Systems Architecture

7. Debugging tools for C programs

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

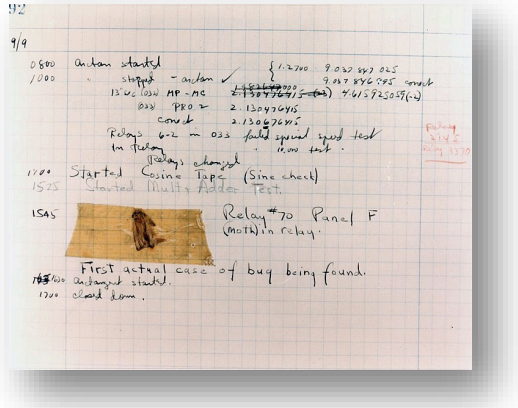
2025/2026

uc3m | Universidad Carlos III de Madrid

Table of contents

1. Introduction

- A software ***bug*** is a problem causing a program to crash or produce invalid output
  - Most of the time, bugs are due to human errors in source code or in the system design
- The term bug was used in an account by computer pioneer Grace Hopper in 1946, while working at the computing laboratory at Harvard University in two computers with names Mark II and Mark III
  - Operators traced an error in the Mark II to a moth trapped in a relay, coining the term bug



Source: https://en.wikipedia.org/wiki/Software_bug

- Since then, the term **debug** is used to name the process of detecting and correcting software defects (or *bugs*) in a system

1. Introduction

- **Logging** (i.e., inserting print statements or using logging libraries to output information about the program's state at various points in the execution) is commonly used by programmers to debug their programs
- In addition, there are specific tools (like **GDB** or **Valgrind**) for debugging

```
* * * Program started * * *
Attempting to divide 10 by 2
Entering divide function with a = 10, b = 2
Division successful, result = 5
Result of 10 / 2 = 5
* * * Program ended * * *
```

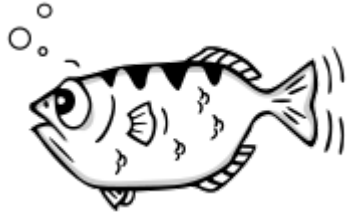


```c
#include <stdio.h>

int divide(int a, int b) {
    printf("Entering divide function with a = %d, b = %d\n", a, b);

    if (b == 0) {
        fputs("Error: Division by zero encountered\n", stderr);
        return 0;
    }

    int result = a / b;
    printf("Division successful, result = %d\n", result);

    return result;
}

int main() {
    puts("* * * Program started * * *");

    int x = 10;
    int y = 2;

    printf("Attempting to divide %d by %d\n", x, y);
    int result1 = divide(x, y);
    printf("Result of %d / %d = %d\n", x, y, result1);

    puts("* * * Program ended * * *");

    return 0;
}
```

Table of contents

2. GDB

- **GDB** (GNU Debugger) is a popular debugger tool that runs on Unix-like systems and works for different programming languages, including C, C++, or Go, among others
  - GDB was first written by Richard Stallman in 1986 as part of the GNU project
- GDB offers different features for tracing the execution of computer programs, such as:
  - Step by step execution
  - Monitor and modify the values of programs' internal variables
  - Call functions independently of the program's normal behavior



https://www.sourceware.org/gdb/

2. GDB

- To debug a C program with GDB, first, the program should be compiled including the `-g` flag
  - This flag is used to include debugging information (know as debug symbols) in the compiled executable
  - Those debug symbols are required by debugging tools like GDB to map the blocks of binary operations onto the original source code in the step-by-step execution

```
$ gcc -g program.c -o program
```

- Then, we need to invoke the command gdb using the program name as argument:

```
$ gdb program
```

2. GDB

```
$ gdb program
GNU gdb (Ubuntu 9.1-0ubuntu1) 9.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from program...
(gdb)
```

The interaction with the debugger is done through different commands typed at the prompt (like in the shell)

2. GDB

- The following table summarizes some of the most relevant gdb commands (I):

Command	Description
`run`	Start debugged program
`quit`	Exit gdb
`where`	Print backtrace (list of functions calls)
`up`	Goes up a level in the backtrace
`down`	Goes down a level in the backtrace
`list`	Show specified function or line
`break <line>`	Put a breakpoint in a line number
`info breakpoints`	Show the current breakpoints
`del <number>`	Delete a given breakpoint (by number)

2. GDB

- The following table summarizes some of the most relevant gdb commands (II):

Command	Description
`enable <number>`	Enable breakpoint by number
`disable <number>`	Disable breakpoint by number
`continue`	Continue execution after breakpoint
`next`	Continue execution after breakpoint until next function invocation
`step`	Continue execution after breakpoint step by step
`print <c_expression>`	Prints any C expression
`condition <number> <c_condition>`	Include a condition for breakpoint by number, using a condition in C language
`condition <number>`	Remove condition in breakpoint by number

