



Debug (Depuración)

Por Ariel Parra

[Γα=Ω5]

¿Qué es la Depuración?

La depuración es el proceso de encontrar y resolver errores (o "bugs") en un programa de software. Los errores pueden incluir:

- **Errores lógicos:** Resultados incorrectos debido a una lógica defectuosa.
- **Errores de tiempo de ejecución:** Fallos que ocurren durante la ejecución del programa.
- **Errores de sintaxis:** Problemas con la estructura del código que impiden la compilación.
- **Errores de memoria:** Problemas en la memoria del heap, como fugas de memoria o acceso a memoria inválida.
- **Otros errores:** Incluyen vulnerabilidades de seguridad y pérdida de datos.

Uso del `#define dbg(...)` de la plantilla

Una forma común de depurar es imprimiendo variables y valores con `cout`. Sin embargo, esto puede ser tedioso y poco eficiente. Nuestra macro `dbg()` simplifica este proceso al imprimir automáticamente el número de línea, el nombre de la variable y su valor, facilitando la identificación de problemas en el código.

```
#define dbg(...) cerr<<"LINE("<<__LINE__<<")->["<<#__VA_ARGS__<<"]: ["<<(__VA_ARGS__)<<"]\n";
```

```
int i=3;
while(--i){
    dbg(i);// imprime: `LINE(3)->[i]: [2]` y `LINE(3)->[i]: [1]`
}
int j = i;
dbg(j);// imprime: `LINE(6)->[j]: [0]`
```

Herramientas de GNU



AddressSanitizer (ASan)

AddressSanitizer es una herramienta para detectar errores de memoria, como desbordamientos de búfer, uso de memoria después de liberarla y accesos a memoria no válida. Está incluida en los compiladores gcc y g++, y se utiliza con el parámetro `-fsanitize=address`.

- **Compilación:** Añade el parámetro `-fsanitize=address`:

```
g++ -fsanitize=address -o my_program my_program.cpp
```

- **Ejecución:**

```
./my_program
```

- **Resultado en la Ejecución:**

```
==12345==ERROR: AddressSanitizer: use-after-free on address 0x0...
```

UndefinedBehaviorSanitizer (UBSan)

UndefinedBehaviorSanitizer detecta comportamientos indefinidos en el código, como divisiones por cero y desbordamientos de enteros. También está incluido en gcc y g++, y se utiliza con el parámetro `-fsanitize=undefined`.

- **Compilación:** Añade el parámetro `-fsanitize=undefined`:

```
g++ -fsanitize=undefined -o my_program my_program.cpp
```

- **Ejecución:**

```
./my_program
```

- **Resultado en la Ejecución:**

```
==12345==ERROR: UndefinedBehaviorSanitizer: division by zero on address 0x0...
```

Este mensaje indica un comportamiento indefinido, como una división por cero.

GNU Debugger (GDB)

GNU Debugger (GDB) es una herramienta de depuración para programas escritos en C, C++, y otros lenguajes. Permite a los desarrolladores observar el comportamiento del programa en tiempo real y corregir errores.

Características Principales

- **Puntos de Interrupción:** Permite detener la ejecución del programa en líneas específicas del código para examinar el estado del programa.
- **Inspección de Variables:** Permite ver y modificar el valor de variables durante la ejecución.
- **Seguimiento de Ejecución:** Permite avanzar línea por línea o función por función para observar cómo se ejecuta el código.
- **Control del Programa:** Permite iniciar, detener y continuar la ejecución del programa bajo depuración.

