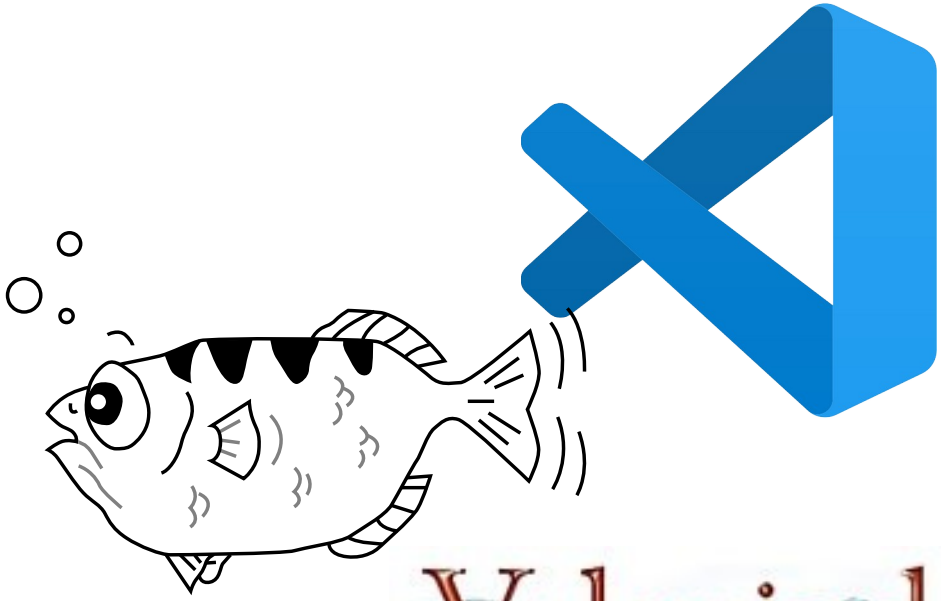
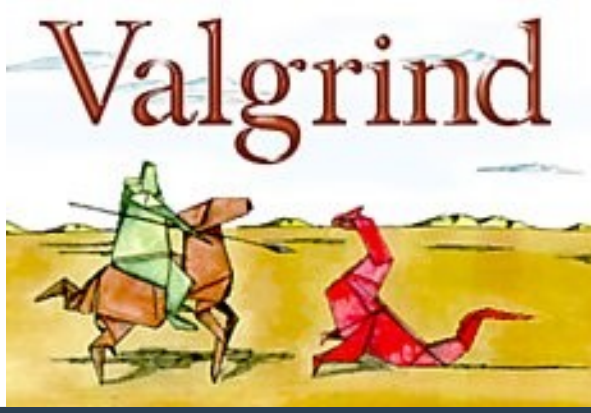
Depurando programas





Física Computacional (curso 2020/2021)
Dpto. de Electromagnetismo y Física de la Materia, UGR

Depuración o “*debugging*”

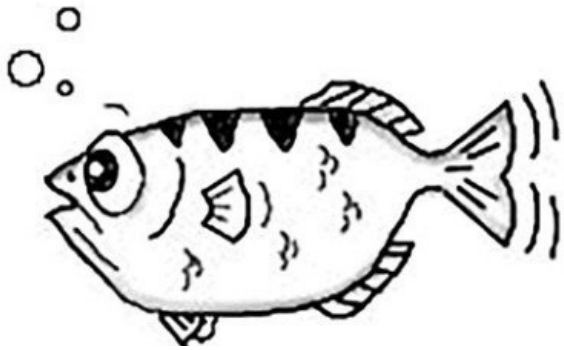
- Que un programa compile no quiere decir que funcione correctamente.
- Al proceso de **identificación** y **corrección** de errores (*bugs*) se le llama **depuración** o ***debugging***.
- Lleva mucho tiempo (**50% o más**)
  - Es importante hacerlo de forma eficiente
- Hay programas que facilitan esta tarea: **depuradores o *debuggers***

Depuradores

- Sirven para **analizar** el funcionamiento del programa a lo largo de su ejecución. Permiten:
  - Observar cambia el valor de las variables a lo largo del programa.
  - Añadir puntos de interrupción (***breakpoints***) donde el programa para para analizar su estado
  - Ejecutar el programa línea por línea

Depurador C/C++: GDB

- *Es el depurador por excelencia de C/C++ en Linux*
- *Funciona a través de la línea de comandos*
- *Usado por muchos entornos de programación como Code::Blocks o Eclipse.*



GDB
The GNU Project
Debugger

Curiosidad: el logo de GDB es un pez arquero, que caza bichos (b*ugs*) disparando chorros de agua

¿Cómo usar GDB?

1) Compilamos el programa con la opción -ggdb y forzando que no haga optimizaciones con -O0:

```
gcc programa.c -ggdb -O0 -o programa.exe
```

2) Abrimos el programa con GDB:

```
gdb programa.exe
```

3) Algunos comandos dentro de GDB:

- `break 9`: añade un breakpoint en la línea 9.
- `break main`: breakpoint al principio de la función main
- `step`: ejecuta la siguiente línea (entra dentro de las funciones)
- `next`: ejecuta la siguiente línea (no entra en las funciones)
- `print var`: muestra el valor de la variable var
- `start`: comienza a ejecutar el programa
- `continue`: continúa hasta el siguiente breakpoint

