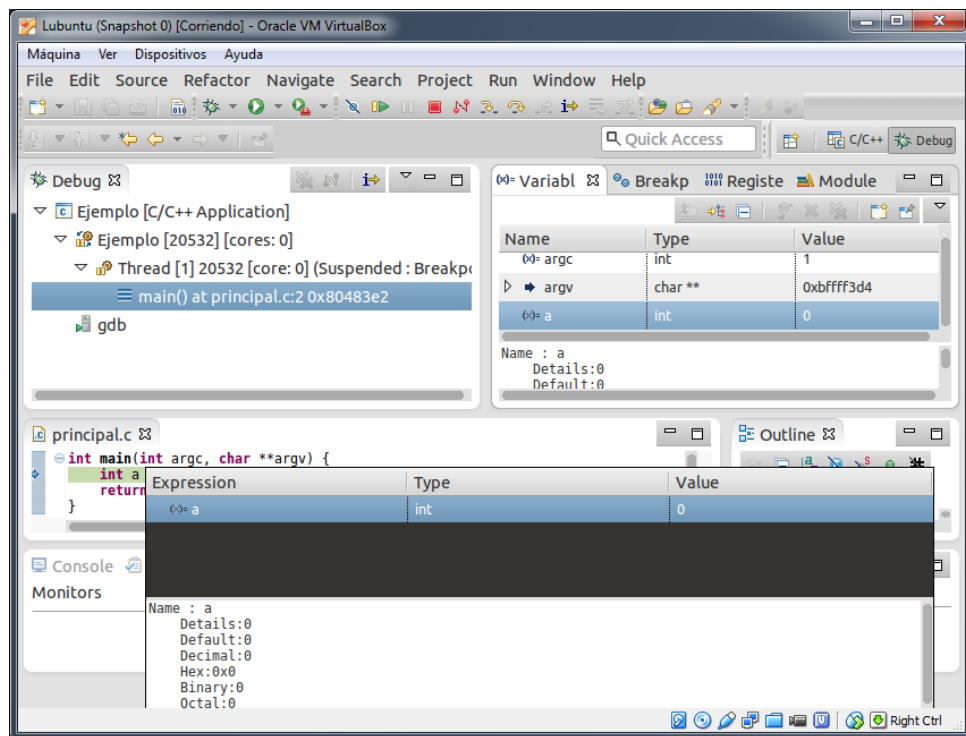
5. Un marco abajo con `Console` , que muestra lo que mostraría la consola en caso de que ejecutemos el programa normalmente... y otros agregados.

Cómo correr un programa



1. Como ya vimos, el botón `Debug` empieza la ejecución en modo Debug. Antes de apretarlo, el resto de los botones marcados permanecen en gris.
2. Ignorar todos los breakpoints, no frena en ningún breakpoint.
3. `Resume` (F8): Ejecutar todo lo que puedas hasta que encuentre un breakpoint activo (es decir, que no esté marcado como ignorado).
4. `Stop` : Terminar el proceso abruptamente.
5. `Step Into` (F5): Ejecutar en la mínima granularidad, es decir que, de haber un llamado a una función, va a entrar a ella.
6. `Step Over` (F6): Ejecutar toda la línea. De ser una línea con llamadas a funciones, las ejecuta pero no frena en ninguna (a menos que exista un breakpoint activo dentro de alguna función llamada).
7. `Step Return` (F7): Ejecuta el resto de la función actual hasta llegar a la línea de donde fue llamada. En este ejemplo, al tener solo la función de Return, no podemos "salir".

Vista Variables



Recordemos por dónde quedamos: estamos con nuestro programa corriendo en modo Debug, por lo que frena antes de ejecutar cualquier cosa.

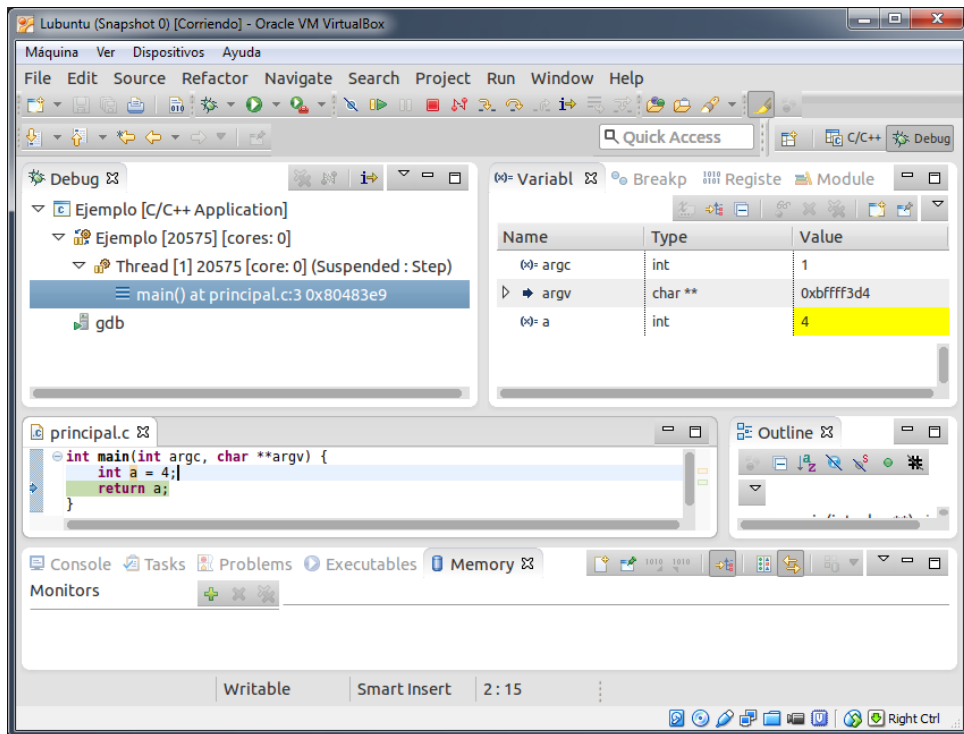
Tenemos marcada la primera línea, por lo que todavía no se ejecutó. Si ponemos el mouse arriba de la variable `a`, veremos que es de tipo `int` con el valor 0. Esto mismo podemos verlo en la vista de Variables.