Uso Básico de GDB

1. **Compilación con Información de Depuración:** Añade la opción `-g` para incluir información de depuración:

```
g++ -g -o my_program my_program.cpp
```

2. **Iniciar GDB:**

```
gdb my_program
```

3. **Comandos Básicos:**

- **Iniciar el Programa:** `run`
- **Establecer un Punto de Interrupción:** `break <número_de_línea>` O `break <nombre_de_función>`
- **Continuar la Ejecución:** `continue`
- **Paso a Paso:** `next` (para avanzar una línea) o `step` (para entrar en funciones)
- **Ver el Valor de una Variable:** `print <nombre_variable>`
- **Salir de GDB:** `quit`

Ejemplo de uso de GDB:

```
gdb my_program
(gdb) break main
Breakpoint 1 at 0x4006d6: file my_program.cpp, line 5.
(gdb) run
Starting program: /path/to/my_program
Breakpoint 1, main () at my_program.cpp:5
5      int a = 10;
(gdb) print a
$1 = 10
(gdb) continue
```

- **Explicación:**

- **break main**: Establece un punto de interrupción en la función **main**.
- **run**: Inicia la ejecución del programa hasta el primer punto de interrupción.
- **print a**: Muestra el valor de la variable **a**.

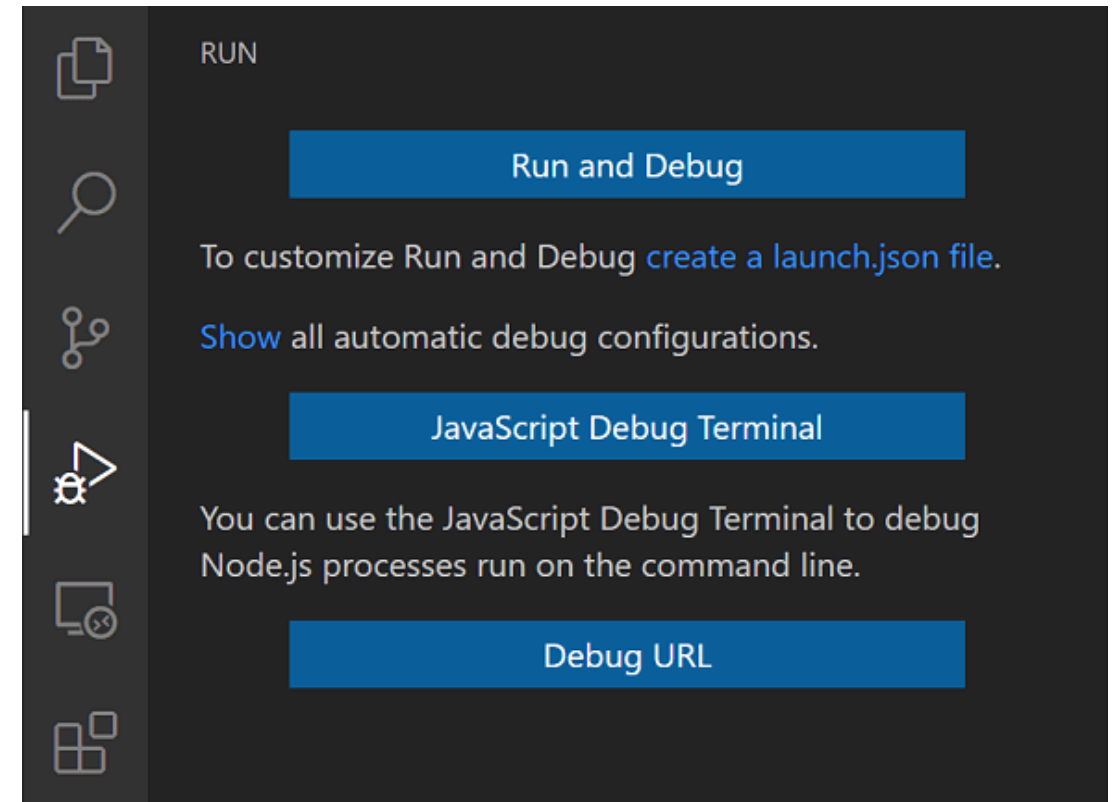
Depuración en VScode

1. Instalar Extensiones:

 C/C++[↑] y  Error lens[↑]

2. Configurar el Lanzador de Depuración

- Haz clic en el ícono de depuración en la barra lateral izquierda o con el atajo `Ctrl+Shift+D`.
- Haz clic en el enlace "Run and Debug" o en el engranaje para crear un archivo de configuración.
- Selecciona la configuración "C++ (GDB/LLDB)"