Más info:
https://www.tutorialspoint.com/gnu_debugger/gdb_quick_guide.htm

Entorno gráfico: Visual Studio Code

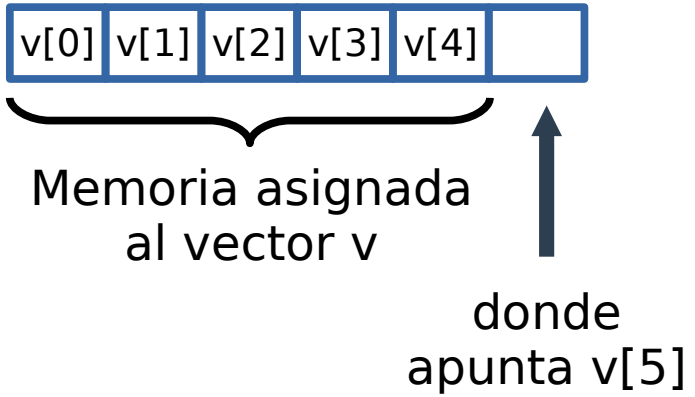
- Por comodidad vamos a depurar con el entorno gráfico **Visual Studio Code**, que usa internamente GDB.
- Es un entorno *open source* creado por Microsoft.
  - Permite programar en muchos lenguajes
  - Dispone de extensiones para añadir funcionalidades

- Enlace de descarga: https://code.visualstudio.com/download
- Guía rápida de depuración con Visual Studio Code: https://code.visualstudio.com/docs/editor/debugging

Segmentation faults (violación del segmento)

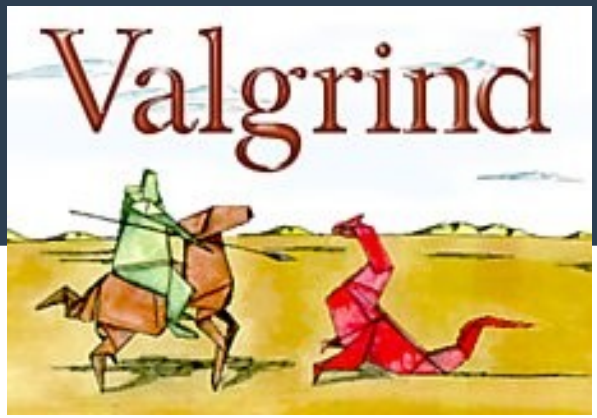
- Tipo de error que ocurre cuando un programa intenta acceder a una **región de la memoria que no le pertenece**.
- Ejemplo:

```
// programa.c

#include <stdlib.h>

int main(void)
{
     int* v;
     v = (int*) malloc(5*sizeof(int));
     v[5] = 10;

     return 0;
}
```



- A veces, el programa no se interrumpe y el error aparece más adelante → **difíciles de encontrar**

Valgrind



- Herramienta para encontrar *segmentation faults* y otros *memory leaks*.
- **Instalación**: `sudo apt install valgrind`
- **Uso:**
  - Compilamos el programa con las opciones -ggdb y -O0
  - Lo ejecutamos con:

```
valgrind --leak-check=full programa.exe
```

- Más info: https://valgrind.org/docs/manual/quick-start.html

Valgrind: ejemplo

- Probamos Valgrind con el programa anterior

```
gcc -ggdb -O0 programa.c -o programa.exe
valgrind --leak-check=full programa.exe
```

- Resultado:

```
 1    // programa.c
 2
 3    #include <stdlib.h>
 4    int main(void)
 5    {
 6          int* v;
 7          v = (int*) malloc(5*sizeof(int));
 8          v[5] = 10;
 9
10          return 0;
11    }
```

Valgrind: ejemplo

- Probamos Valgrind con el programa anterior

```
gcc -ggdb -O0 programa.c -o programa.exe
valgrind --leak-check=full programa.exe
```

- Resultado:

```
 1    // programa.c
 2
 3    #include <stdlib.h>
 4    int main(void)
 5    {
 6          int* v;
 7          v = (int*) malloc(5*sizeof(int));
 8          v[5] = 10;
 9
10          return 0;
11    }
```

```
==52391== Memcheck, a memory error detector
==52391== Copyright (C) 2002-2017, and GNU GPL'd, by Julian
Seward et al.
==52391== Using Valgrind-3.16.1 and LibVEX; rerun with -h
for copyright info
==52391== Command: ./a.out
==52391==
==52391== Invalid write of size 4
==52391==    at 0x109157: main (programa.c:8)
==52391==  Address 0x4a45054 is 0 bytes after a block of
size 20 alloc'd
==52391==    at 0x483E77F: malloc (vg_replace_malloc.c:307)
==52391==    by 0x10914A: main (programa.c:7)
==52391==
==52391==
```

Escritura inválida en la línea 8 de programa.c

```
==52391== HEAP SUMMARY:
==52391==     in use at exit: 20 bytes in 1 blocks
==52391==   total heap usage: 1 allocs, 0 frees, 20 bytes
allocated
==52391==
==52391== 20 bytes in 1 blocks are definitely lost in loss
record 1 of 1
==52391==    at 0x483E77F: malloc (vg_replace_malloc.c:307)
==52391==    by 0x10914A: main (programa.c:7)
==52391==
==52391== LEAK SUMMARY:
==52391==    definitely lost: 20 bytes in 1 blocks
==52391==    indirectly lost: 0 bytes in 0 blocks
==52391==      possibly lost: 0 bytes in 0 blocks
==52391==    still reachable: 0 bytes in 0 blocks
==52391==         suppressed: 0 bytes in 0 blocks
==52391==
==52391== For lists of detected and suppressed errors, rerun
with: -s
==52391== ERROR SUMMARY: 2 errors from 2 contexts
(suppressed: 0 from 0)>
```

Hay 20 bytes que se han “perdido” y que se asignaron en la línea 7 de programa.c

Valgrind: ejemplo

- Programa corregido:

```
 1    // programa.c
 2
 3    #include <stdlib.h>
 4    int main(void)
 5    {
 6          int* v;
 7          v = (int*) malloc(5*sizeof(int));
 8          v[4] = 10;
 9          free(v);
10
11          return 0;
12    }
```