

depuración de código en C con VSCode

Ernesto Bossi

Table of Contents

I: Introduccion	1
1. Antes de empezar	2
1.1. Objetivo	2
1.2. Que es un depurador (debugger)?	2
1.2.1. Por que no usar gdb directamente?	3
1.2.2. Que beneficios nos brinda depurar en Visual Studio Code?	3
2. Preparando el entorno	5
2.1. Instalando dependencias	5
II: depuración de código C en VSCode con GDB	7
3. depuración en Linux/windows	8
3.1. Introduccion	8
3.2. Antes de empezar	8
3.3. Depurando en VScode	9
3.4. Secciones básicas del depurador	12
3.5. Continuando con la ejecución	17
III: Extras	20
4. FAQ	21
4.1. Agregue el flag -g a la configuracion de mi Makefile, pero aun no toma los breakpoints seteados en el editor de VSCode. Porque VSCode no toma los breakpoints?	21
4.2. Que pasa si me aparece un error como el siguiente > Unable to determine path to debugger?	21
5. Apendice	23
5.1. Configuraciones de ejemplo	23
5.1.1. Configuracion de ejemplo e inicial para Windows	23
5.2. Configuracion de ejemplo e inicial para Linux	23
5.3. Configuracion de ejemplo e inicial para MacOS	24
6. Changelog	25
6.1. Sobre el Changelog	25
6.1.1. 1.0.2 (2022-08-04) - @bossiernesto	25
6.1.2. 1.0.1 (2022-05-25) - @bossiernesto	25
6.1.3. 1.0.0 (2022-05-22) - @bossiernesto	25

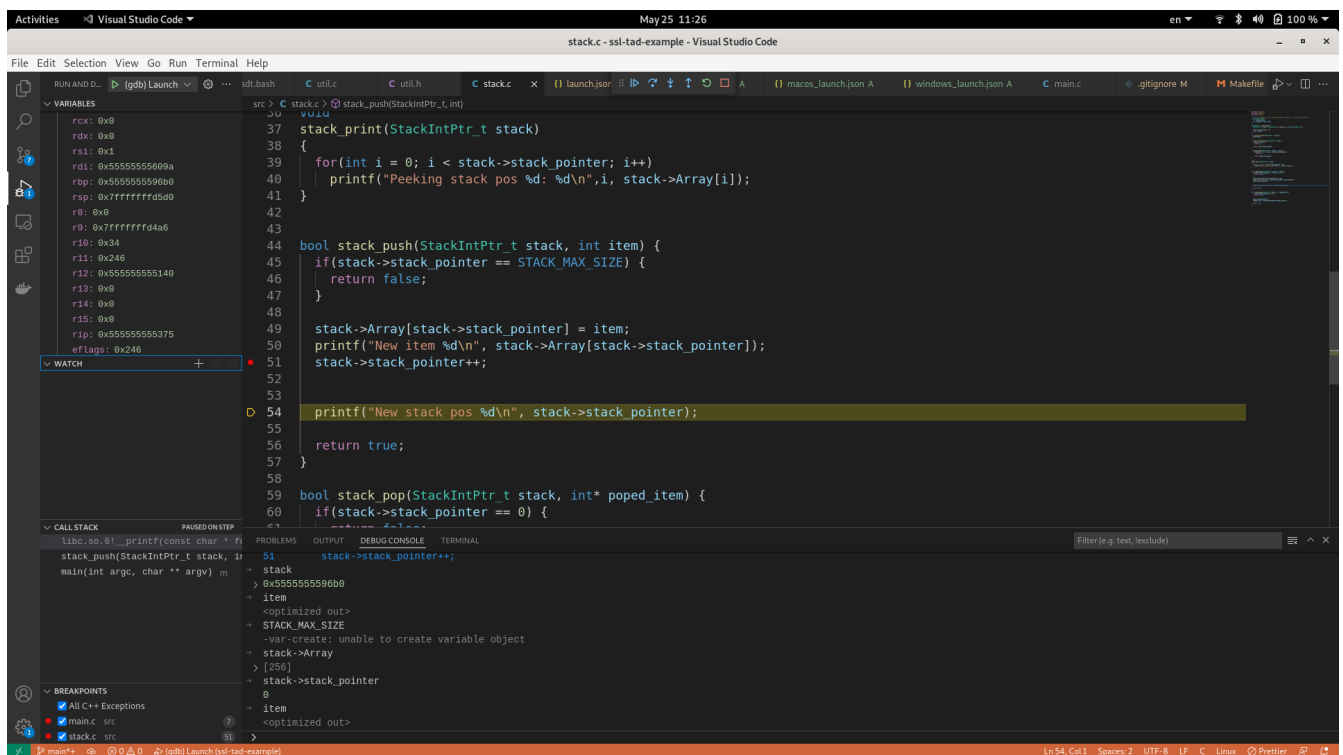
Part I: Introduccion

Chapter 1. Antes de empezar

1.1. Objetivo

Este documento describirá los pasos para poder empezar a utilizar el modo de **depuración** (debugging) en VSCode en Linux, MacOS y Windows.

Este documento tiene como propósito, describir los requisitos, y pasos a seguir para poder utilizar el depurador de C **gdb** en Linux y windows.



La elección de Visual Studio Code^[1], fue debido a que la mayoría de los alumnos que cursan *Sintaxis y Semántica de los Lenguajes* utiliza dicho entorno para programar en C.

1.2. Que es un depurador (debugger)?

El propósito de un depurador (**debugger**), tal como lo es **gdb**^[2], es el de permitir ver lo que hay "adentro" de otro programa mientras se ejecuta, o lo que este otro programa estaba haciendo antes de fallar.

El depurador **gdb** puede hacer cuatro tareas principales, existen muchas cosas más que pueden hacer pero las que nombraremos son las principales:

- Inicializar un programa, especificando cualquier cosa que afecte su comportamiento.
- Poder parar la ejecución del programa en condiciones específicas, por medio de **breakpoints** (puntos de corte) y condiciones de punto de corte.
- Examinar e investigar que ha sucedido con el programa, cuando la ejecución se detuvo por un

punto de corte. Examine what has happened, when your program has stopped.

- Cambiar cosas en el programa, para poder experimentar los cambios que generarian en el programa dichos cambios.

Ademas este depurador, funciona en muchos lenguajes, por lo que puede usarse esta misma herramienta en ellos. Los lenguajes que soporta GDB son:

- Assembly
- C/C++
- D
- Go
- Objective-C
- Pascal
- Rust



Esta guia detallara los pasos para depurar código solo en C/C++.

1.2.1. Por que no usar gdb directamente?

Si bien GDB puede utilizarse sin necesidad de usar Visual Studio Code, se recomienda usar este por medio de este entorno, debido a que la interfaz que expone GDB puede ser algo difícil de dominar al comienzo^[3].

Ademas Visual Studio Code, no posee un depurador por su cuenta, sino que utilizaremos el depurador gdb, que es bastante potente, y el entorno de VScode, nos dara una interfaz mucho mas amigable sin renunciar a utilizar el depurador *de facto* de C/C++.

1.2.2. Que beneficios nos brinda depurar en Visual Studio Code?

Tal como se menciono antes, Visual Studio Code utilizara el depurador gdb, y permitira que se pueda depurar código, de manera simple, y que se puedan establecer breakpoints sin necesidad de tener que modificar o agregar lineas adicionales a nuestro código existente.

