Debugging Crash Course

Lluís Vilanova
vilanova@ac.upc.edu

1 Introduction

The debugger will help you in examining the state of a program and analyzing sources of errors. This document provides a short introduction to using gdb, the GNU Project Debuggger. You can get more information on its webpage (http://www.gnu.org/software/gdb/), as well as can access its manual using the info program (you must install the gdb-doc package):

```
$ info gdb
```

2 Starting a debugging session

First of all, you must ensure your program is compiled with debugging information enabled (see the appendix section for the source code). You can do this by passing the -g flag to the gcc compiler. It is also recommended to disable all compiler optimizations (-O0) in order to have a more pleasant debugging experience, and to tell the compiler to issue warnings to catch potential errors during compilation (-Wall):

```
$ gcc -g -O0 -Wall -o program program.c
program.c: In function ‘main’:
program.c:26:10: warning: ‘arr3’ may be used uninitialized in this
                          function [-Wuninitialized]
$ ./program 2
0
Segmentation fault
```

To start gdb with your program, simply type:

```
$ gdb ./program
GNU gdb (GDB) 7.6 (Debian 7.6-5)
...
Reading symbols from ./program... done.
(gdb)
```

You can then start you program with the run command, which can also pass arguments to your program:

```
(gdb) run 2
```

Alternatively, you can start the debugger with specific program arguments, and run (without arguments) will always restart the program with the arguments specified at command-line:

```
$ gdb -args ./program 2
GNU gdb (GDB) 7.6 (Debian 7.6-5)
...
Reading symbols from program... done.
(gdb) run
...
```

To run a command in `gdb`, you just need to type the first letters of its name, and then press `tab` to automatically complete it (or show all possible completions). Completion not only works for `gdb` commands, but also for application symbols (i.e., variables and functions). Many commonly used commands also have a shorter command name; for example, `run` and `r` are the same command; see the `gdb` manual for more details. Additionally, you can always get information about a command with the `help` command:

```
(gdb) help run
Start debugged program.  You may specify arguments to give it.
Args may include "*", or "[...]"; they are expanded using "sh".
Input and output redirection with ">", "<", or ">>" are also allowed.

With no arguments, uses arguments last specified (with "run" or "set args").
To cancel previous arguments and run with no arguments,
use "set args" without arguments.
```

The rest of the examples on this document assume you are always running the same `gdb` session.

Finally, remember to enable optimizations after ensuring the program is correct, in order to get better performance (`-O3`; see `man gcc`).

3 Understanding crashes

After starting `program`, it will crash with a *Segmentation fault*, an access to an invalid memory address. The debugger will tell us the offending line, and the `backtrace` command will show us how we did get there:

```
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
0x00000000004005ad in print_elem (arr=0x0, pos=2) at program.c:7
7           int elem = arr[pos];
(gdb) backtrace
#0  0x00000000004005ad in print_elem (arr=0x0, pos=2) at program.c:7
#1  0x000000000040064b in main (argc=2, argv=0x7fffffffe2f8) at program.c:25
```

The backtrace shows what functions called what other functions. In the example, we see that function *main* in line 24 of *program.c* called function *print\*$_{elem}$, which is now stopped at line 7 of *program.c*. This tells us the second call to *print\*$_{elem}$ is the one that crashed (*program.c* at line 25). We can actually see this with the `frame` (to move between frames in the backtrace) and `list` (to see a piece of the source code) commands:

```
(gdb) frame 1
#1  0x000000000040064b in main (argc=2, argv=0x7fffffffe2f8) at program.c:25
```

```
25          print_elem(arr2, position);
(gdb) list
20          }
21
22          int position = atoi(argv[1]);
23
24          print_elem(arr1, position);
25          print_elem(arr2, position);
26          print_elem(arr3, position);
27
28          return 0;
29      }
```

Since the value of *arr* in the call to $print\backslash_{elem}$ is zero (we see that on frame 0 of thebacktrace, as *arr=0x0*), that's clearly the reason for the invalid error. It's customary on most systems to have the first 4KB of memory non-accessible, such that zero (`NULL`) can be used as an address that will fail if used.