Junto con las variables locales, nos aparecen los parámetros de la función; en este caso, `argc` y `argv`.

Intentemos recordar cómo podemos ejecutar esta línea.

Pausa dramática y ruido de "pensamiento"...

Si optaron por `Step Into` o `Step Over`, están en lo correcto:



Como vemos, ahora está pintada la siguiente línea, el valor de `a` se alteró y quedó pintado de amarillo.

Programa Complejo

Primero que nada, armemos un programa más complejo...

Supongamos que viene la NASA y nos encarga armar un programa que sea capaz de almacenar patentes (3 letras y 3 números), y queremos almacenar las siguientes: "ABC 123", "SQL 035", "UTN 999".

De nuevo, los invito a pensar la solución en C.

Pausa para pensar...

► Spoiler

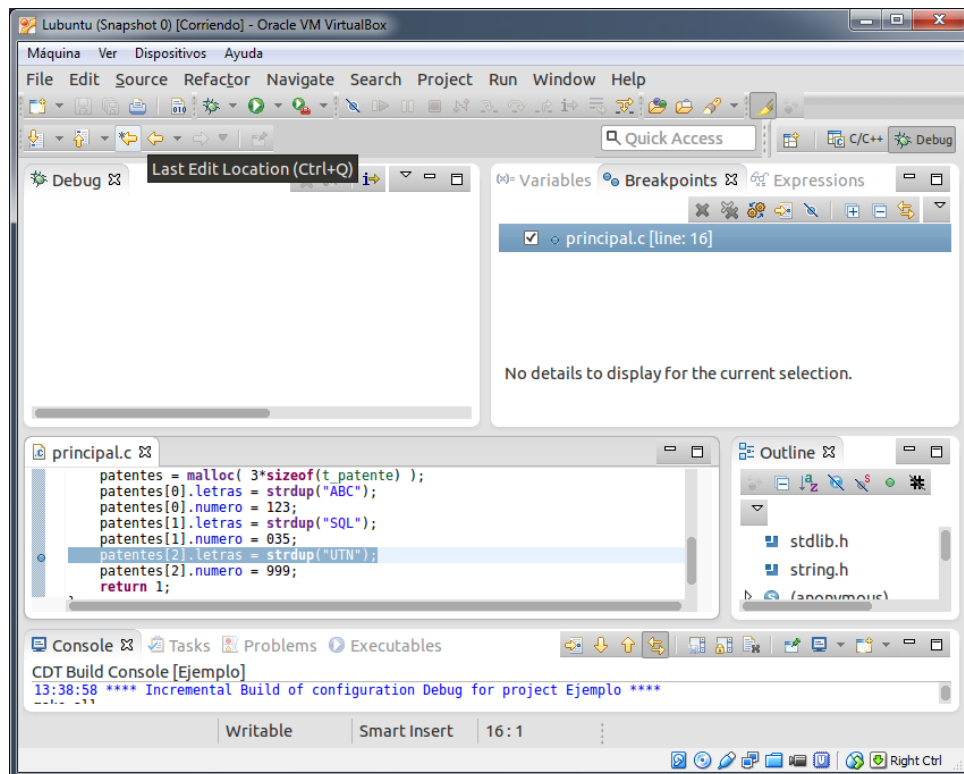
Intentemos correrla como debug y frenar justo antes de insertar la palabra "UTN".


Probablemente hayan apretando como 6 veces `step over`. Tiene que existir una mejor forma de hacer esto y no tener que contar los steps 🤔

Breakpoints

Agregar un breakpoint es tan simple como hacer doble click en la barra a la izquierda de la línea de código que queremos interrumpir.

Esto se puede hacer tanto en la perspectiva de programación C/C++ como en la de Debug. Incluso se pueden hacer mientras el programa esté en ejecución (o frenado) en modo Debug.