2. GDB


```c
#include <stdio.h>

void printArray(int *arr, int size) {
    for (int i = 0; i <= size; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }
}

int main() {
    int numbers[] = { 10, 20, 30, 40, 50 };
    int size = sizeof(numbers) / sizeof(numbers[0]);

    printf("Array contents:\n");
    printArray(numbers, size);

    return 0;
}
```

This program has some bug. Let's debug it with GDB

```
Array contents:
arr[0] = 10
arr[1] = 20
arr[2] = 30
arr[3] = 40
arr[4] = 50
arr[5] = 32766
```

2. GDB

1. Compile the program with debugging information

```
gcc -g debug_gdb.c -o debug_gdb
```

2. Launch GDB with the compiled executable:

```
gdb ./debug_gdb
```

3. Set a breakpoint:

```
break 5
```

4. Run the program:

```
run
```

5. Inspect variables:

```
print i
print size
```

6. Step through the code:

```
next
continue
```

Table of contents

3. Valgrind

- **Valgrind** is a suite of tool for debugging (finding bugs) and profiling (performance analysis) C programs in **Linux** systems
- There are several tools included with Valgrind, such as:
  - Memcheck, which is a memory error detector (tool used by default)
  - Helgrind, which is a tool for detecting synchronization errors (e.g. race conditions)
  - Callgrind, which is a tool for profiling CPU usage

https://valgrind.org/

3. Valgrind

- In this unit we focus on **memcheck**, which allows to detect the following problems:
  - **Memory leaks**, which occurs when programmers create a memory in heap and forget to release it
    - Memory leaks reduce the performance of the computer by reducing the amount of available memory
    - Memory leaks can be particularly serious issues for programs like daemons and servers which are designed to work indefinitely
  - **Illegal memory frees**: free statement performed on a memory position which has not been previously allocated
  - **Illegal read/write**: Code that it is reading/writing illegal memory positions
  - **Uninitialized variables**: An uninitialized variable has an unspecified value, leading to potential problems

3. Valgrind

- To use memcheck in Valgrind we need to do the following:

1. Compile our program with the debug options:

```
gcc -g -o my_program my_program.c
```

2. Invoke Valgrind passing the executable as argument:

```
valgrind --tool=memcheck ./my_program
```

```
valgrind ./my_program
```

Since memchek is the default tool in Valgrind, these two command works exactly the same

3. Valgrind

- We can use other another capabilities of memcheck using the following configuration parameters:

1. `--leak-check=full` : To check memory leaks

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

2. `--show-leak-kinds=all` : To show all types of memory leaks

```
valgrind --leak-check=full --show-leak-kinds=all ./my_program
```

3. `--track-origins=yes` : To check uninitialized variables

```
valgrind --track-origins=yes ./my_program
```

https://www.valgrind.org/docs/manual/mc-manual.html

3. Valgrind - Memory leaks

- Example of program with memory leak:

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void leak() {
    void *ptr = malloc(100);
}

int main() {
    puts("Let's leak 100 bytes");
    leak();
    puts("100 bytes leaked");

    return 0;
}
```

Step 1. Compile program with debug option

```
gcc -g -o valgrind_memory_leak valgrind_memory_leak.c
```

Step 2. Invoke Valgrind using the generated binary

```
valgrind ./valgrind_memory_leak
```

Step 3. Rerun Valgrind (since a memory leak problem is found):

```
Rerun with --leak-check=full to see details of leaked memory
```

```
valgrind --leak-check=full --show-leak-kinds=all ./valgrind_memory_leak
```

3. Valgrind - Memory leaks

- Example of program with memory leak (output):

```
==2410== HEAP SUMMARY:
==2410==     in use at exit: 100 bytes in 1 blocks
==2410==   total heap usage: 2 allocs, 1 frees, 1,124 bytes allocated
==2410==
==2410== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2410==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2410==    by 0x10869B: leak (valgrind_memory_leak.c:12)
==2410==    by 0x1086C1: main (valgrind_memory_leak.c:17)
==2410==
==2410== LEAK SUMMARY:
==2410==    definitely lost: 100 bytes in 1 blocks
==2410==    indirectly lost: 0 bytes in 0 blocks
==2410==      possibly lost: 0 bytes in 0 blocks
==2410==    still reachable: 0 bytes in 0 blocks
==2410==         suppressed: 0 bytes in 0 blocks
==2410==
==2410== For counts of detected and suppressed errors, rerun with: -v
==2410== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