Con gdb en Visual Studio Code podremos hacer lo siguiente:

- Puntos de corte (Break points)
 - break points
 - condition break points
 - function break points
- Tipos de ejecución en modo depuración
 - Step Over
 - step into
 - Step out

- Continue
- Call Stack
- Variables
 - Ver variables
 - Observar variables (watch variables)

Desgraciadamente no podemos hacer las siguientes acciones en modo depuración con gdb en Visual Studio Code:

- Step Back
- Move to

[1] <https://code.visualstudio.com/>

[2] <https://www.sourceware.org/gdb/>

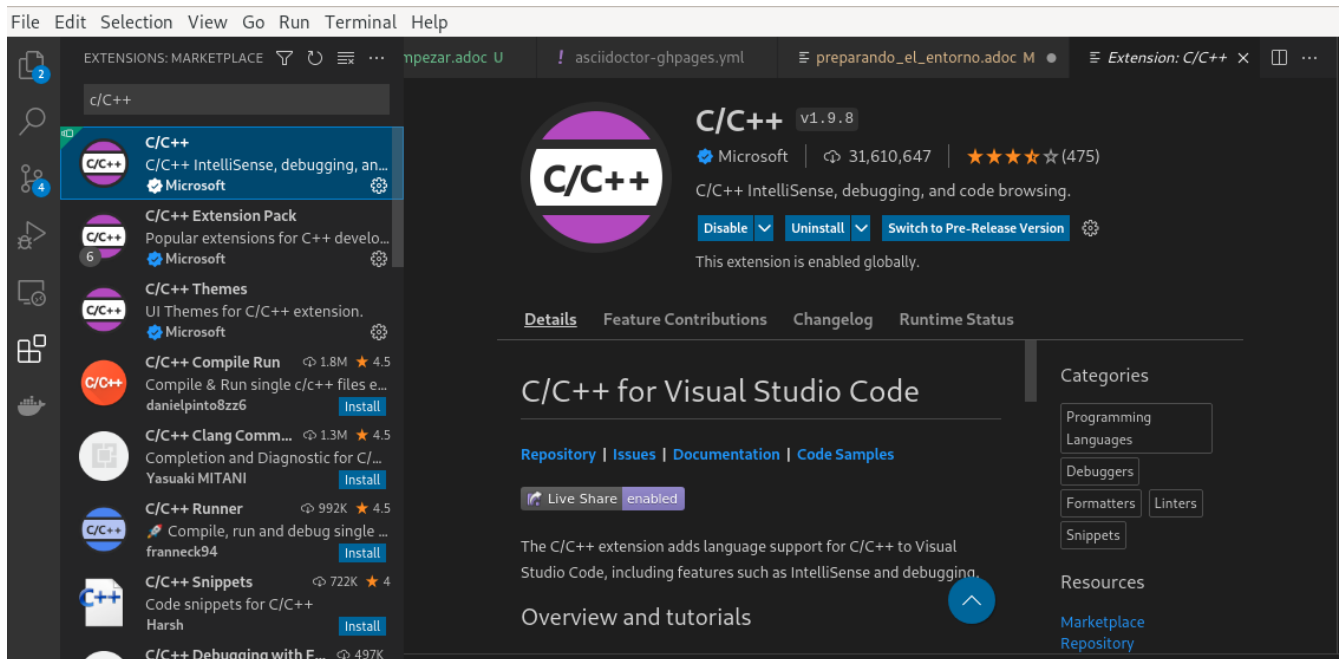
[3] <https://www.eecs.umich.edu/courses/eecs373/readings/Debugger.pdf>

Chapter 2. Preparando el entorno

2.1. Instalando dependencias

Primero hay que instalar las distintas dependencias para empezar a utilizar el depurador, siendo la primera, la instalacion de [VSCode aqui](#).

Tambien hay que instalar la [extension de C/C++](#). Aunque una vez que tengamos instalado el entorno, podemos ir a la sección de extensiones e instalarlo desde ahi.



Obviamente necesitaremos el depurador instalado tambien, por lo que deberiamos tener es el depurador en si, para eso si estamos en linux deberemos instalar el compilador y depurador por separado:



En el caso de Arch Linux^[1], hay que usar *pacman* en vez de *apt*, esta sentencia solo es aplicable a Ubuntu/Debian.

```
$ sudo apt-get install gcc gdb
```

En el caso de **Windows**, tendremos que instalar MingGW-64^[2] [aqui](#).



Una herramienta que puede ayudarnos bastante a instalar el depurador y compilador de C/C++, y que nos de una consola que nos ayude a instalar dependencias y una terminal similar a **nix* es MSSYS2^[3].

Una vez instalado el depurador (gdb), compilador (gcc) y entorno (VSCode), estamos listos para empezar.

[1] <https://archlinux.org/>

[2] <https://www.mingw-w64.org/>

[3] <https://www.msys2.org/>

Part II: depuración de código C en VSCode con GDB

Chapter 3. depuración en Linux/windows

3.1. Introduccion

Esta sección tratara el uso y los pasos para empezar a depurar código en C. Se describira como empezar la depuración de cualquier programa con y sin uso de **Makefiles**.

El objetivo de esta sección es la de describir un uso basico de esta herramienta.

EL código de prueba que se utilizara en este ejemplo puede encontrarse [en este repositorio](#).

3.2. Antes de empezar

Cuando tengamos un programa que deseemos depurar, tenemos que tener en cuenta que tendremos que habilitar al menos uno o dos banderas (**flags**) adicionales a nuestro proceso de compilacion. Las banderas a agregar son:

- **-g**: Que permite que GCC agregue la informacion de depuración necesaria en nuestros archivos objetos (archivos **.o**) para que gdb pueda funcionar correctamente.
- **-O0**: Permite que se fuerze al compilador para que no realice ningun tipo de optimizacion sobre nuestro código. Esto nos dara mucha mas informacion que no estara obfuscada, y podremos depurar con una mayor facilidad.



Cuando tengamos un *Makefile* existente y ya hayamos corrido previamente una compilacion sin estas banderas, una vez que hayamos agregado estas tendremos que limpiar todo con *make clean*(limpiar todos los archivos con extension **.o**) y luego volver a compilar con las nuevas banderas.

-W and -Wall : muestra los warnings (no es mandatorio, pero ayuda a arreglar mejoras a nuestro código, en cuanto a normas de estandarizacion.)

En el caso del ejemplo sin **Makefile**, estaremos usando este simple ejemplo que calcula el inverso de un numero. Este código estara en un solo archivo llamado **main.c**

```
#include <stdio.h>

int sum=0,rem;

int reverse_function(int num){
    if(num){
        rem=num%10;
        sum=sum*10+rem;
        reverse_function(num/10);
    }
    else
        return sum;
    return sum;
}
```

```

}

int main(){
    int num,reverse_number;

    //User would input the number
    printf("\nEnter any number:");
    scanf("%d",&num);

    //Calling user defined function to perform reverse
    reverse_number=reverse_function(num);

    printf("Reverse of entered number is = %d\n", reverse_number);

    return 0;
}

```



Este ejemplo puede encontrarse, junto con el makefile, y todos los archivos necesarios en este [repositorio](#)

En este caso solo basta con agregar los flags a

```
gcc main.c -g -O0 -W -Wall
```



Podemos reemplazar el `-Wall` por `-Werror`, para que solo mencione los errores y no continúe la compilación, sin tener en cuenta los warnings.

Ahora, si tenemos un proyecto mas completo, en el que necesitamos que exista el `Makefile`, tendremos que agregar una regla para agregar estos flags y que pueda utilizarse el depurador.



Es útil tener otra regla, si se usa *Makefile*, para correr la compilación con las banderas de depuración y otra para la compilación normal, en especial, si usamos la bandera `-O0`, debido a que esta opción, desactiva la optimización del código, y permitiera que podamos correr la depuración sin mayores problemas.