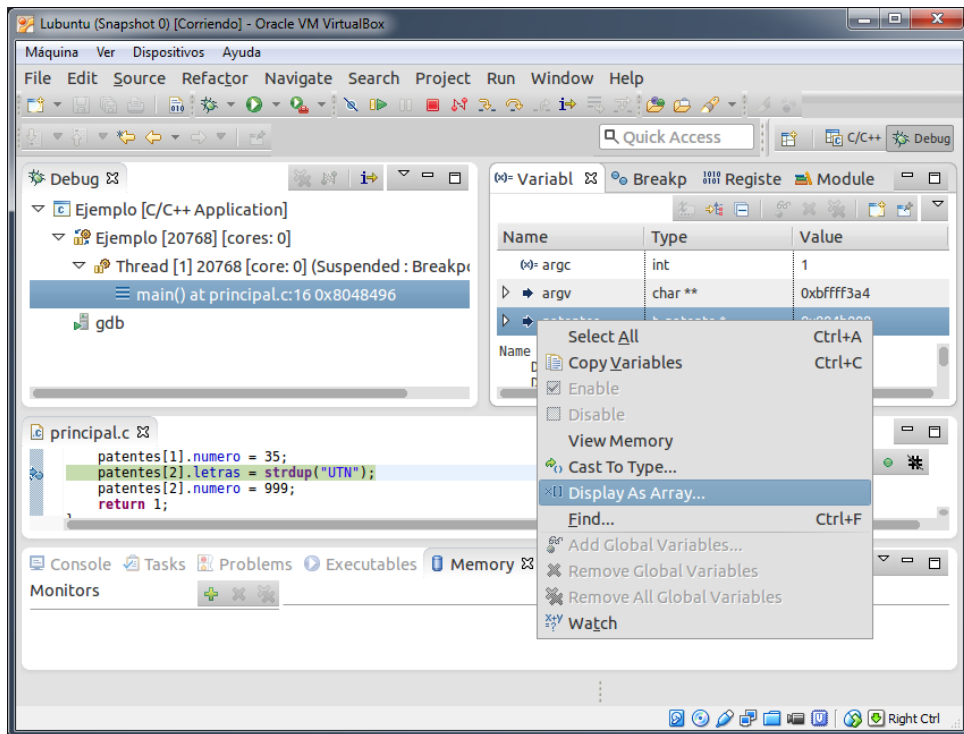
Adicionalmente, en la perspectiva de Debug está la vista de `Breakpoints`, donde pueden ver los breakpoints que tengan y desactivarlos temporalmente haciendo click en el .

Ahora podemos correr en Debug nuestro programa y darle `Resume` tranquilos, porque la ejecución se va a trabar donde pusimos el breakpoint, sin ejecutar la línea seleccionada.

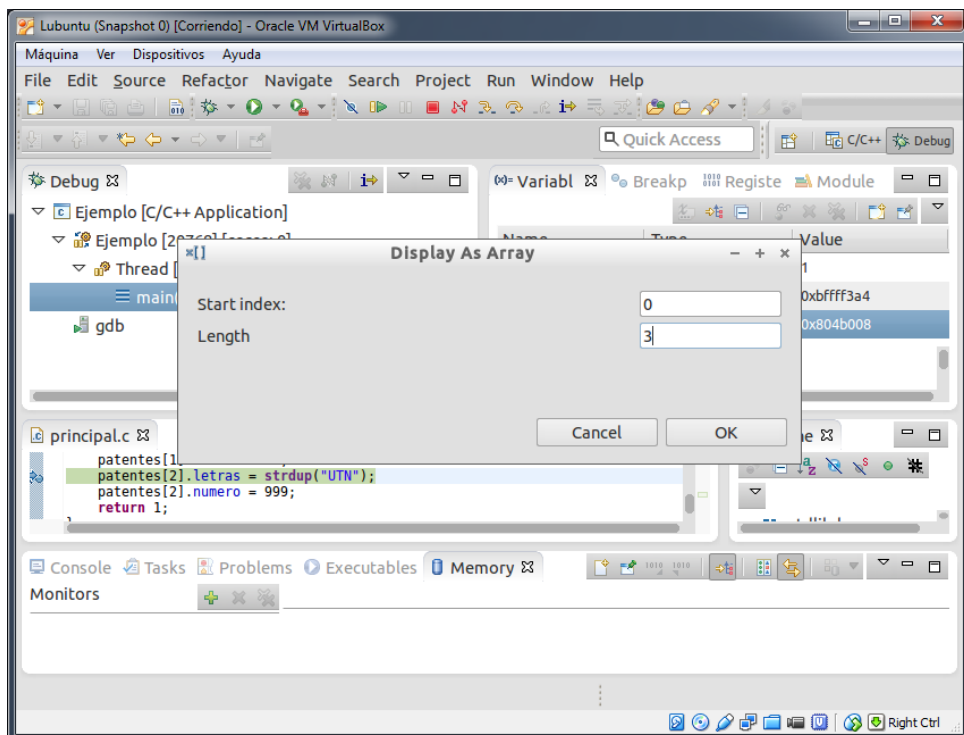
Cuando logres hacer esto, intentá mirar la variable local `patentes`.

Como esta variable es un puntero, GDB no puede suponer si apunta a un único elemento o si es un vector. Por lo tanto, cuando lo ves con la vista, solo vas a ver el primero. ¿Cómo podemos arreglar esto? 😞

Casteos en Variables



Si uno apreta Click derecho en la variable que queremos, nos va a aparecer la opción de ver la memoria tal cual la grabó, castear a un tipo particular o, como queremos ahora, mostrar como una array.

