Piscine — Tutorial D4 PM

version **#b4173ffc5996c36bf7f9730185ebc8357500bd86**

Assistants C/Unix 2024 <assistants@tickets.assistants.epita.fr>

Contents

*https://intra.forge.epita.fr

1 GDB(1)

The **GNU Debugger** also called **GDB**, is the standard debugger of the GNU project. It can run on most popular UNIX and Microsoft Windows variants and supports many languages like *Ada*, *C*, *C++* and others. It was created by Richard Stallman in 1988 and is an open source piece of software distributed under the GNU GPL license.

Although there are many GDB GUI front ends, they provide no additional features and you first have to master it with its default text interface. During the entire *piscine* period, GDB is the only **debugger** allowed.

2 Basic

2.1 Debugging symbols

If you execute the `file` command on an object file or an executable, you can see that the file format is called ELF (Executable and Linkable Format). ELF is used by most UNIX systems, including GNU/Linux, to store compiled code. As a medieval fantasy complement to ELF, the DWARF debugging data format was created. The purpose of DWARF is to link your source code with the compiled instructions in ELF binaries.

We can compile our code any way we want and ask `GCC` to include debugging symbols (link between a name and an address) in the generated binary that we want to debug.

`GCC` provides several options to include debugging information. We ask you to remember the most common one, `-g`, which produces debugging information in the system's **native format** that GDB can use.

`GCC` allows us to select the level of debugging information included in the binary thanks to the `-g<level>` option. The default level is `2` and the maximum level is `3`. For additional information on those levels, please check the manual. The usage of `-g` is preferred but you can always use `-g3` if you need to.

- **File:** `debugme1.c`

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

void reverse(char *input, unsigned short index)
{
    if (index >= 0)
    {
        putchar(input[index]);
        reverse(input, --index);
    }
    else
    {
        putchar('\n');
    }
}
```

(continues on next page)

(continued from previous page)

```c
int main(void)
{
    char str[] = "orea.gnieob@s_ottacs";
    size_t len = strlen(str);
    reverse(str, len);

    return 0;
}
```

The next part of this tutorial is based on a set of programs to debug. Source code of the first program is **debugme1**. Compile it to continue (*with debugging symbols*): `gcc -g debugme1.c -o debugme1` and then run it. It crashes.

```
42sh$ ./debugme1
    segmentation fault (core dumped)  ./debugme1
```

2.2 Using GDB

It is now time to debug this first program. Run:

```
42sh$ gdb debugme1
```

GDB is quite wordy at startup:

```
GNU gdb (GDB) 9.2
Copyright (C) 2018 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-pc-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".
```

Notice the second to last line, telling you that GDB has actually found debugging symbols:

```
Reading symbols from debugme1...
```

If not, please check your `Makefile` or `command line`. Finally, we get to the `GDB shell`:

```
(gdb)
```

It works like a traditional shell, providing a set of commands and basic auto-completion.

Tips

You can also run `gdb` whitout any arguments, but note that if you don't specify the executable file to run, you will have to do it later inside GDB.

2.2.1 help

The `help` command allows you to have a quick description of a command, as well as its syntax. Without options, it gives the list of *classes of commands*. You can then search and explore all opportunities given by GDB by searching a *class of commands*.

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

2.2.2 apropos

For documentation again, the `apropos` command, stated in the GDB introductory message, waits for a **regular expression** and returns a list of commands for the searched expression. This is a good way to find GDB commands related to a concept or key words.

For example, to list GDB commands related to **breakpoint**:

```
(gdb) apropos breakpoint
b -- Set breakpoint at specified location
br -- Set breakpoint at specified location
bre -- Set breakpoint at specified location
brea -- Set breakpoint at specified location
break -- Set breakpoint at specified location
break-range -- Set a breakpoint for an address range
breakpoints -- Making program stop at certain points
c -- Continue program being debugged
cl -- Clear breakpoint at specified location
```

(continues on next page)

(continued from previous page)

```
clear -- Clear breakpoint at specified location
commands -- Set commands to be executed when a breakpoint is hit
condition -- Specify breakpoint number N to break only if COND is true
continue -- Continue program being debugged
[...]
```

Then you can just check the help of the `continue` command.

2.2.3 quit

The `quit` command is perfectly described by:

```
(gdb) help quit
Exit gdb.
```

Tips

Copyright and various information messages appearing at GDB startup can be skipped, thanks to the `-q` option.

2.3 Running a program inside GDB

The goal of this part is to run our program inside GDB, see it crash, and be able to get a fundamental debugging information: the backtrace.

The **backtrace** is a view on the call stack. It gives you the name and the parameters of the function you are in, the function that called it etc., all the way from the `main` function. It also gives the corresponding location of these functions in the source file (*provided that you have debugging symbols*). It allows you to figure out two essential pieces of information:

- The exact line in your code where the program crashed.
- The path your program followed to get there, with the successive function calls.

2.3.1 run

Let's run our program with the `run` command. It can take the arguments that you would normally give the program. You could have used the `--args` option of `gdb`, followed by the arguments to give to your program.