3.3. Depurando en VScode

Una vez que tenemos disponible el proyecto, hay que compilar el proyecto con `make` y también correr la compilación, cuando se quiera con la directiva de "debug" `make debug`, la regla del makefile tiene que ser similar a la siguiente:

```

debug: CFLAGS+=-O0
debug: mkdir $(OFILES)
      $(CC) $(DEBUG_CFLAGS) $(OFILES) -o $(BIN)/$(TARGET)

```

```
~/utn/ssl/ssl-debugging-example$ make debug

mkdir -p ./bin
mkdir -p ./build
gcc -std=c11 -g -I./inc ./build/main.o -o ./bin/reverse_number
```

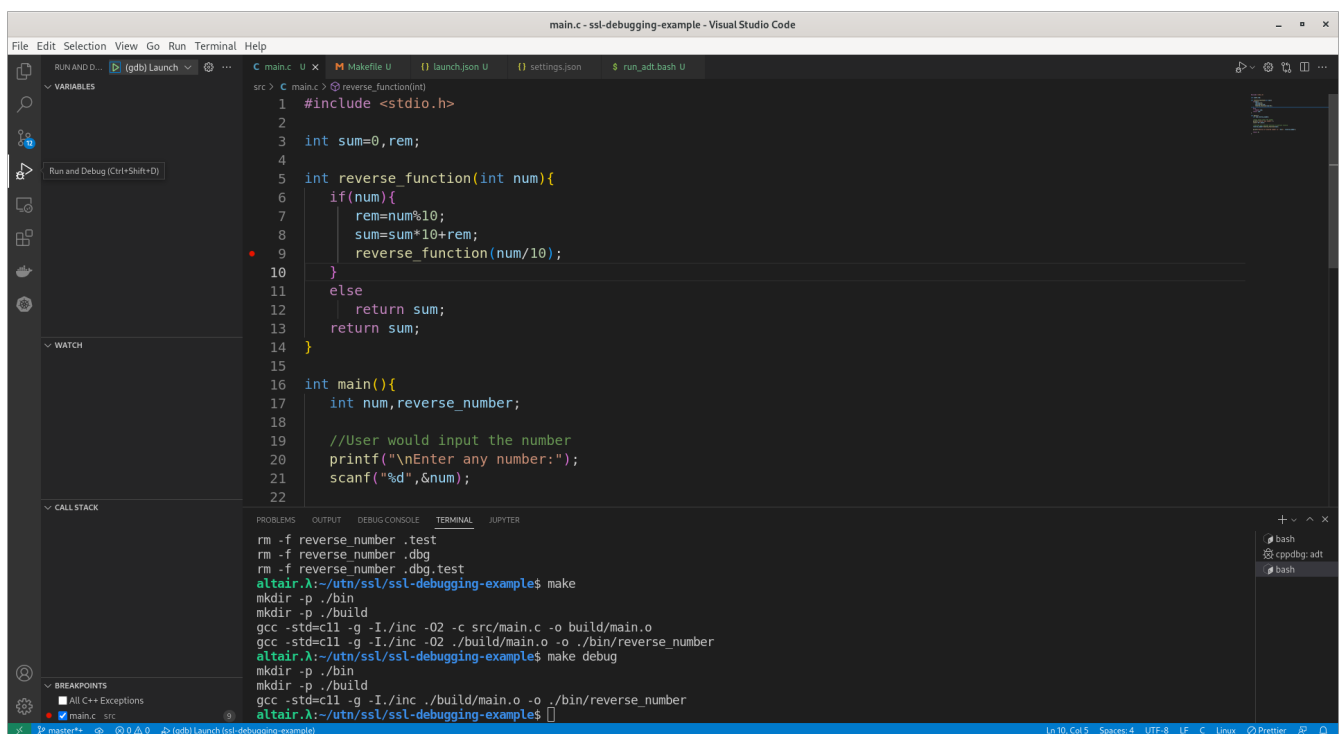
Una vez que compilamos la aplicación con la regla de makefile de debug, usando las banderas descritas en la sección anterior, estamos listos para depurar nuestra aplicación. solo resta verificar que en el archivo `launch.json`, que esta en la carpeta de `.vscode`, este el nombre del ejecutable correctamente escrito. Esto podremos encontrarlo en la sección `configurations/program`

```
"program": "${workspaceRoot}/bin/reverse_number"
```

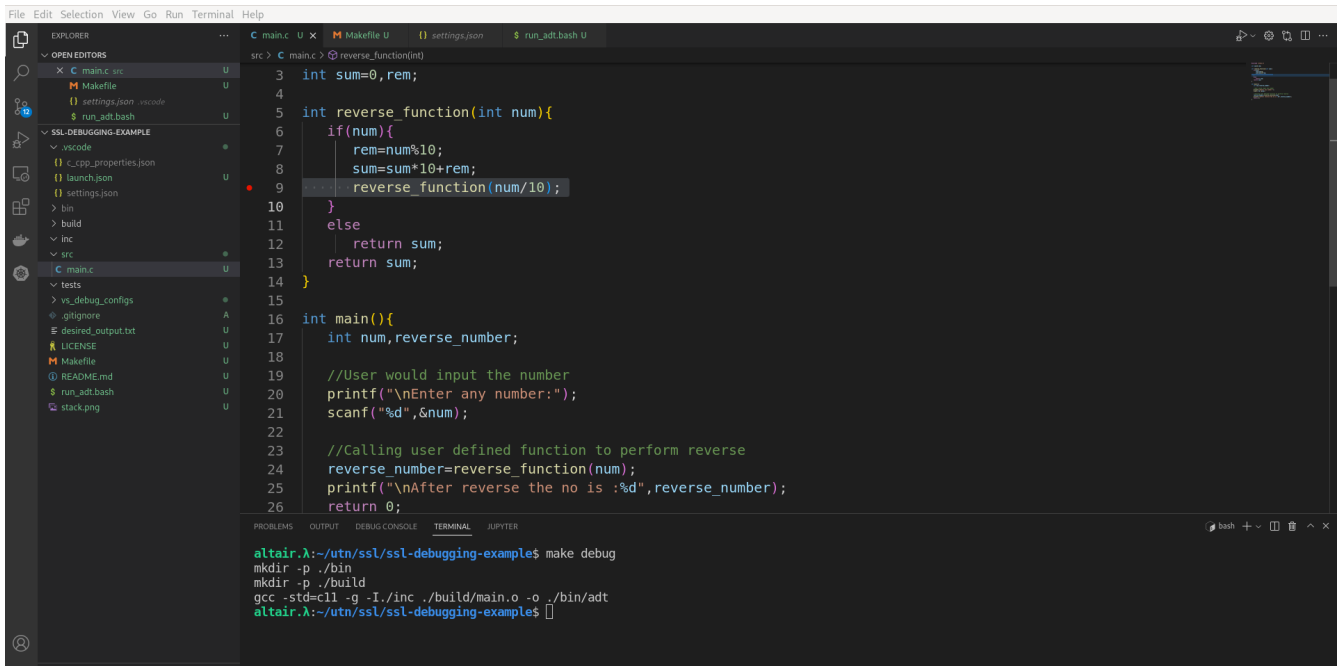


Siempre hay que cambiar este nombre, con el nombre del ejecutable que estemos generando, al compilar la aplicación. En caso contrario, el depurador no funcionara al no encontrar el archivo.

Una vez que estemos listos, podremos ver que tenemos una ventana para empezar a depurar el código.