In addition to `frame` (which jumps to a specific frame), you can also use `up` and `down` to move to the previous/next frame:

```
(gdb) backtrace
#0  print_elem (arr=0x0, pos=2) at program.c:7
#1  0x000000000040064b in main (argc=2, argv=0x7fffffffe2f8) at program.c:25
(gdb) frame 0
#0  print_elem (arr=0x0, pos=2) at program.c:7
7           int elem = arr[pos];
(gdb) up
#1  0x000000000040064b in main (argc=2, argv=0x7fffffffe2f8) at program.c:25
25          print_elem(arr2, position);
(gdb) down
#0  print_elem (arr=0x0, pos=2) at program.c:7
7           int elem = arr[pos];
```

4 Breakpoints

You can tell `gdb` to stop execution of your program at a specific place with the `breakpoint` command, which also accepts stopping only if a condition is true. Once the program is stopped, execution can be resumed with the `continue` command:

```
(gdb) break print_elem
(gdb) break print_elem if arr == 0
(gdb) break program.c:25
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400599 in print_elem at program.c:7
2       breakpoint     keep y   0x0000000000400599 in print_elem at program.c:7
        stop only if arr == 0
3       breakpoint     keep y   0x000000000040063a in main at program.c:25
(gdb) run
Breakpoint 1, print_elem (arr=0x601010, pos=2) at program.c:7
```

```
7           int elem = arr[pos];
(gdb) continue
Continuing.
0

Breakpoint 3, main (argc=2, argv=0x7fffffffe2f8) at program.c:25
25          print_elem(arr2, position);
(gdb) continue
Continuing.

Breakpoint 1, print_elem (arr=0x0, pos=2) at program.c:7
7           int elem = arr[pos];
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005ad in print_elem (arr=0x0, pos=2) at program.c:7
7           int elem = arr[pos];
```

You can delete a breakpoint with the `delete` command, and can temporarily disabled/enable it with the `disable` or `enable` commands:

```
(gdb) delete 1
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x0000000000400599 in print_elem at program.c:7
        stop only if arr == 0
        breakpoint already hit 1 time
3       breakpoint     keep y   0x000000000040063a in main at program.c:25
        breakpoint already hit 1 time
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: ./program 2
0

Breakpoint 3, main (argc=2, argv=0x7fffffffe2f8) at program.c:25
25          print_elem(arr2, position);
(gdb) continue
Continuing.

Breakpoint 2, print_elem (arr=0x0, pos=2) at program.c:7
7           int elem = arr[pos];
```

You can always stop the program with `Ctrl-c`. When the program is stopped, you can also use the `next` and `step` commands to control the execution of the program at the granularity of lines of code (`nexti` and `stepi` will do the same at the granularity of assembly instructions).

5 Examining state

When the application is stopped, you can use the `print` command to see the value of arbitrary expressions, including the use of variables in the program you are debugging:

```
...
Breakpoint 2, print_elem (arr=0x0, pos=2) at program.c:7
7           int elem = arr[pos];
(gdb) print 10 + 10
$1 = 20
(gdb) print /x 10 + 10
$2 = 0x14
(gdb) print &arr[pos]
$3 = (int *) 0x8
```

You might also find `display` (and `undisplay`) interesting, which will print the given expression every time after executing a command. In certain situations it is also useful to use the `x` command to examine the raw contents of memory.

6 Miscellaneous

There's a few more commands that can come in handy. The `layout` command shows a more user-friendly interface (*Ctrl+x o* to switch between windows; *Ctrl+x a* disables the interface). The `info symbol` and `info address` commands show information about specific addresses and symbols. Finally, for more complex programs, you can use the `valgrind` application to detect memory problems.

7 Appendix: `program.c`

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


void print_elem (int *arr, int pos)
{
    int elem = arr[pos];
    printf("%d\n", elem);
}

int main (int argc, char *argv[])
{
    int *arr1 = malloc(sizeof(int) * 10);
    int *arr2 = NULL;
    int *arr3;

    if (argc != 2) {
        printf("Usage: %s <position>\n", argv[0]);
        return 1;
    }

    int position = atoi(argv[1]);
    print_elem(arr1, position);
    print_elem(arr2, position);
    print_elem(arr3, position);

    return 0;
}
```