3. Añadir Puntos de Interrupción (breakpoints)

- **Establecer un Punto de Interrupción:** Haz clic en el margen izquierdo junto a la línea de código donde deseas detener la ejecución. Aparecerá un círculo rojo, indicando que se ha establecido un punto de interrupción.

```
378
379 →   return config;
380 }
381
382 ∨ function loadJSON(folder: vscode.WorkspaceFolder | undefined, file: string): any {
383 ∨ →   if (folder) {
384 ∨ →     try {
385 →       const path = join(folder.uri.fsPath, file);
386 →       const content = fs.readFileSync(path, 'utf8');
387 →       return JSON.parse(content);
388 ∨ →     } catch (error) {
389 →       // silently ignore
390 →     }
391 →   }
```



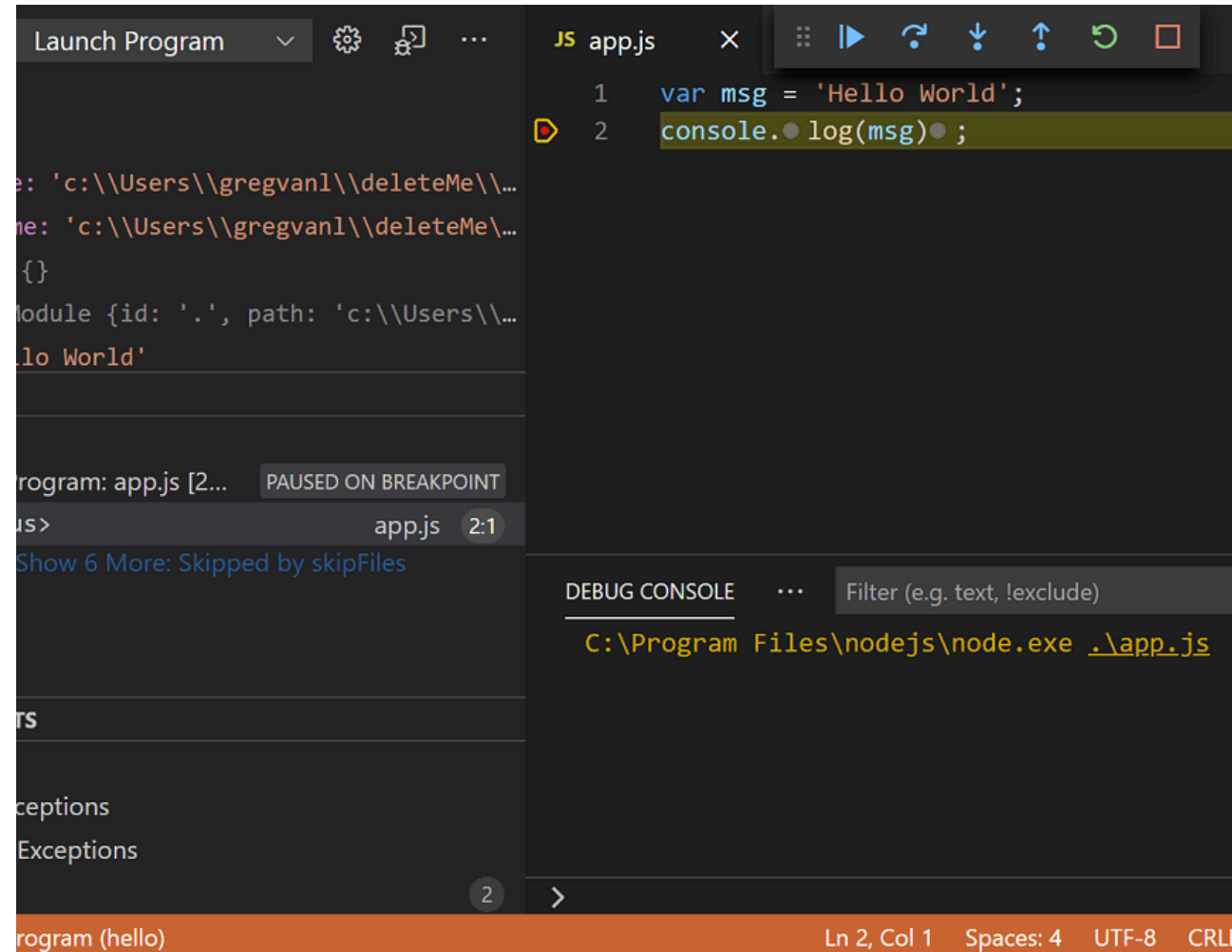
4. Iniciar la Depuración

- **Ejecutar el Depurador:**

- Haz clic en el ícono de "Play" o selecciona "Start Debugging" en el menú "Run".
- El programa se ejecutará hasta el primer punto de interrupción.

- **Control de Ejecución:**

- **Continuar (F5):** Continúa la ejecución hasta el siguiente punto de interrupción.
- **Paso a Paso (F10):** Avanza una línea a la vez, sin entrar en funciones.
- **Entrar en Función (F11):** Avanza una línea, entrando en la función si se llama a una.
- **Salir de Función (Shift+F11):** Continúa la ejecución hasta que la función actual termine.



```
Launch Program [v] [gear] [bug] [dots] JS app.js [x] [play] [refresh] [down] [up] [refresh] [stop]
```

```
1 var msg = 'Hello World';  
2 console.log(msg);
```

```
Program: app.js [2... PAUSED ON BREAKPOINT  
JS> app.js 2:1  
Show 6 More: Skipped by skipFiles
```

```
DEBUG CONSOLE [dots] Filter (e.g. text, !exclude)  
C:\Program Files\nodejs\node.exe .\app.js
```

```
Exceptions  
Exceptions
```

```
Program (hello) Ln 2, Col 1 Spaces: 4 UTF-8 CRLF
```