Antes de correr el programa, bajo el modo de depuración, tendremos que setear los `breakpoints`. Estos son puntos, en donde queremos detener la aplicación, para saber



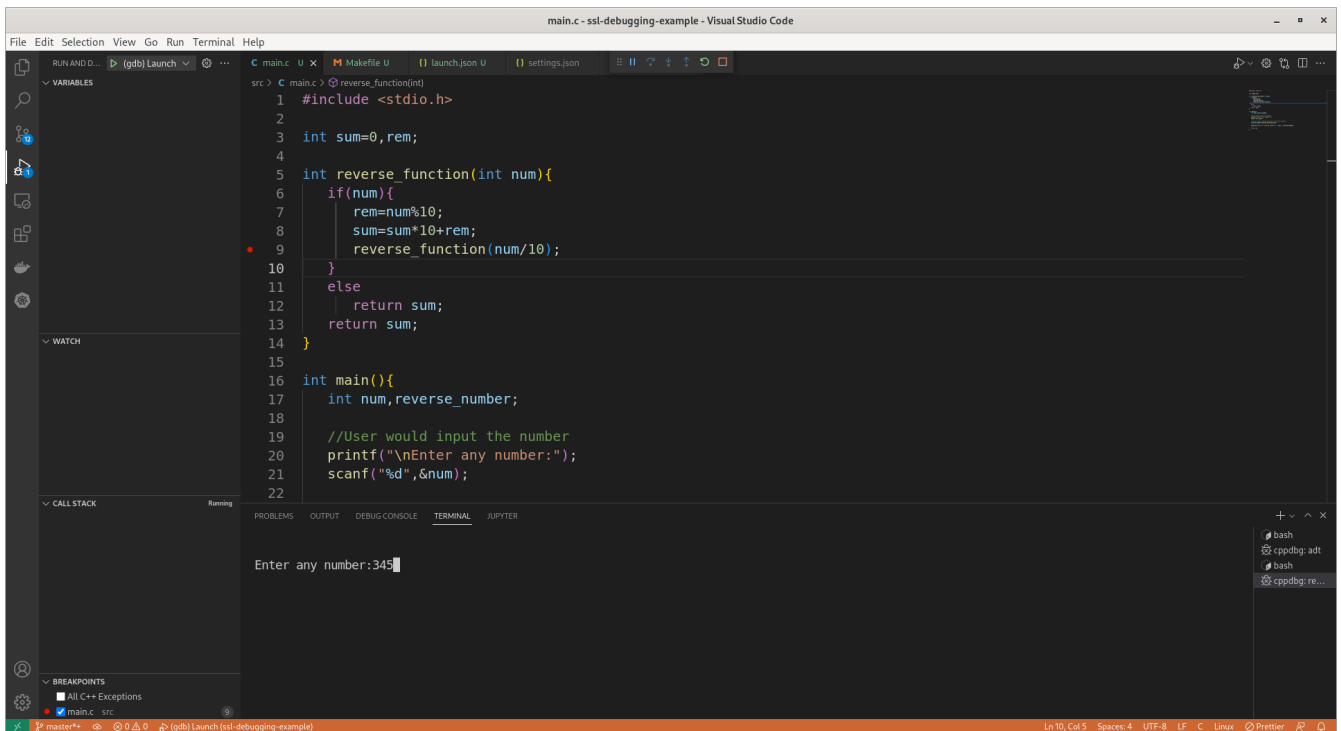
```
File Edit Selection View Go Run Terminal Help
C main.c U X M Makefile U I settings.json U $ run_adt.bash U
src > C main.c > reverse_function(int)
3 int sum=0,rem;
4
5 int reverse_function(int num){
6     if(num){
7         rem=num%10;
8         sum=sum*10+rem;
9         reverse_function(num/10);
10    }
11    else
12        return sum;
13    return sum;
14 }
15
16 int main(){
17     int num,reverse_number;
18
19     //User would input the number
20     printf("\nEnter any number:");
21     scanf("%d",&num);
22
23     //Calling user defined function to perform reverse
24     reverse_number=reverse_function(num);
25     printf("\nAfter reverse the no is :%d",reverse_number);
26     return 0;
27 }
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
altair.λ:~/utn/ssl/ssl-debugging-example$ make debug
mkdir -p ./bin
mkdir -p ./build
gcc -std=c11 -g -I./inc ./build/main.o -o ./bin/adt
altair.λ:~/utn/ssl/ssl-debugging-example$
```



Los breakpoints son un número de línea de interés en su programa. Cuando el depurador se está ejecutando, detiene la ejecución del programa en esta línea.

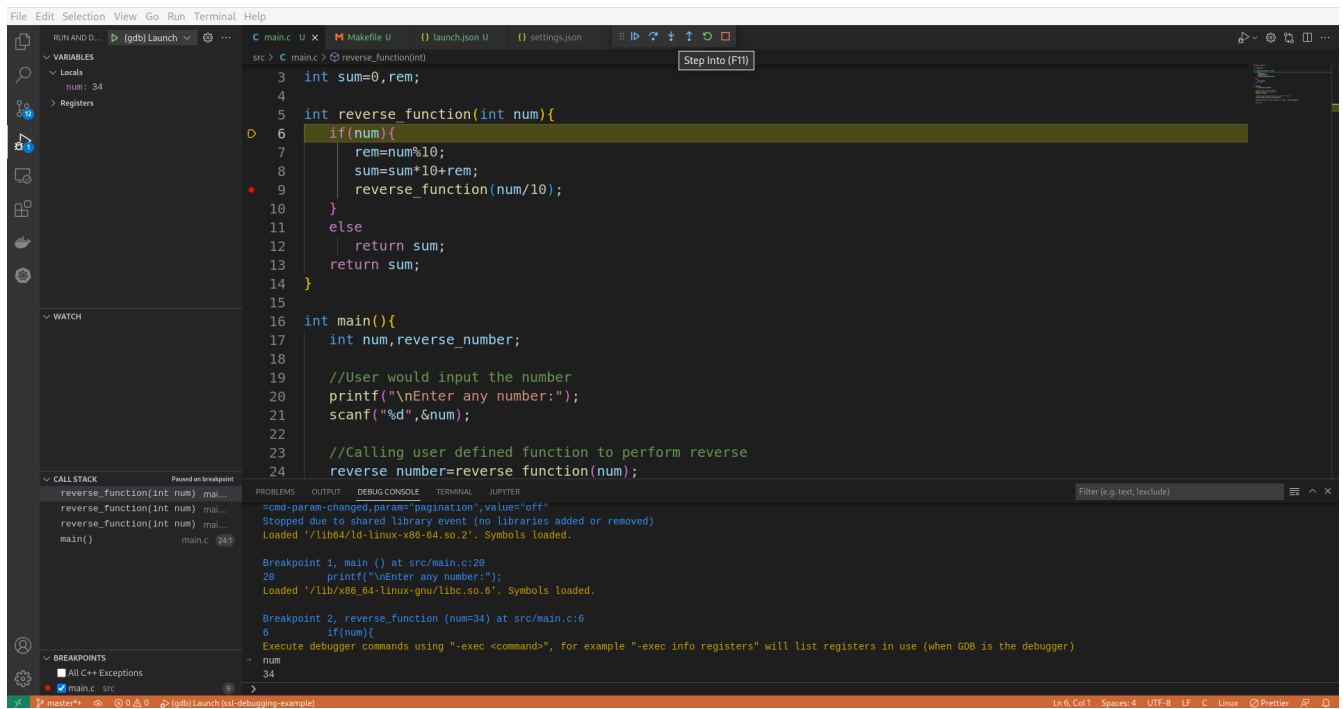
Una vez que hayamos definido los lugares, donde queremos que la aplicación se detenga, podremos iniciar la depuración.

En el ejemplo tendremos que utilizar el input para ingresar un numero, una vez que lo hagamos, se llamara a la función que calcula el numero con los digitos invertidos.