```
(gdb) run
Starting program: /home/acu/gdb/debugme1/debugme1

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555176 in reverse (input=0x7fffffffe7b1 "orea.gnieob@s_ottacs", index=65535) at␣
↪debugme1.c:8
8           putchar(input[index]);
(gdb)
```

The program crashes, as expected. We observe here that we received a `SIGSEGV` signal from the operating system, more commonly called a **Segmentation Fault**. The output tells us, in order:

- The address of the guilty instruction: `0x0000555555555176`
- The function where the error occurred: `reverse`
- The name and the values of its parameters: `(input=0x7fffffffe7b1 "orea.gnieob@s_ottacs", index=65535)`
- Location in the source code, file and line: `at debugme1.c:8`
- The corresponding line of code: `8 putchar(input[index]);`

Tips

Some of you may have noticed, sometimes `run` produces the error:

```
(gdb) run
Starting program:
No executable file specified.
Use the "file" or "exec-file" command.
(gdb)
```

Which means you have not specified an executable file when you first ran `gdb`. This can be easily fixed with the `file` command.

2.3.2 file

`file` specifies the program to be debugged. It reads for its symbols and also allows the program to be executed when you use the run command.

```
(gdb) file debugme1
Reading symbols from debugme1...done.
(gdb)
```

`file` will ask you for permission if you try to load a new symbol table from another executable, but have already set one up or specified it when you ran `gdb`.

2.3.3 backtrace

GDB already gave you some information, and you know where to start tracking the bug's cause. But you can ask for more running the `backtrace` command:

```
(gdb) backtrace
#0  0x0000555555555176 in reverse (input=0x7fffffffe7b1 "orea.gnieob@s_ottacs", index=65535)␣
↪at debugme1.c:8
#1  0x000055555555519a in reverse (input=0x7fffffffe7b1 "orea.gnieob@s_ottacs", index=65535)␣
↪at debugme1.c:9
#2  0x000055555555519a in reverse (input=0x7fffffffe7b1 "orea.gnieob@s_ottacs", index=0) at␣
↪debugme1.c:9
#3  0x000055555555519a in reverse (input=0x7fffffffe7b1 "orea.gnieob@s_ottacs", index=1) at␣
```

(continues on next page)

(continued from previous page)

```
debugme1.c:9
#4  0x000055555555519a in reverse (input=0x7fffffffe7b1 "orea.gnieob@s_ottacs", index=2) at debugme1.c:9
#5  0x000055555555519a in reverse (input=0x7fffffffe7b1 "orea.gnieob@s_ottacs", index=3) at debugme1.c:9
...
#21  0x000055555555519a in reverse (input=0x7fffffffe7b1 "orea.gnieob@s_ottacs", index=19) at debugme1.c:9
#22  0x00005555555551f1 in main () at debugme1.c:19
```

GDB gives you here, in reverse order, the list of functions that were called from `main()`. This call stack is composed of `frames`; every `frame` contains function arguments and local variables.

2.3.4 frame

It is now time to introduce the command of the same name: `frame`. With no arguments, it prints information about the current frame. But it mostly allows you to switch to any `frame` composing the `backtrace`, allowing you to analyze their state. Thus, by selecting `frame 22`, you will walk up the call stack and position gdb's view point in frame 22 of the backtrace, in the `main` function. At this point, every command you run will give you information about the `main` function at the time it called `reverse`.

The interest of navigating in the backtrace is to be able to get information about a specific frame.

Here, the `info` command will allow us to extract data contained in the frame we are currently focused on. With `info locals`, you can print the local variables. The `str` and `len` variables are declared in the `main` function. Print their value:

```
(gdb) frame 22
#22  0x00005555555551f1 in main () at debugme1.c:19
19      reverse(str, len);
(gdb) info locals
str = "orea.gnieob@s_ottacs"
len = 20
```

Remember the `up` and `down` commands, they are here to ease your navigation within the backtrace, switching to the previous or next frame. You can obtain more information with `backtrace full`, you will get the list of the function's local variables.