5. Inspeccionar y Modificar Variables

```
▲ VARIABLES
  ▲ Local
    ▲ args Array[5]
      length 5
      ▶ 0 Object
      ▶ 1 function require(path) {
      ▶ 2 Module
      3 "c:\w\hot-towel\src\server\app.js"
      4 "c:\w\hot-towel\src\server"
    ▲ compiledWrapper function (exports, require, module, __file...
      arguments undefined
      caller undefined
```

- **Ver Variables:**
 - Las variables locales y globales se mostrarán en la ventana de "Variables" durante la depuración.
- **Modificar Valores:**
 - Puedes modificar los valores de las variables durante la ejecución para probar diferentes escenarios.
- **Ver Pila (stack) de Llamadas:**
 - La "Call Stack" muestra la secuencia de llamadas de funciones hasta el punto de interrupción actual, permitiendo entender cómo se llegó a ese punto.

Actividad en clase

Depurar en VScode este código:

```
#include <bits/stdc++>
using namespace std;
int main() {
    int arr[] = { -12, -1234, -45, -67, -1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int res = 0;
    for (int i = 0; i < n; i++) {
        if (arr[i] > res)
            res = arr[i];
    }
    cout << "Maximum element of array: " << res;
}
```

Valgrind y Dr. Memory



Valgrind

Es la herramienta para la detección de errores más conocida en la industria, detecta errores tanto de memoria, como accesos inválidos y fugas de memoria. Es usado principalmente en sistemas Linux.

- **Compilación y Ejecución:**

- i. **Compilar el Programa:** Asegúrate de compilar con información de depuración:

```
g++ -g -o my_program my_program.cpp
```

- ii. **Ejecutar Valgrind:**

```
valgrind --leak-check=full ./my_program
```

- **Resultado en la Ejecución:**

- Valgrind imprimirá un informe detallado sobre errores de memoria, incluyendo accesos inválidos y fugas.