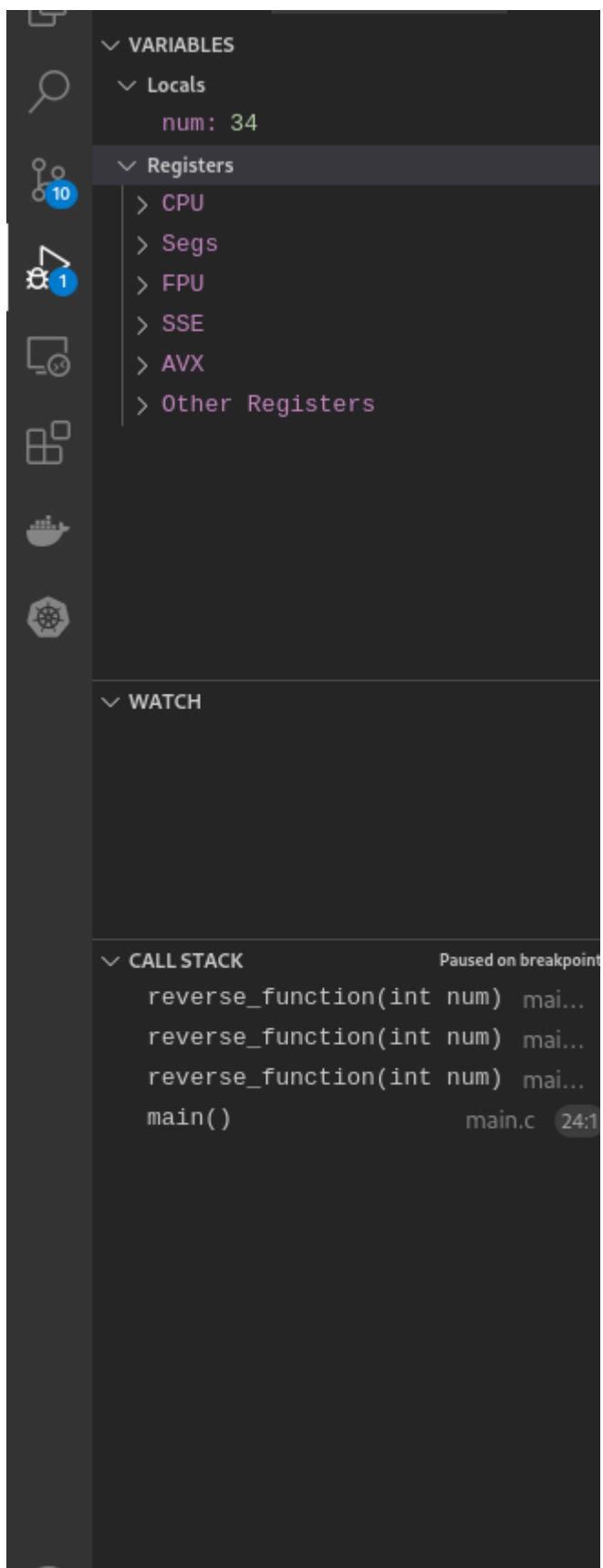
```
main.c - ssl-debugging-example - Visual Studio Code
File Edit Selection View Go Run Terminal Help
C main.c U X M Makefile U I launch.json U I settings.json U
src > C main.c > reverse_function(int)
1 #include <stdio.h>
2
3 int sum=0,rem;
4
5 int reverse_function(int num){
6     if(num){
7         rem=num%10;
8         sum=sum*10+rem;
9         reverse_function(num/10);
10    }
11    else
12        return sum;
13    return sum;
14 }
15
16 int main(){
17     int num,reverse_number;
18
19     //User would input the number
20     printf("\nEnter any number:");
21     scanf("%d",&num);
22 }
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
Enter any number:345
```

Una vez que se llegue al primer endpoint, la aplicación se detendra.

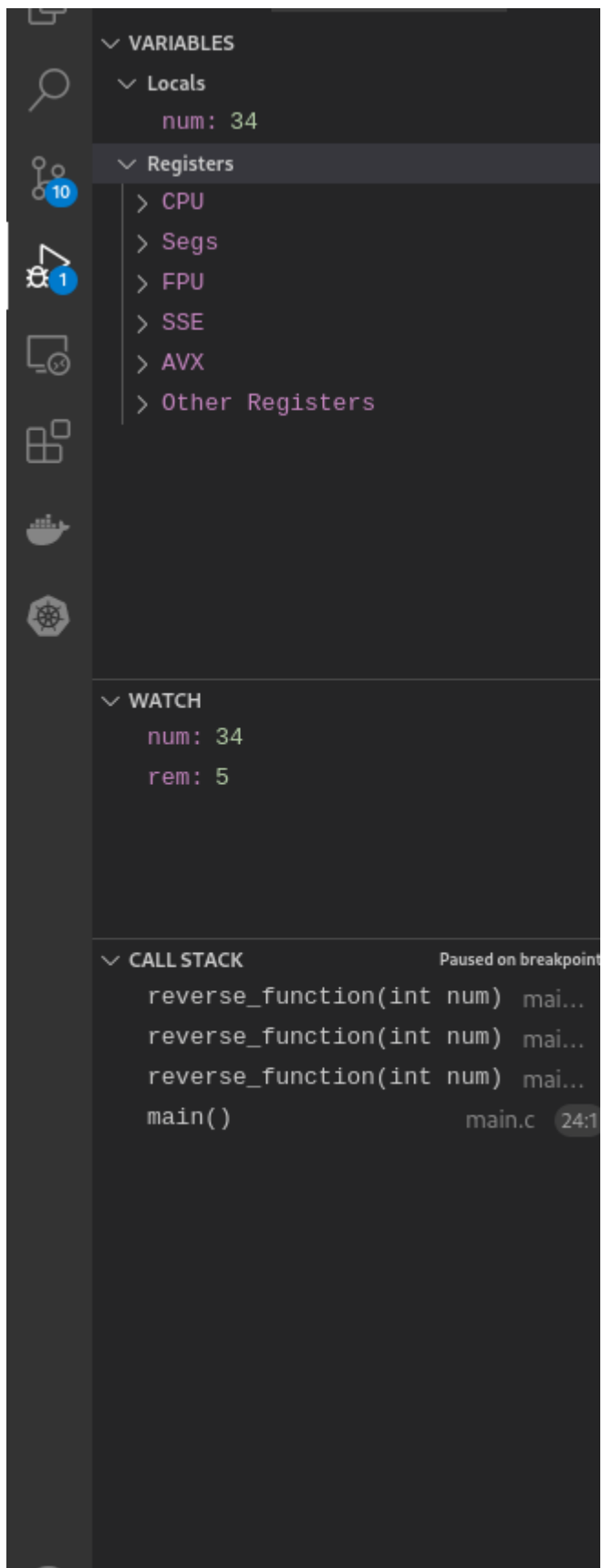


3.4. Secciones básicas del depurador.

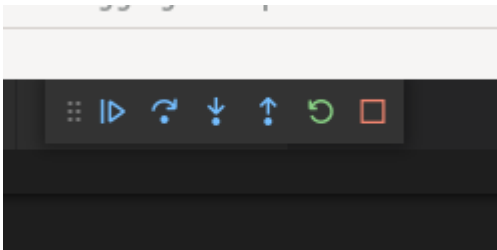
Podremos ver en la sección de la izquierda, las variables, los registros, y las llamadas al stack, y como fueron esas llamadas. Ademas tendremos una sección de **watch**, que nos permite agregar variables o condiciones que podremos ir viendo como cambian a lo largo del tiempo y la corridad de la aplicación en este modo.



En este ejemplo vamos a agregar las variables `rem` y `num`, en observacion (`watch`)



En la sección superior vamos a poder ver las diferentes acciones que nos ofrece el debugger:



Las referencias de los simbolos son los siguientes:

- **Continue**: una acción a realizar en el depurador que continuará la ejecución hasta que se alcance el siguiente punto de interrupción o se cierre el programa.
- **Step over**: una acción a realizar en el depurador que pasará por encima de una línea determinada. Si la línea contiene una función, la función se ejecutará y se devolverá el resultado sin depurar cada línea.
- **Step into**: Si la línea no contiene una función, se comporta igual que **Step over** ("pasar por alto"), pero si la contiene, el depurador ingresará a la función llamada y continuará la depuración línea por línea allí.
- **Step out**: una acción a realizar en el depurador que regresa a la línea donde se llamó a la función actual.
- **Restart**: una acción, que fuerza al depurador reiniciar la ejecución de la aplicación en este modo.
- **Stop**: Detiene la ejecución del depurador, y de la aplicación.

| Mostraremos una ejecución simple en este documento.