```
(gdb) backtrace full
#0  0x0000555555555176 in reverse (input=0x7fffffffe7b1 "orea.gnieob@s_ottacs", index=65535) at debugme1.c:8
No locals.
#1  0x000055555555519a in reverse (input=0x7fffffffe7b1 "orea.gnieob@s_ottacs", index=65535) at debugme1.c:9
No locals.
[...]
#22  0x00005555555551f1 in main () at debugme1.c:19
    str = "orea.gnieob@s_ottacs"
    len = 20
```

Tips

`backtrace full` will also print the value of local variables in each function of the backtrace.

Tips

Many `gdb` commands can be shortened. For example you can use the command `bt` instead of `backtrace`, `r` for `run`, etc. Be curious and search others. You will save an incredible amount of time.

2.3.5 Application

It's time to practice!

Some commands you can try:

- `run`: Run the program in GDB
- `backtrace`: Print the backtrace
- `info locals`: Print the local variables declared in the current `frame`
- `up`: Go up in the backtrace
- `down`: Go down in the backtrace
- `backtrace full`: Print the backtrace with local variable

Tips

Now that you will work with arrays and pointers, segfaults will often occur during your practice[1]. Try to use `gdb(1)` before calling an assistant, it will save you a lot of time!

[1] And they are a lot less fun than the facebook page.

With all the information gathered, **find**, **explain**, and **fix** the bug in **debugme1** and **debugme2**.

- **File:** `debugme2.c`

```
#include <stdlib.h>

#define SIZE 69000

void initialize(int *tab, int index, int value)
{
    tab[index] = value;
}

void guilty_function()
{
    int *tab;
    int i;

    tab = malloc(SIZE);

    for (i = 0; i <= 69000000; ++i)
```

(continues on next page)

(continued from previous page)

```
        initialize(tab, i, i);

    free(tab);
}

int main(void)
{
    guilty_function();

    return 0;
}
```

2.4 Breakpoints and execution flow

The goal of this part is to learn how to control the execution flow of your program by setting *breakpoints*. Indeed, if getting the *backtrace* of a program that crashed is instructive, controlling its execution flow and pausing the program wherever you want is even more informative.

You will learn how to manually execute your program step by step, meticulously observing the behavior of your program and tracking the values of your variables.

2.4.1 break

A *breakpoint* is a marker placed at a specific location in the code. The program will temporarily pause its execution so that you can inspect its state (like previously when debugging a *Segmentation Fault*).

To set a *breakpoint*, just use the `break` command. Without arguments, it sets a breakpoint at the current instruction (the one where the program is currently stopped). But you can also specify where to set the breakpoint in your code, for example at the beginning of a function with the `break function_name` syntax (*see tips below for additional syntaxes*).

For example with `debugme1`:

```
(gdb) break reverse
Breakpoint 1 at 0x401148: file debugme1.c, line 8.
```

Once your *breakpoints* are set, run your program with `run`. If your program does not crash before, it will stop at your first *breakpoint*!

```
(gdb) run
Starting program: /home/acu/gdb/debugme1/debugme1

Breakpoint 1, reverse (input=0x7fffffffe330 "orea.gnieob@s_ottacs", index=20) at debugme1.c:8
8           putchar(input[index]);
```

Tips

If you want to break from the start of the `main` function, simply use `start` command, this will set a *temporary breakpoint* on `main` and then run the program.

2.4.2 Print information

Once your program has reached a breakpoint, use the `print` command to print the value of a variable. You can set other breakpoints too.

```
(gdb) print index
index = 20
```

With `info args`, you can print a list of the current function's arguments.

```
(gdb) info args
input = 0x7fffffffe330 "orea.gnieob@s_ottacs"
index = 6
```

If you want to know more about the code *around* your position, just use the `list` command. Used successfully, it prints the 10 lines around your location in the code source, then the 10 lines that follows, etc. With `list -`, you can get 10 lines back.

To resume the execution, you can do:

- `continue`: executes the program until the next breakpoint (or the next segfault...).
- `next`: executes the next line of code and stops (if this line is a function call, it will stop after the function returned).
- `step`: executes the next line of code and stops (if this line is a function call, it will stop at **the first line of the called function**).
- `finish`: executes the program until the current function returns.

The list of set breakpoints can be accessed with `info breakpoints`

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000401148 in reverse at debugme1.c:8
        breakpoint already hit 1 time
2       breakpoint     keep y   0x00000000004006a1 in main at debugme1.c:12
3       breakpoint     keep y   0x0000000000400652 in reverse at debugme1.c:7
```

To remove a breakpoint, use `delete <breakpoin_num>`. Without arguments, it deletes all breakpoints.