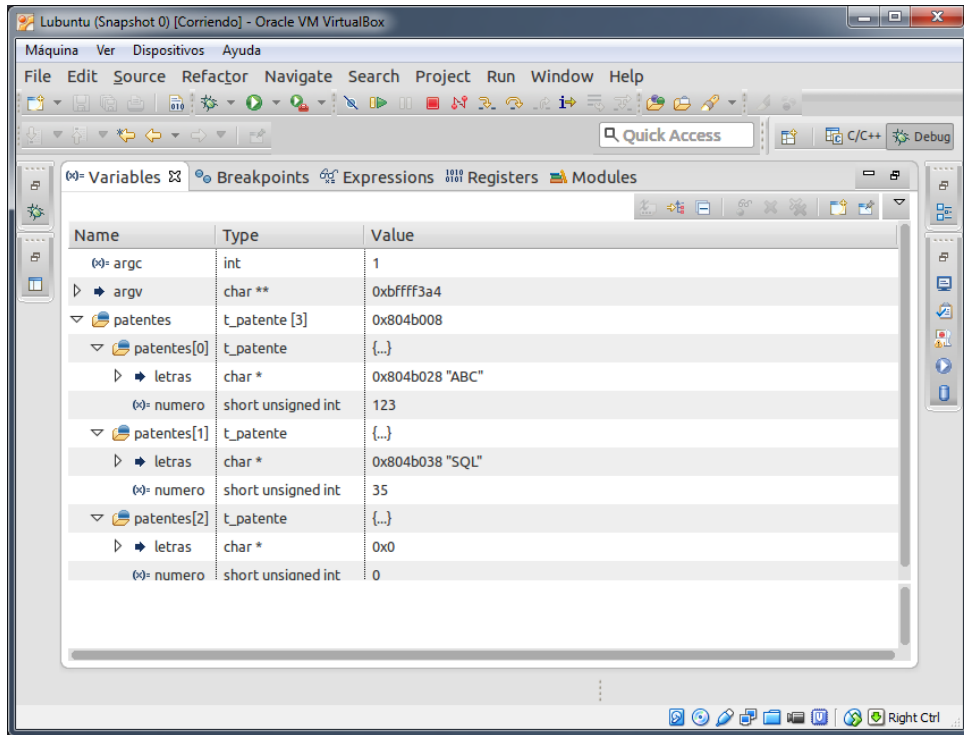


Recordemos que esto ya esta en ejecución, por lo que, si quisiéramos mostrar un array de como 100 elementos, las patentes a partir de la 4 estarían fuera del segmento alocado y tendría que dar un fallo de segmento.

Pueden probarlo, pero esto no pasa porque solo nos muestra la memoria. Si casualmente hay algo, nos va a mostrar algo corrupto; o, si no hay nada asignado después de la posición, nos dice que no puede accederla.

TIP

Si uno hace doble click en una pestaña, (en este caso, la vista de Variables), se agranda. Volver a hacer doble click vuelve todo a la normalidad.



Y vemos exactamente el valor de cada variable dentro de un vector.

Podríamos tener un vector dentro de otro. De ser así, repetimos este proceso de `Click derecho > Display as Array...`

Programa Complejo Complejo

Supongamos que queremos refactorizar nuestra solución y generar un TAD de patentes; algo que se vea así:

```
#include "patentes.h"

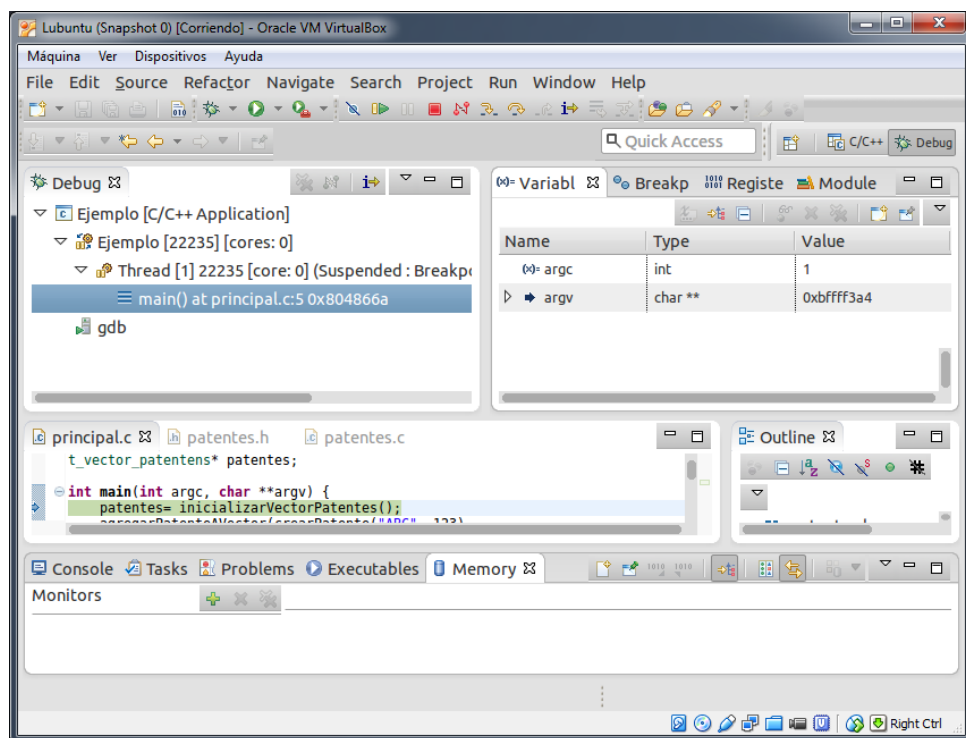
t_vector_patentes *patentes;

int main(int argc, char **argv) {
    patentes = inicializar_vector_patentes();
    agregar_patente(patentes, crear_patente("ABC", 123));
    agregar_patente(patentes, crear_patente("SQL", 035));
    agregar_patente(patentes, crear_patente("UTN", 999));
    imprimir_vector_patentes(patentes);
    destruir_vector_patentes(patentes);
    return 1;
}
```

Y queremos un resultado en la consola como:

```
Patente 0: ABC 123
Patente 1: SQL 035
Patente 2: UTN 999
```

Antes de empezar a pensar la solución, esto es lo que pasaría de correr como Debug ese código:



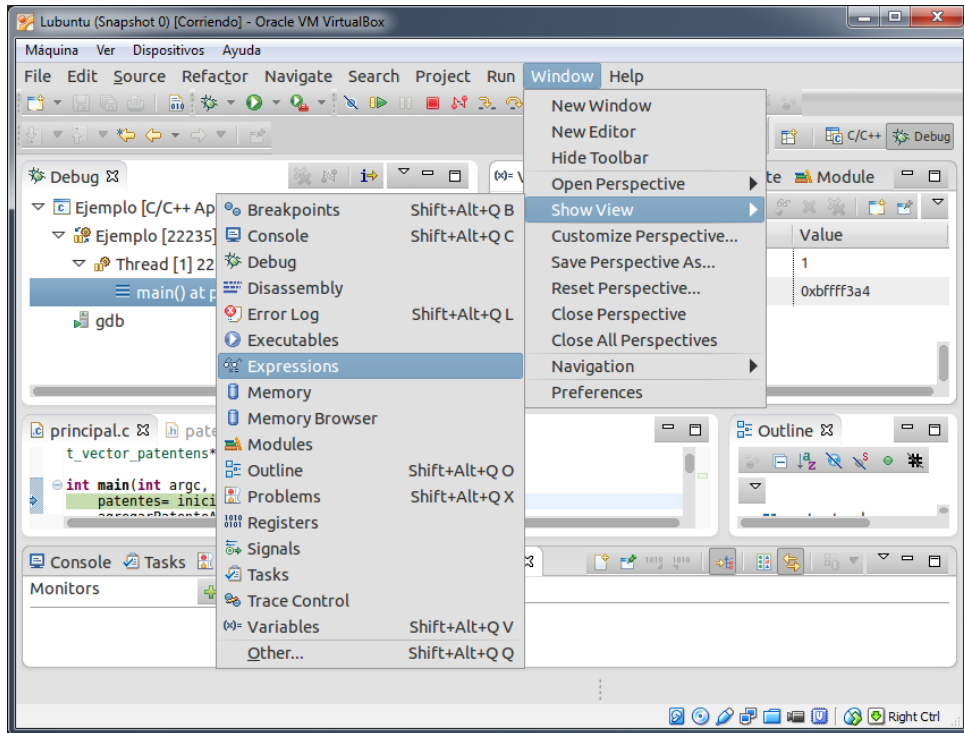
¿Notás cómo en la vista de `Variables` no está la variable `patentes` ?

Esto no es un *bug*, sino que, como `patentes` fue definida de forma global, no aparece junto con las variables locales. Entonces, ¿cómo podemos ver el contenido? 🤔

Variables globales

Una solución válida es poner el mouse arriba, pero aprovecho esta situación para introducirles la vista de `Expresiones` .

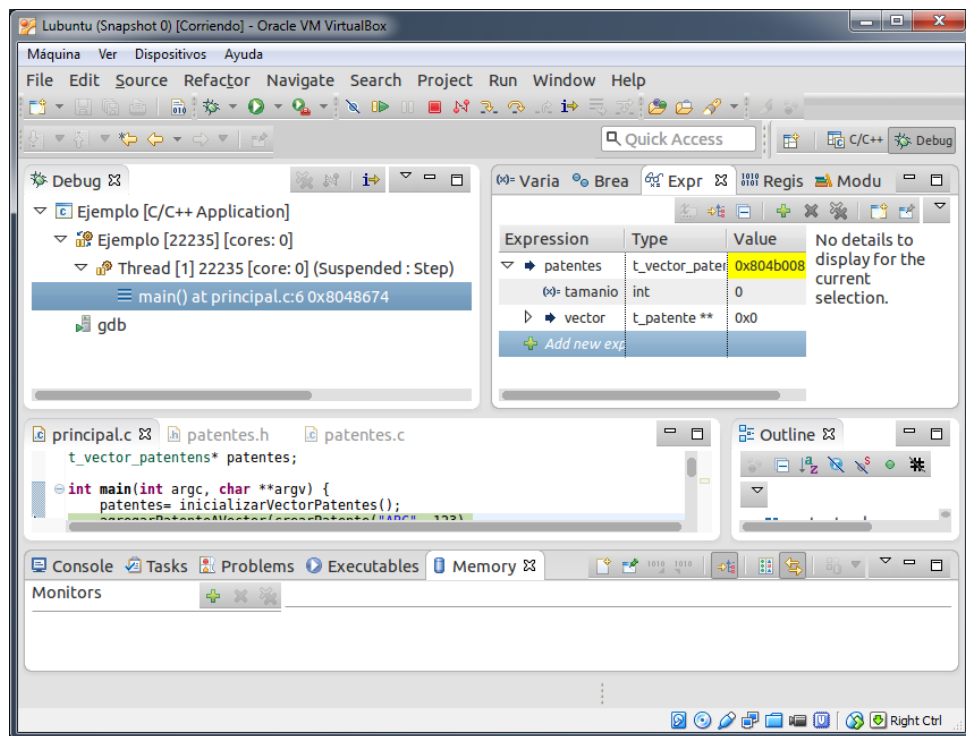
Muy posiblemente no tengan esta vista por defecto: para accederla uno puede ir a `Window` > `Show View` > `Expressions` .



Ahora tenemos, en el mismo marco que `Variables` , una nueva pestaña de expresiones con un `+`, donde podemos hacer click y escribir la expresión que querramos ver.

Recordemos que el programa ya está en ejecución, por lo que no podemos pedir que ejecute una función, solo podemos ver variables.