Dr. Memory

Es una herramienta para la detección de errores de memoria similar a Valgrind, como desbordamientos de búfer y fugas. Pero es más rápido en comparación con Valgrind y a diferencia de Valgrind, si esta disponible tanto para Windows, así como para Linux.

- **Uso:**

- i. **Compilar el Programa:** Asegúrate de compilar con información de depuración:

```
g++ -g -o my_program my_program.cpp
```

- ii. **Ejecutar Dr. Memory:**

```
drmemory -- ./my_program
```

- **Resultado en la Ejecución:**

- Dr. Memory proporcionará informes sobre errores de memoria, destacando problemas como accesos a memoria no válida y fugas.

Referencias

- angelbeats. (2020). *Macros for debugging*. Recuperado de <https://codeforces.com/blog/entry/85544> ↑
- Cutler, B. *Memory Debugging*. Recuperado de https://www.cs.rpi.edu/academics/courses/fall19/csci1200/memory_debugging.php ↑
- Dr. Memory. (s.f.). *Running Dr. Memory* https://drmemory.org/page_running.html ↑
- Divesh, K. (2024). *How To Debug Your Code | For Beginners* Recuperado de <https://www.geeksforgeeks.org/how-to-debug-your-code-for-beginners/> ↑
- GeeksforGeeks. (2024). *Segmentation Fault in C++*. Recuperado de <https://www.geeksforgeeks.org/segmentation-fault-c> ↑
- Gokhale, S. (2019). *Debugging in C++*. Recuperado de <https://codeforces.com/blog/entry/65543> ↑
- IBM. (s.f.). *What is debugging?*. Recuperado de <https://www.ibm.com/topics/debugging> ↑
- Low Level Learning. (2021). *GDB is REALLY easy! Find Bugs in Your Code with Only A Few Commands* [video]. Recuperado de https://youtu.be/Dq8l1_-QgAc?si=nFKzwCVWaLsfw43S ↑

- Low Level Learning. (2022). *you need to stop using print debugging (do THIS instead)* [video]. Recuperado de <https://youtu.be/3T3ZDquDDVg?si=OYcMOH09ePz9A7IK> ↑
- Mike Shah. (2021). *Using Valgrind and GDB together to fix a segfault and memory leak* [video]. Recuperado de https://youtu.be/8JEEYwdrexc?si=xsrKpm27iqv_QEc7 ↑
- Newhall, T. (2014). *Using valgrind.* Recuperado de <https://www.cs.swarthmore.edu/~newhall/unixhelp/purify.html> ↑
- ProgrammingKnowledge. (2023). *Debugging C Program with Visual Studio Code (VSCode)* [video]. Recuperado de <https://youtu.be/NJYcRcqPyOw?si=-uaKnT8a2LAqIEnh> ↑
- Qi, B. & Shrivastava, S. (s.f.). *Debugging C++*. Recuperado de <https://usaco.guide/general/debugging-cpp?lang=cpp> ↑
- The Builder. (2022). *The C++ memory leak detector no one told me about | address sanitizer* [video]. Recuperado de https://youtu.be/Ce6xF8ByOKY?si=-h_yiGI-JEV1Rx0g ↑

- The Chernob. (2017). *How to DEBUG C++ in VISUAL STUDIO* [video]. Recuperado de <https://youtu.be/0ebzPwixrJA?si=BJH7wDTq4cJGcK0k> ↗
- Valgrind. (s.f.). *The Valgrind Quick Start Guide*. Recuperado de <https://valgrind.org/docs/manual/quick-start.html> ↗
- Visual Studio Code. (2021). *Debug a C++ project in VS Code* [video]. Recuperado de <https://youtu.be/G9gnSGKYlg4?si=M0wuJMwbv33YiP48> ↗
- Visual Studio Code. (2022). *Debug C++ in Visual Studio Code*. Recuperado de <https://code.visualstudio.com/docs/cpp/cpp-debug> ↗
- Visual Studio Code. (2024). *Debugging* . Recuperado de <https://code.visualstudio.com/Docs/editor/debugging> ↗