# Ayudantía 2: debugging y profiling

# Debugging

Es el  proceso de identificar y corregir errores o bugs en el código.

- **Tipos de errores de Sintaxis:**
  a. Fallos en la estructura del código.
  b. Errores de Ejecución: Fallos que ocurren durante la ejecución.
  c. Errores Lógicos: El código se ejecuta sin errores, pero el resultado no es el esperado.

- **Técnicas:**
  a. Impresión de mensajes: std::cout para mostrar valores en tiempo de ejecución.
  b. Uso de asserts: Verifica condiciones durante la ejecución: assert(condition);
  c. Depuradores interactivos: Como GDB para inspeccionar el estado del programa.

# Herramienas debugging

- **GDB** (GNU Debugger): Herramienta estándar en Linux.
- **Valgrind**: Herramienta para detectar errores de memoria.
- **Visual Studio Debugger**: Integrado en el entorno de desarrollo Visual Studio.

# GDB

GDB permite ejecutar el programa paso a paso, inspeccionar variables, con la posibilidad de detenerlo cuando se cumple cierta condición (breaking points)

- GDB: http://www.gnu.org/software/gdb/
- GDB online: https://www.onlinegdb.com/

# GDB

GDB (GNU Debugger) es una herramienta para depurar programas en C++. Permite ejecutar el programa paso a paso, inspeccionar variables, y más.

**Comandos básicos:**

- **list:** muestra el código
- **run:** para iniciar la ejecución.
- **break num_linea:** coloca un punto de ruptura en el código, es decir, donde se dentendrá el programa
- **delete num:** elimina el punto de ruptura num
- **break funcion**: coloca un punto de ruptura en una función
- **continue:** continua con la ejecución del programa
- **next**: para avanzar una línea.
- **print variable:** para mostrar el valor de una variable.

**Ejemplo de uso:**

```
# Compilación
g++ -g -o programa programa.cpp

# Ejecución
gdb ./programa
```

# Ejercicio 1

**Comandos básicos:**

- **list:** muestra el código
- **run:** para iniciar la ejecución.
- **break num_linea:** coloca un punto de ruptura en el código, es decir, donde se dentendrá el programa
- **delete num:** elimina el punto de ruptura num
- **break funcion**: coloca un punto de ruptura en una función
- **continue:** continua con la ejecución del programa
- **next**: para avanzar una línea.
- **print variable:** para mostrar el valor de una variable.

**Ejemplo de uso:**

```
# Compilación
g++ -g -o programa programa.cpp

# Ejecución
gdb ./programa
```

# Valgrind

Valgrind es una herramienta que permite detectar problemas relacionados con la memoria en programas de C++ como fugas de memoria, accesos inválidos, uso de memoria no inicializada, entre otros.

**Comandos básicos:**

- **leak-check=full:** Detección completa de fugas de memoria.
- **track-origins=yes:** Comprobación de accesos inválidos a la memoria.
- **leak-check=summary:** Resumen de errores en la memoria.
- **log-file:** Guardar el log de Valgrind en un archivo.

**Ejemplo de uso:**

```
# Compilación
g++ -g -o programa programa.cpp

# Ejecución
valgrind --leak-check=full ./programa
```

# Ejercicio 2

**Comandos básicos:**

- **leak-check=full:** Detección completa de fugas de memoria.
- **track-origins=yes:** Comprobación de accesos inválidos a la memoria.
- **leak-check=summary:** Resumen de errores en la memoria.
- **log-file:** Guardar el log de Valgrind en un archivo.

**Ejemplo de uso:**

```
# Compilación
g++ -g -o programa programa.cpp

# Ejecución
valgrind --leak-check=full ./programa
```

# Profiling

Análisis de la ejecución de un programa para identificar cuellos de botella y mejorar el rendimiento.

**Herramientas**:

- Perf: Herramienta avanzada de Linux para el análisis de rendimiento.

# Perf

Perf es una herramienta para el análisis de rendimiento en sistemas Linux. Permite obtener información detallada sobre el uso de CPU, eventos de hardware, entre otros.

**Comandos básicos:**

- **perf stat:** Medición básica del rendimiento.
- **perf record:** Registro de eventos durante la ejecución.
- **perf report:** Análisis del perfil de rendimiento.
- **perf timechart record:** Medición de la latencia en la ejecución.
- **perf record -g:** Medición de rendimiento con análisis de gráfico de llamadas.

**Ejemplo de uso:**

```
# Compilación
g++ -o programa programa.cpp

# Ejecución
perf stat ./programa
```

# Ejercicio 3

**Comandos básicos:**

- **perf stat:** Medición básica del rendimiento.
- **perf record:** Registro de eventos durante la ejecución.
- **perf report:** Análisis del perfil de rendimiento.
- **perf timechart record:** Medición de la latencia en la ejecución.
- **perf record -g:** Medición de rendimiento con análisis de gráfico de llamadas.

**Ejemplo de uso:**

```
# Compilación
g++ -o programa programa.cpp

# Ejecución
perf stat ./programa
```

# Medición de tiempos

- Llamada al sistema
  - Biblioteca: **chrono**
  - Funciones
    - std::chrono::high_resolution_clock
    - std::chrono::duration

**Ejemplo de uso:**

```cpp
#include <iostream>
#include <chrono>

void funcion() {
    for (volatile int i = 0; i < 10000000; ++i);
}

int main() {
    auto start = std::chrono::high_resolution_clock::now();

    funcion();

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;

    std::cout<< "Tiempo en segundos: "<< duration.count() <<std::endl;
    return 0;
}
```

# Medición de tiempos

- **time** es una herramienta de línea de comandos en Linux que permite medir el tiempo de ejecución de un proceso.
- **Salidas**:
  - Real: Tiempo total de ejecución desde el inicio hasta el final del proceso.
  - User: Tiempo de CPU en modo usuario.
  - Sys: Tiempo de CPU en modo núcleo

**Ejemplo de salida:**

```
# Ejecución
time ./programa

# Salida
real 0m1.234s
user 0m0.567s
sys  0m0.123s
```

# FIN