Si escribimos `patentes` , podemos acceder a la variable global tal como haríamos en la vista de `Variables` .



Ahora si, a programar una posible solución.

Pausa para pensar...

Supongamos que generamos esta función que inicializa nuestro TAD, e intentamos ejecutar esta instrucción:

```
#include "patentes.h"
t_vector_patentes *patentes;

int main(int argc, char **argv) {
    patentes = inicializar_vector_patentes();
    agregar_patente(patentes, crear_patente("ABC", 123));
    agregar_patente(patentes, crear_patente("SQL", 035));
    agregar_patente(patentes, crear_patente("UTN", 999));
    imprimir_vector_patentes(patentes);
    destruir_vector_patentes(patentes);
    return 1;
}
```

Tenemos una función cuyo argumento es otra función.

- ¿Qué pasaría si apretamos Step Over ? 🤔

Step Over ejecutaría ambas funciones y nos frenaría en la línea que se encuentra debajo de ésta (la que intenta crear y agregar una nueva patente).

- ¿Qué pasaría si apretamos Step Into? 🤔

Step Into ejecutará la llamada a la primera función. En este caso, crearPatente() (tal como en matemática, se procesa primero lo que está dentro de los paréntesis).

Supongamos un código de crearPatente de este estilo:

```
#include "patentes.h"

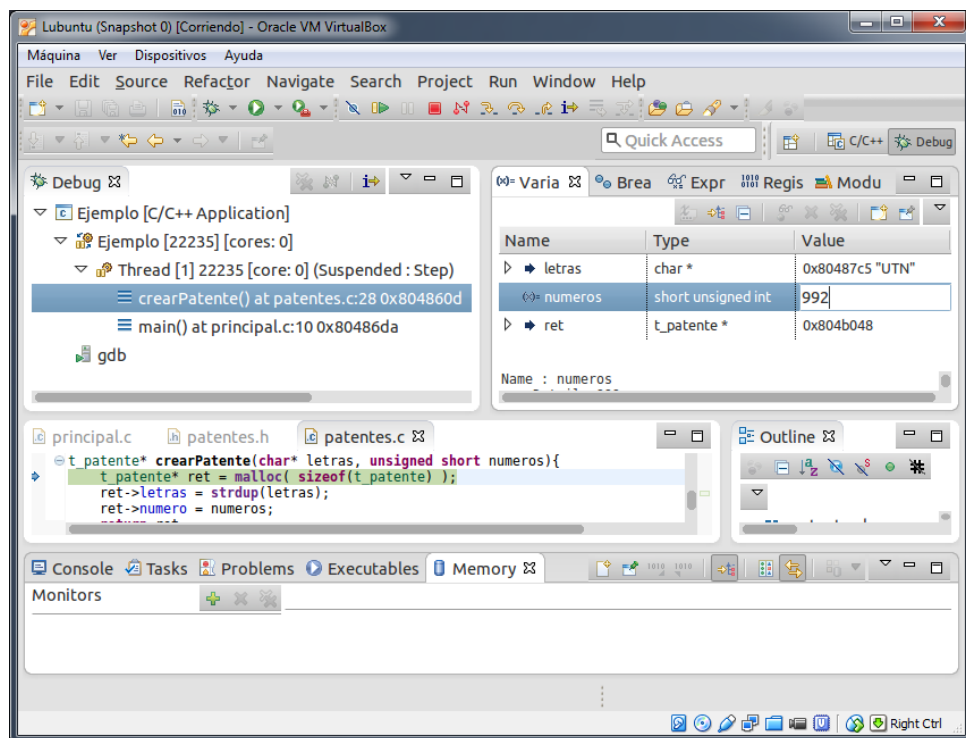
t_patente* crear_patente(char* letras, int numeros){
    t_patente* ret = malloc(sizeof(t_patente));
    ret->letras = strdup(letras);
    ret->numero = numeros;
    return ret;
}
```

Al hacer Step Into podemos ver que (dentro de la vista de Debug) en el stack aparece que estamos dentro de la función `crearPatente()` , llamada por `main()` .

Editar memoria *on the fly*

Antes de seguir adelante, es una buena oportunidad de comentarles que dentro de la vista de `Variables` (y de `Expressions`) uno puede alterar el valor de una variable.

Haciéndole doble click al Value, uno puede ingresar lo que quiera y, de ese punto en adelante, el programa tendrá ese nuevo número. **Usar con cuidado.**