3. Valgrind - Illegal memory frees

- Example of program with illegal memory frees:

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    void *ptr = malloc(4);
    free(ptr);
    ptr++; // To mismatch
    free(ptr);

    return 0;
}
```

At this line, `ptr` is a **dangling pointer**, i.e., a pointer that does not point to a valid address

Step 1. Compile program with debug option

```
gcc -g -o valgrind_illegal_free valgrind_illegal_free.c
```

Step 2. Invoke Valgrind using the generated binary

```
valgrind ./valgrind_illegal_free
```

3. Valgrind - Illegal memory frees

- Example of program with illegal memory frees (output):

```
==3044== Invalid free() / delete / delete[] / realloc()
==3044==    at 0x4C32D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3044==    by 0x1086BC: main (valgrind_illegal_free.c:14)
==3044==  Address 0x522f041 is 1 bytes inside a block of size 4 free'd
==3044==    at 0x4C32D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3044==    by 0x1086AB: main (valgrind_illegal_free.c:12)
==3044==  Block was alloc'd at
==3044==    at 0x4C31B0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3044==    by 0x10869B: main (valgrind_illegal_free.c:11)
==3044==
==3044==
==3044== HEAP SUMMARY:
==3044==     in use at exit: 0 bytes in 0 blocks
==3044==   total heap usage: 1 allocs, 2 frees, 4 bytes allocated
==3044==
==3044== All heap blocks were freed -- no leaks are possible
==3044==
==3044== For counts of detected and suppressed errors, rerun with: -v
==3044== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

3. Valgrind - Illegal read/write

- Example of program with illegal read/write:

```c
#include <stdlib.h>

int main() {
    int *ptr = 0;
    *ptr = 33;

    return 0;
}
```

Step 1. Compile program with debug option

```
gcc -g -o valgrind_illegal_write valgrind_illegal_write.c
```

Step 2. Invoke Valgrind using the generated binary

```
valgrind ./valgrind_illegal_write
```

3. Valgrind - Illegal read/write

- Example of program with illegal read/write (output):

```
==3243== Invalid write of size 4
==3243==    at 0x10860A: main (valgrind_illegal_write.c:11)
==3243==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==3243==
==3243==
==3243== Process terminating with default action of signal 11 (SIGSEGV)
==3243==  Access not within mapped region at address 0x0
==3243==    at 0x10860A: main (valgrind_illegal_write.c:11)
==3243==  If you believe this happened as a result of a stack
==3243==  overflow in your program's main thread (unlikely but
==3243==  possible), you can try to increase the size of the
==3243==  main thread stack using the --main-stacksize= flag.
==3243==  The main thread stack size used in this run was 8388608.
==3243==
==3243== HEAP SUMMARY:
==3243==     in use at exit: 0 bytes in 0 blocks
==3243==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==3243==
==3243== All heap blocks were freed -- no leaks are possible
==3243==
==3243== For counts of detected and suppressed errors, rerun with: -v
==3243== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

3. Valgrind - Uninitialized variables

- Example of program with uninitialized variables:

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    int number;
    if (number == 0) {
        printf("number is zero");
    }

    return 0;
}
```

Step 1. Compile program with debug option

```
gcc -g -o valgrind_unitialized valgrind_unitialized.c
```

Step 2. Invoke Valgrind using the generated binary

```
valgrind ./valgrind_unitialized
```

Step 3. Rerun Valgrind (since uninitialized variables are found):

```
Use --track-origins=yes to see where uninitialised values come from
```

```
valgrind --track-origins=yes ./valgrind_unitialized
```

3. Valgrind - Uninitialized variables

- Example of program with uninitialized variables (output):

```
==3745== Conditional jump or move depends on uninitialised value(s)
==3745==    at 0x108656: main (valgrind_unitialized.c:12)
==3745==  Uninitialised value was created by a stack allocation
==3745==    at 0x10864A: main (valgrind_unitialized.c:10)
==3745==
number is zero==3745==
==3745== HEAP SUMMARY:
==3745==     in use at exit: 0 bytes in 0 blocks
==3745==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==3745==
==3745== All heap blocks were freed -- no leaks are possible
==3745==
==3745== For counts of detected and suppressed errors, rerun with: -v
==3745== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Table of contents

4. Takeaways

- **Debugging** is the process of finding and fixing defect (bugs) in the source code of any software
- **GDB** is a tool which allows to debug C program using the command line
- One of most relevant problems related to dynamic memory are called **memory leaks**, which happens when allocating memory in the heap and forget to release it
- We can use **Memcheck** (through **Valgrind**) to detect problems related to memory management such as memory leaks, illegal memory frees, illegal read/write, and uninitialized variables