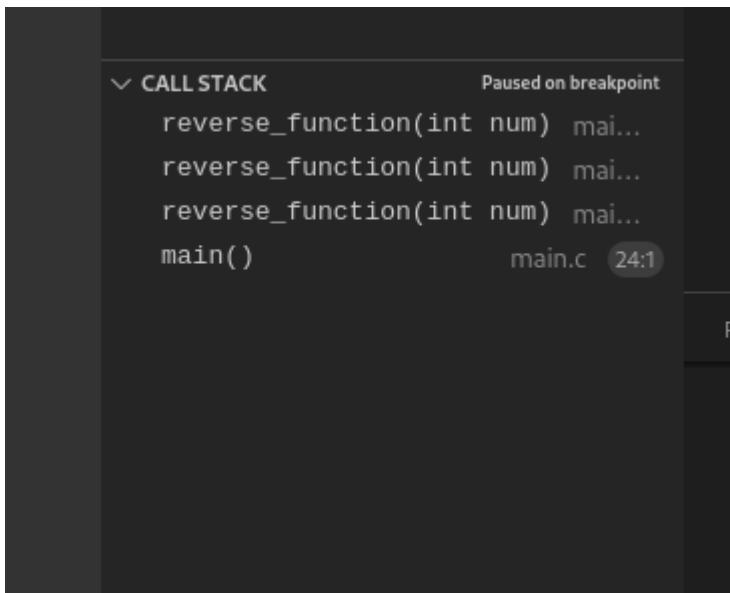
Tambien vamos a poder acceder, una vez que el depurador, para la ejecución, a una consola en donde podremos evaluar las variables y ejecutar funciones a fin de poder arreglar algun problema en nuestra aplicación o inspeccionar el estado de esta.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
=cmd-param-changed,param="pagination",value="off"
Stopped due to shared library event (no libraries added or removed)
Loaded '/lib64/ld-linux-x86-64.so.2'. Symbols loaded.

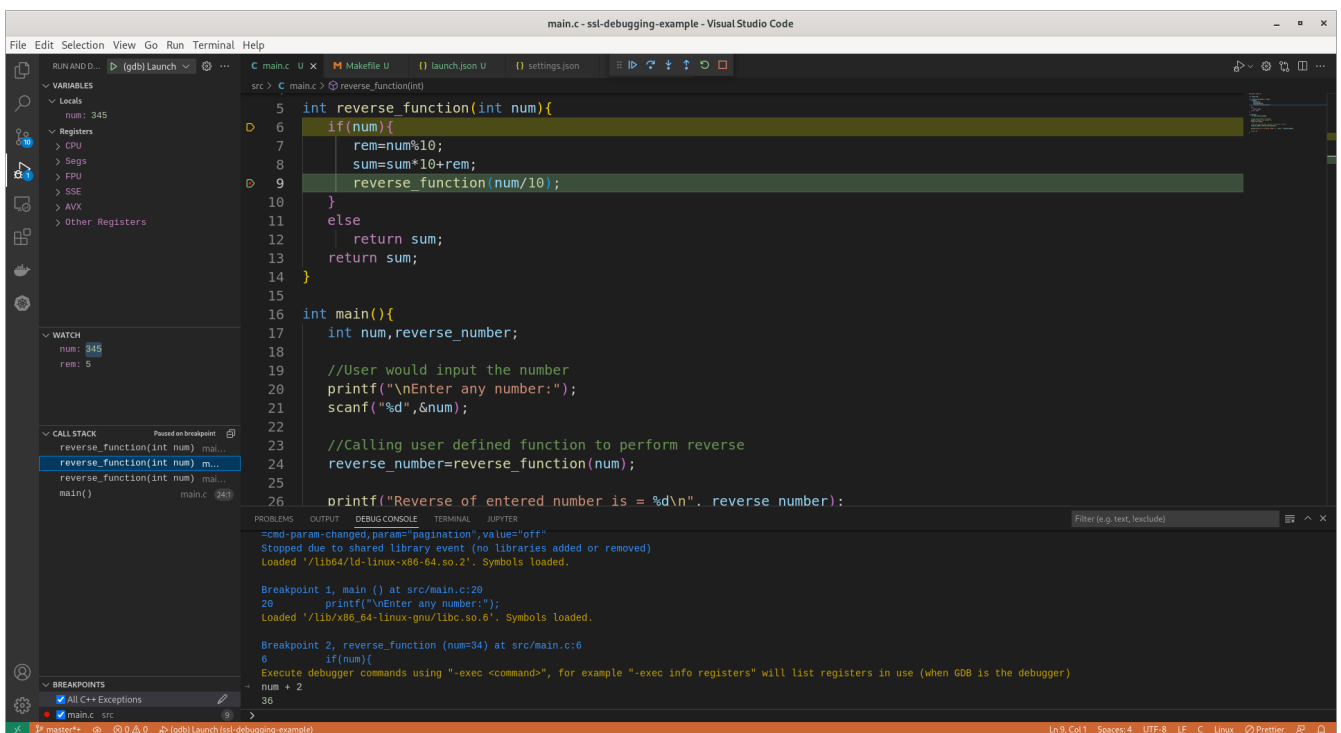
Breakpoint 1, main () at src/main.c:20
20  printf("\nEnter any number:");
Loaded '/lib/x86_64-linux-gnu/libc.so.6'. Symbols loaded.