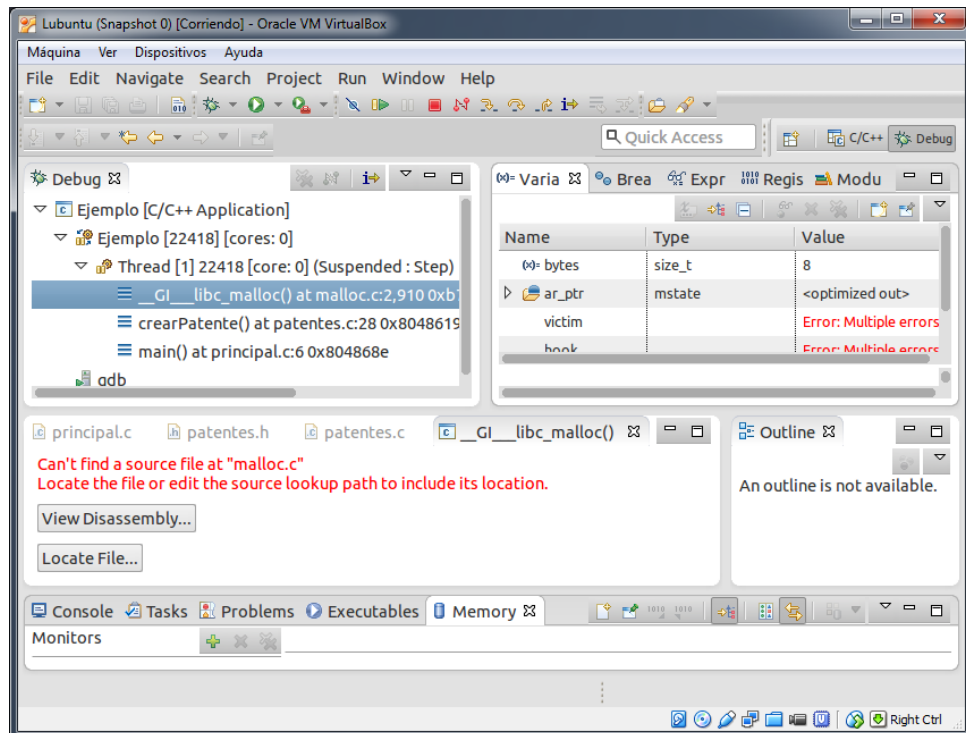
Ahora estamos en una línea con una llamada a una función que no programamos nosotros: `malloc()` .

Si apretamos Step Into, ¿qué tendría que pasar? 🤔

Pausa para pensar...

Can't find source file at: El error que no es

Si, se merece todo un título para esto nomás.




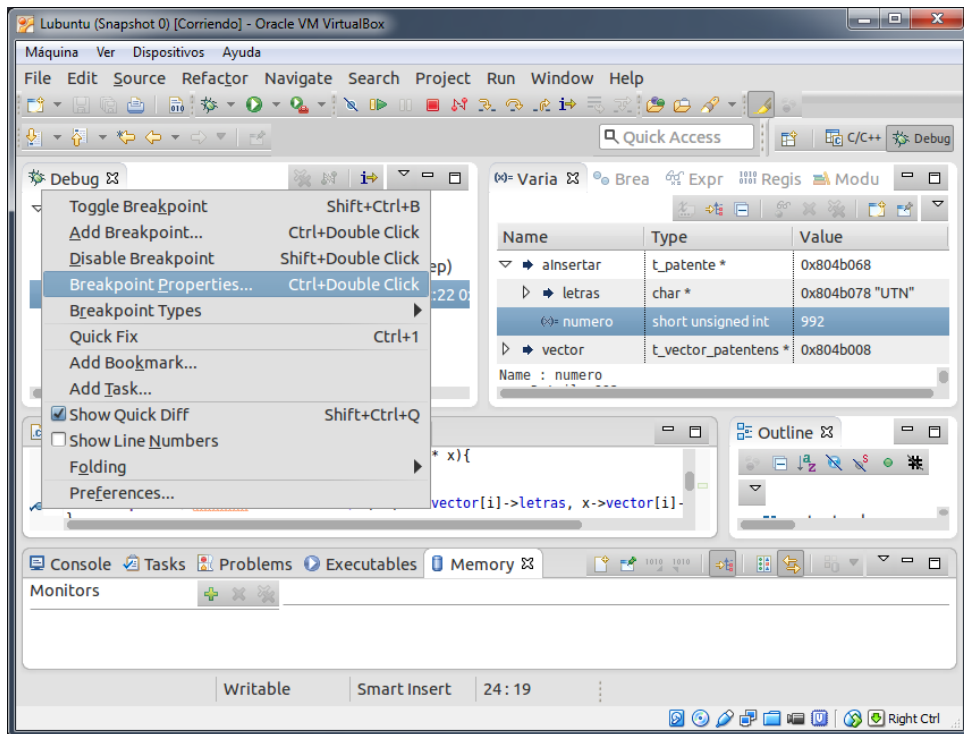
Al igual que el `Step Into` anterior, el stack crece y se llama a `malloc()` , pero como está programado por otras personas y no tenemos acceso al código, Eclipse nos informa que no puede encontrar el código.

Sí puede encontrar la biblioteca compartida y ejecutar el código; el problema es que ese código tiene a su vez varias llamadas a otras funciones y varias líneas, por lo que entrar en pánico y apretar `Step Into` y `Step Over` nos lleva más adentro de la madriguera del conejo.

