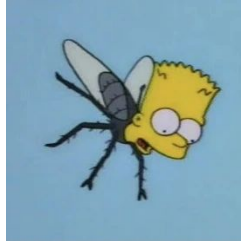


# Debugging

---



# ¿ Qué es un bug ?



Un bug es un error en nuestro código que provoca comportamientos indeseados.

Generalmente tienen dos causas:

- Error humano
- Factores externos:
  - Diferencias entre SO.
  - Comportamiento no determinado: Es una situación en la que el estándar del lenguaje **no define qué debería pasar**, por lo tanto, **el compilador puede hacer cualquier cosa**.
    - Acceder a memoria no inicializada.
    - Acceder fuera del rango de vectores.
    - Dividir por cero.

# Tipos de errores

- Errores de sintaxis
  - Ejemplo: falta de punto y coma, paréntesis mal cerrados
- Errores de compilación
  - Tipos de datos incompatibles, funciones no declaradas
- Errores de lógica
  - El programa compila, pero no hace lo correcto (p. ej., usar `<=` en lugar de `<`)
- Errores de ejecución (runtime)
  - División por cero, acceso a memoria inválida, segmentation fault



# Debugging

El proceso de debugging (también llamado depurar o debuggear) es el acto de eliminar bugs de nuestro código.

Hay dos formas de debuggear nuestro programa:

- Usando prints:
  - Es la forma más simple, seguro es la que usaron hasta ahora.
  - Tiene varios problemas. Es muy limitado y nos obliga a cambiar nuestro código.
- Usando un debugger:
  - Son herramientas dedicadas y construidas específicamente para debuggear, en clase usaremos una llamada GDB.
  - Permiten hacer mucho más que imprimir variables. Además, no hace falta editar el código.





**GDB**



# ¿ Qué es GDB ?

GDB es una herramienta esencial para depurar programas en C.

Permite:

- Detener la ejecución en puntos específicos (breakpoints).
- Ver el contenido de variables.
- Analizar la pila de llamadas.
- Observar cambios en variables.

# ¿ Cómo se usa ?

Primero debemos compilar nuestro programa y agregar el flag -g:

***gcc -g archivo.c -o programa***

*“To tell GCC to emit extra information for use by a debugger, in almost all cases you need only to add -g to your other options.”*

Luego ejecutamos nuestro programa anteponiendo el comando “gdb”:

***gdb ./programa***

# Comandos...

<b>break</b> <nombre_funcion>	# Coloca break point. O break <arch>:<linea>
<b>run</b>	# Ejecuta el programa hasta encontrar un break
<b>start</b>	# Ejecuta el programa deteniéndose en el inicio
<b>next</b>	# Ejecuta la siguiente línea
<b>step</b>	# Ingresa a una función
<b>print</b> <var>	# Muestra el valor de la variable
<b>backtrace</b>	# Muestra la pila de llamadas
<b>watch</b> <var>	# Vigila un cambio en una variable
<b>display</b> <var>	# Muestra el valor de la variable por cada next
<b>continue</b>	# Sigue la ejecución
<b>finish</b>	# Termina la función actual
<b>quit/exit</b>	# Sale de GDB

# TUI (Text User Interface)

Es una interfaz de usuario textual que te permite ver el código fuente mientras debugueás, todo desde la consola.

Op1:

Ejecutar ***“gdb ./programa”***

Luego ***“Ctrl + x”*** y luego ***“A”***

Op2:

***gdb -tui ./programa***



# Alternativa VSCode



# clang + lldb + CodeLLDB

clang -> otro compilador que compila código en C.

lldb -> debugger de clang.

CodeLLDB -> extensión de VSCode que integra lldb.

# Instalación: clang + lldb + CodeLLDB

Desde una terminal:

```
sudo apt install clang lldb
```

CodeLLDB

Buscar e instalar desde la solapa (la 4ta) de extensiones de VSCode.



## CodeLLDB

Vadim Chugunov



9.118.281

★★★★★ (93)

A native debugger powered by LLDB. Debug C++, Rust and other compiled languages.

Actualización a v1.11.5

Deshabilitar ▼

Desinstalar ▼

☐ Actualización automática

# Configuración: clang + lldb + CodeLLDB

En el directorio/carpeta donde tendremos nuestro archivo .c que contenga el main de nuestro programa, crear una carpeta con nombre **'vscode'** (el punto '.' es importante) en la cual crearemos dos archivos de extensión **.json**

*launch.json* -> configuración del debugger

*task.json* -> configuración de la compilación

# Configuración: clang + lldb + CodeLLDB

```
launch.json ×
.vscode > {} launch.json > ...
1  {
2    "version": "0.2.0",
3    "configurations": [
4      {
5        "type": "lldb",
6        "request": "launch",
7        "name": "configuracion de bernardo",
8        "program": "${workspaceFolder}/<nombre del ejecutable>",
9        "args": [],
10       "cwd": "${workspaceFolder}",
11       "preLaunchTask": "build",
12       "stopOnEntry": false
13     ]
14   }
15 }

tasks.json ×
.vscode > {} tasks.json > ...
1  {
2    "version": "2.0.0",
3    "tasks": [
4      {
5        "label": "build",
6        "type": "shell",
7        "command": "clang",
8        "args": [
9          "-g",
10         "*.c",
11         "-o",
12         "${workspaceFolder}/<nombre del ejecutable>",
13         "-std=c99",
14         "-Wall",
15         "-Wconversion",
16         "-Werror",
17         "-lm"
18       ],
19       "group": "build"
20     },
21   ]
22 }
```

**¡Importante!: reemplazar <nombre del ejecutable> por el nombre que le quieran poner al ejecutable que sea crea al compilar**