Breakpoint 2, reverse_function (num=34) at src/main.c:6
6  if(num){
Execute debugger commands using "-exec <command>", for example "-exec info registers" will list registers in use (when GDB is the debugger)
num + 2
36
```

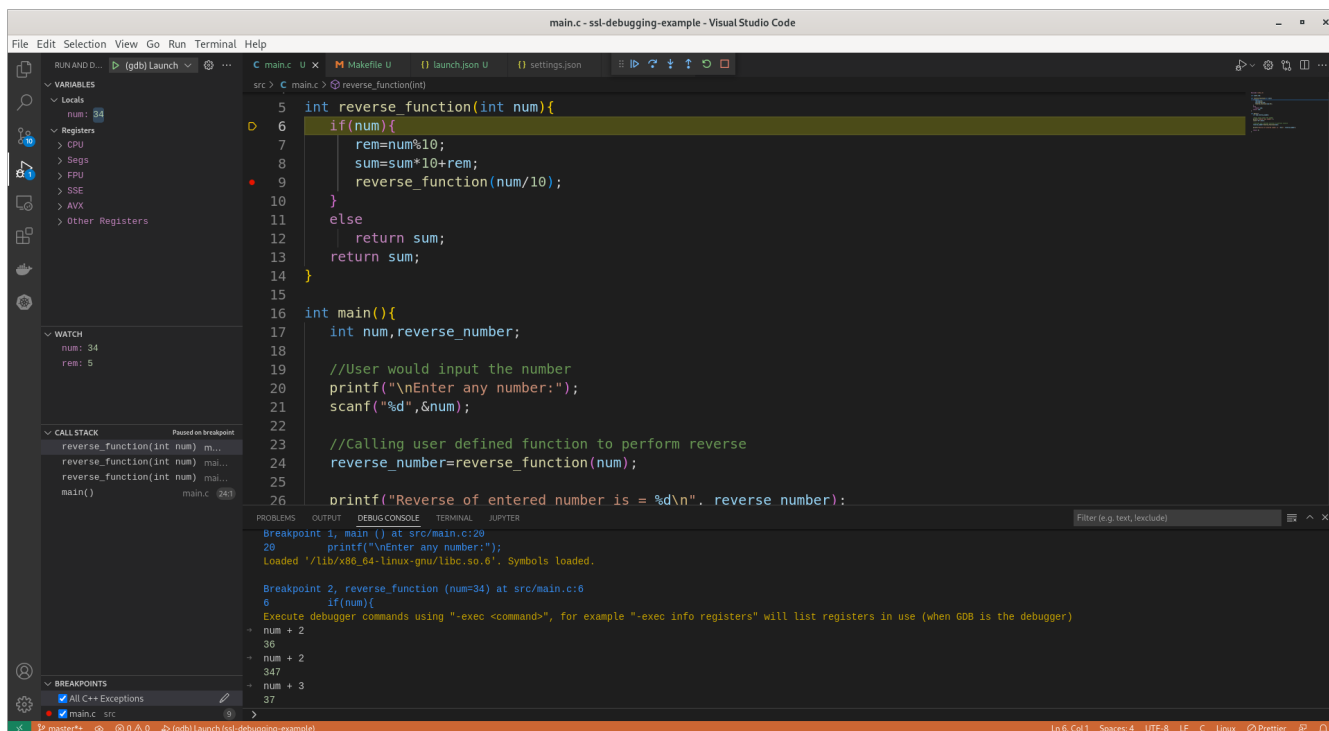
Sobre el **call stack** (Llamadas de la pila), podemos mencionar que, esta es una lista de funciones en el depurador que explica cómo llegó el programa a donde está actualmente. Piense en esto como un seguimiento de pila en vivo, sin excepción.



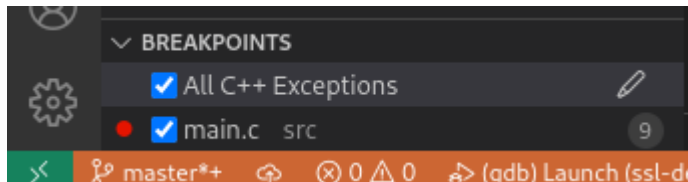
Podemos seleccionar cualquier punto de las llamadas hechas, y ver el estado en el que estaban las variables, y evaluar estas en la consola de depuración.



Podemos ver en esta otra imagen, como cambia el contexto, dependiendo de que llamada en la pila nos situemos.



Otra sección que no se menciona es la de los archivos que queremos depurar, podemos incluso seleccionar cualquier excepción de C que se genere, a fin de que se generen **breakpoints** cuando se lanza un error o excepción en la aplicación, si es que no sabemos el punto en el que se produce el error en nuestro programa.



3.5. Continuando con la ejecución

Haciendo un **Continue**, podemos ver que avanzamos en la depuración hasta llegar al proximo **breakpoint**.

The screenshot displays the Visual Studio Code interface for debugging a C program. The title bar indicates the file is `main.c - ssl-debugging-example - Visual Studio Code`.

Left Sidebar:

- VARIABLES:** Shows local variables like `num: 34`.
- Registers:** Lists CPU, Segs, FPU, SSE, AVX, and Other Registers.
- WATCH:** Shows watched variables like `num: 34` and `rem: 8`.
- CALL STACK:** Shows the call stack with `reverse_function` and `main` functions.
- BREAKPOINTS:** Shows breakpoints set for `All C++ Exceptions` and `main.c:src`.

Main Editor:

```

1  int main()
2  {
3      int num;
4      reverse_function(num);
5      int reverse_function(int num){
6          if(num){
7              rem=num%10;
8              sum=sum*10+rem;
9              reverse_function(num/10);
10         }
11         else
12             return sum;
13         return sum;
14     }
15
16     int main(){
17         int num,reverse_number;
18
19         //User would input the number
20         printf("\nEnter any number:");
21         scanf("%d",&num);
22
23         //Calling user defined function to perform reverse
24         reverse_number=reverse_function(num);
25
26         printf("Reverse of entered number is = %d\n". reverse_number);
27     }
28 }

```

Bottom Panel:

- PROBLEMS:** Empty.
- OUTPUT:** Empty.
- DEBUG CONSOLE:** Shows the execution flow:


```

Breakpoint 2, reverse_function (num=34) at src/main.c:6
6      if(num){
Execute debugger commands using "-exec <command>", for example "-exec info registers" will list registers in use (when GDB is the debugger)
+ num + 2
+ 36
+ num + 2
+ 347
+ num + 3
+ 37
Breakpoint 2, reverse_function (num=34) at src/main.c:9
9      reverse_function(num/10);

```
- TERMINAL:** Empty.
- JUPYTER:** Empty.

The status bar at the bottom shows the current file is `main.c` at line 9, column 1.

Despues podremos ir haciendo **Step into**, si es que queremos avanzar de a poco en la ejecución de la aplicación, viendo como cambia el estado de las variables.

The screenshot shows the Visual Studio Code interface with a C program for reversing a number. The code is in a file named `main.c`. The program defines a `reverse_function` that takes an integer `num` and returns its reverse. The `main` function prompts the user to enter a number and calls `reverse_function` to reverse it.

```

1  #include <stdio.h>
2
3  int sum=0,rem;
4
5  int reverse_function(int num){
6      if(num){
7          rem=num%10;
8          sum=sum*10+rem;
9          reverse_function(num/10);
10     }
11     else
12         return sum;
13     return sum;
14 }
15
16 int main(){
17     int num,reverse_number;
18
19     //User would input the number
20     printf("\nEnter any number:");
21     scanf("%d",#);
22 }

```

The terminal output shows the program running and waiting for input. The user has entered the number 345.