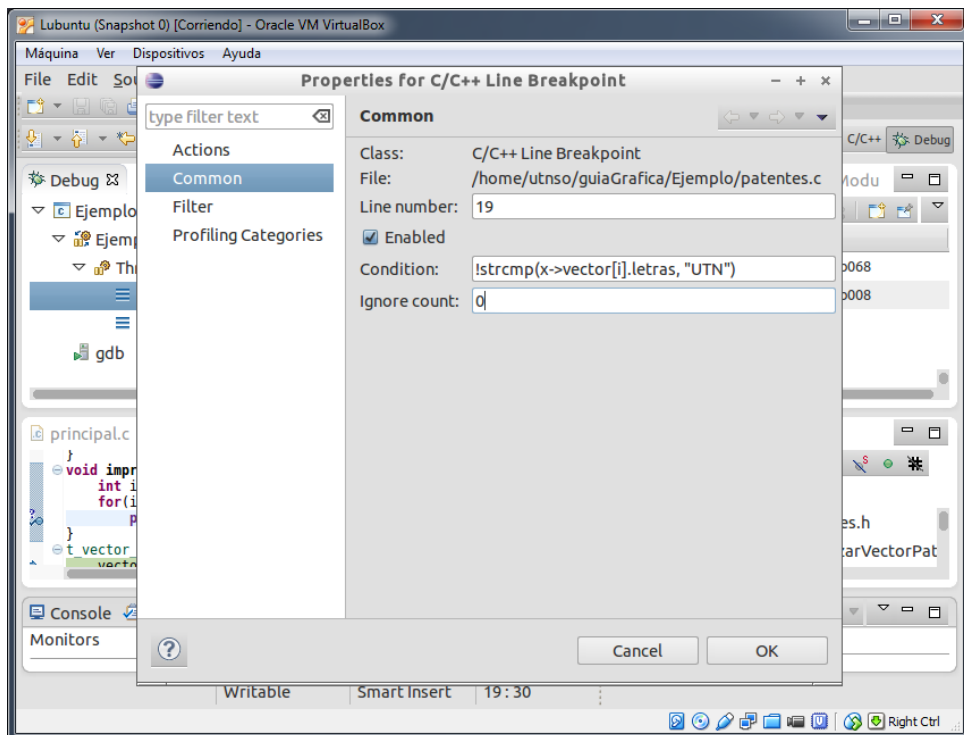
Lo que nos lleva a la salvación es el `Step Return` , que nos devuelve a la función de donde se llamó, en este caso `crearPatente()` .

Breakpoints RELOADED

Una vez que tenemos breakpoints, uno puede hacer click derecho en la pelotita  e ir a sus propiedades.

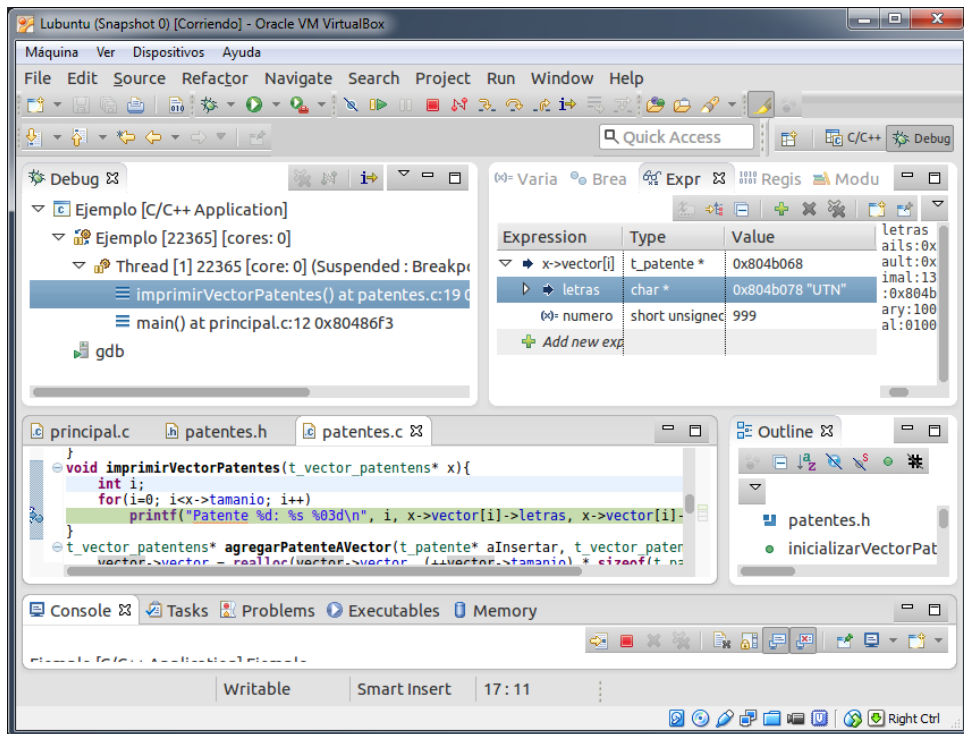


Por ejemplo, podríamos querer un breakpoint que interrumpa la ejecución en el código de imprimir las patentes **solo si** las letras del vector a mirar son "UTN":



Notar que en esta condición tenemos una llamada a una función y evaluamos el retorno. Sí, esto se puede hacer.

Una vez creado este nuevo breakpoint, podríamos *resumir* nuestra solución y ver si efectivamente frena en algún lugar.



Técnicas más avanzadas de debug

Antes de terminar, te dejamos un par de videos que explican técnicas más avanzadas para debuggear en Eclipse, que te pueden servir para casos muy particulares.

Desde una terminal externa

Autor del video: Gastón Prieto

Si tu TP requiere usar la **biblioteca gráfica de la cátedra**, notarás que no es posible ejecutar y debuggear desde Eclipse, ya que la consola integrada no es interactiva.

La solución a esto es debuggear en forma remota utilizando una terminal externa como en el siguiente video:

Partiendo de un proceso que ya crasheó

Autor del video: Matías García Isaia

Cuando ejecutamos una aplicación desde la consola y ésta es interrumpida por una señal (como, por ejemplo, `SIGSEGV`, que es `segmentation fault`), se *dumpea* el estado del proceso en un archivo.

Esto nos permite debugear la aplicación después de que se haya interrumpido, lo cual se conoce como **debug postmortem**:

The End

Ahora, ¡a debugear!

1. Una perspectiva es una forma diferente de ver el Eclipse, donde cambian las vistas y la forma en la que están organizadas ↩
2. Salvo que ustedes sí habrán puesto los `free()` al final, ya que son buenos programadores 🙄 ↩