```

Enter any number:345

```

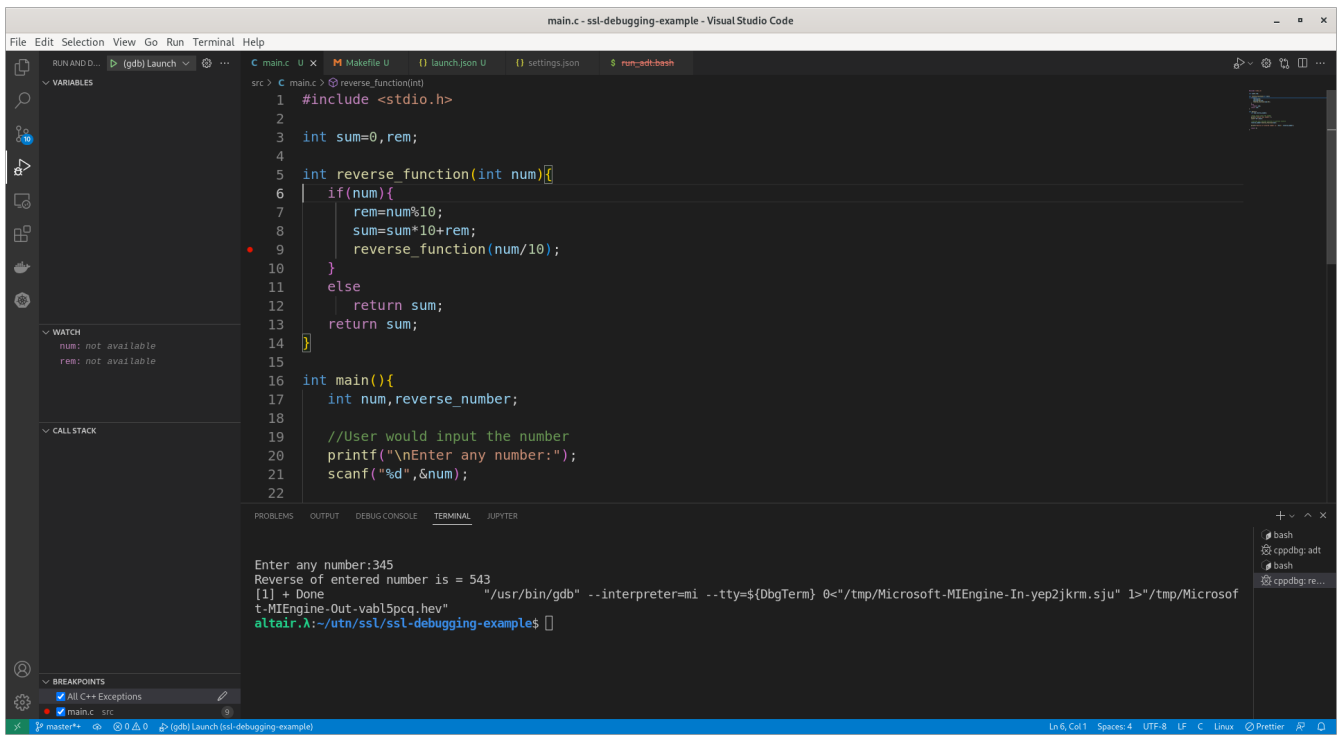
The left sidebar shows the Explorer, Search, and Run and Debug views. The bottom status bar shows the file path: `/home/.../main.c`.

Mediante la consola y viendo las variables, en que valores estan, podremos ver y tratar de inspeccionar o arreglar algun problema, si es que existe alguno.

Una vez que finalice la ejecución, se verá, que todos los datos que mostraba el depurador de nuestra aplicación, han desaparecido. Podremos despues volver a



Al hacer algun cambio de nuestro código, siempre hay que volver a compilar de nuevo, para que el depurador tome los cambios. El depurador no trabaja sobre nuestro código fuente, sino sobre el ejecutable compilado.



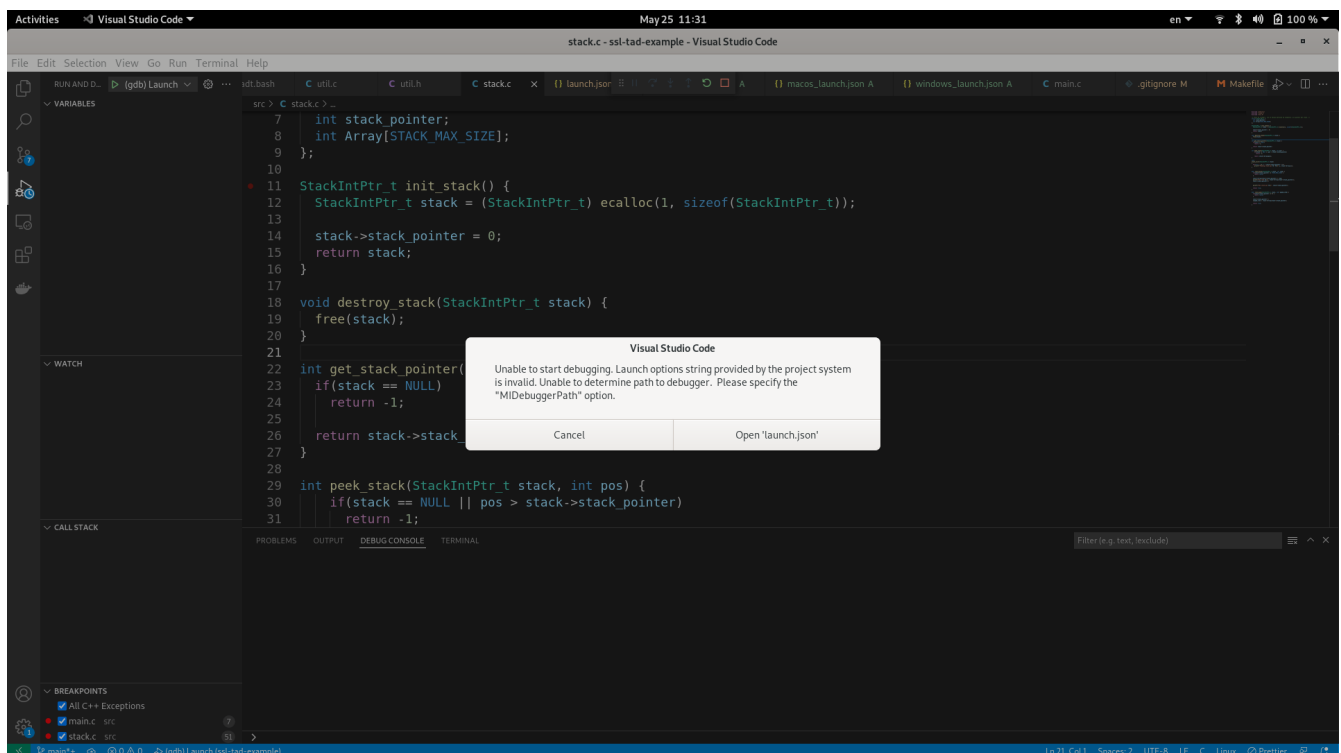
Part III: Extras

Chapter 4. FAQ

4.1. Agregue el flag **-g** a la configuracion de mi Makefile, pero aun no toma los breakpoints seteados en el editor de VSCode. Porque VSCode no toma los breakpoints?

Esto sucede generalmente o bien porque corrimos el **make** de nuevo o puede ser porque si no hicimos un **make clean**, para limpiar los **object files**. En este caso siempre que cambiamos un flag deberemos

4.2. Que pasa si me aparece un error como el siguiente > **Unable to determine path to debugger?**



Primero hay que ver si tenemos el depurador habilitado o instalado, en linux basta con hacer algo como:

```
altair.λ:~$ gdb
bash: gdb: command not found
```

en este caso **gdb** no esta instalado, entonces tendríamos que instalar gdb en este caso:

En Debian/Ubuntu basta con hacer un **apt-get install**, y en arch podemos usar en cambio **pacman -S**:

```
altair.λ:~$ sudo apt-get install gdb
```

Una vez instalado no deberiamos tener este problema presente, en caso de que estemos en windows, tendremos que verificar que el `$SOURCES` tiene seteado el path al depurador correctamente, si es que `Mingw` no lo agrego en la instalacion.

Chapter 5. Apendice

5.1. Configuraciones de ejemplo



Estas son configuraciones de prueba, y son el puntapie inicial y minimo para poder hacer funcionar el depurador.



Hay que siempre cambiar los argumentos `program` y `args`, con el ejecutable que necesitemos *"debuggear"* e inspeccionar, y con los argumentos que necesite el mismo.

5.1.1. Configuracion de ejemplo e inicial para Windows

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "(Windows) Launch",
      "type": "cppvsdbg",
      "request": "launch",
      "program": "${workspaceRoot}/main.exe",
      "args": ["4", "3", "2", "1"],
      "stopAtEntry": false,
      "cwd": "${workspaceRoot}",
      "environment": [],
      "externalConsole": false
    }
  ]
}
```

5.2. Configuracion de ejemplo e inicial para Linux

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "(gdb) Launch",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceRoot}/a.out",
      "args": ["4", "3", "2", "1"],
      "stopAtEntry": false,
      "cwd": "${workspaceRoot}",
      "environment": [],
      "externalConsole": false,
    }
  ]
}
```

```

    "MIMode": "gdb",
    "setupCommands": [
      {
        "description": "Enable pretty-printing for gdb",
        "text": "-enable-pretty-printing",
        "ignoreFailures": true
      }
    ]
  }
]
}

```

5.3. Configuración de ejemplo e inicial para MacOS

```

{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Launch Program(lldb)",
      "type": "cppdbg",
      "request": "launch",
      "program": "${workspaceRoot}/a.out",
      "args": ["4", "3", "2", "1"],
      "stopAtEntry": false,
      "cwd": "${workspaceRoot}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "lldb"
    }
  ]
}

```

Chapter 6. Changelog

6.1. Sobre el Changelog

En esta sección se describirá los distintos cambios que tiene dicha guía.

6.1.1. 1.0.2 (2022-08-04) - @bossiernesto

Documentation

- Publicación final del documento de depuración en C.
- Agregando imágenes y links al repositorio de ejemplo.

6.1.2. 1.0.1 (2022-05-25) - @bossiernesto

Documentation

- Agregando apéndice con configuración inicial para poder empezar a "*debuggear*" en distintos sistemas operativos.
- Publicación inicial de esta guía en github pages y formato PDF.

6.1.3. 1.0.0 (2022-05-22) - @bossiernesto

Primera versión de esta guía de depuración, con código de prueba y descripción de uso en Linux y Windows.

Documentation

- Agregando documentación en formato asciidoc