2.4.3 Application

Use your new skills to debug `debugme3`.

You can see the ways of setting a *breakpoint* by typing `help break`. Here are the different ways of setting a **breakpoint**:

line	*break 36*
file:line	*break list.c:42*
function	*break print_array*
**address*	*break *0xdeadcafe*

You can get the backtrace at any time during execution, not only when your program crashed!

break	Set a *breakpoint* on the current line
break <location>	Set a *breakpoint* at the specified location
info breakpoints	List set *breakpoints*
delete <breakpoint_num>	Delete the specified *breakpoint*
delete	Delete **all** the *breakpoints*
print <var>	Print the value of a variable
continue	Continue the program execution until the next *breakpoint*
next	Execute the next line of code
step	Same as *next* but enters the function calls
list	Print 10 lines around the current line of code
list -	Print 10 lines above the current line of code

- **File:** `debugme3.c`

```
#include <stdio.h>

int recursive_function(int n, int a)
{
    if (n == 13)
        return a;
    else
        return recursive_function(n * 2, a + n);
}

int main(void)
{
    int n = 1;
    int a = 42;

    printf("result: %i\n", recursive_function(n, a));

    return 0;
}
```

2.4.4 Conditional breakpoints

Being able to set breakpoints is very useful. However, when the bug is in a loop and occurs only at a particular iteration, it becomes painful to use *continue* until we reach the good iteration. Fortunately, there are *conditional breakpoints*, which allow us to assign a specific condition to the breakpoint.

Here is an example:

```
(gdb) break my_function if variable == 100
Breakpoint 1 at 0x4005e8: file test.c, line 10.
(gdb) run
```

In the case you have already set a breakpoint and you want to assign a condition to it afterwards, just use the *condition* command:

```
(gdb) b my_function
Breakpoint 1 at 0x4005e8: file test.c, line 10.
(gdb) condition 1 variable == 100
```

Moreover if you want to stop the program after *n* crossings of a specified breakpoint for example, you can use *ignore* command. It works by specifying how much time you want to cross this breakpoint without being stopped.

Let's say we want to know to value of *res* after 100 crossings:

```
(gdb) break my_function
Breakpoint 1 at 0x4005e8: file test.c, line 10.
(gdb) ignore 1 100
Will ignore next 100 crossings of breakpoint 1.
```

Therefore, when the breakpoint is crossed for the 101st time, it will break as usual.

2.4.5 Application

Use your new skills to get the value of the *res* variable when *i* equals 200, and then after 350 crossings of the *recursive_function*.

break <location> if <condition>	Set a *conditional breakpoint*
condition <breakpoint> <condition>	Apply a condition to a specified breakpoint
ignore <breakpoint> <count>	Skip <breakpoint> <count> times.
display <var>	*<var>* will print each time time the program is stopped
info display	List the active displays
undisplay <disp_num>	Disable the display of *<disp_num> display*
undisplay	Disable all the displays

- **File:** `debugme4.c`

```
#include <stdio.h>

int recursive_function(int i)
{
    int res = 0;

    if (!i)
    {
        return i;
    }
    else
    {
        for (int j = 0; j <= i; ++j)
        {
            res += j + j * j / i;
        }
    }

    return res + recursive_function(i - 1);
```

(continues on next page)

(continued from previous page)

```
}

int main(void)
{
    int val = 6942;

    printf("recursive_function(%d) = %d\n", val, recursive_function(val));

    return 0;
}
```

2.4.6 Variable assignment

GDB gives you total control over the execution environment. This includes editing variables from the current `frame`. The `set var` command allows you to change a variable value. This command coupled with `checkpoints` (*explained in the advanced part of this tutorial*) allows you to perform some tests and modify the program execution without recompiling (*and interrupting your debugging session*).

```
(gdb) set var width = 42
```

2.4.7 Application

- **File:** `debugme5.c`

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

void never_called(size_t a)
{
    printf("Never Call Me Daddy ! (a = %zu)\n)", a);
}

int main(void)
{
    char string[] = "Adrien";
    size_t a = 2;

    if (a == strlen(string))
        never_called(a);

    return 0;
}
```

Try to call the *never_called* function.

2.4.8 Execution flow

GDB allows you to alter the execution path of your program:

- *jump <address>*: jumps at the given address and continue the execution from this point (if the given address is invalid, you will probably segfault easily);
- *call function(params...)*: calls the specified function. Beware of eventual side effects the function you are calling may have (global variables modification, ...)

set var <var> = <value>	Replace the value *<var>* with *<value>*
jump <address>	Continue execution at the specified
call function(params...)	Call a specified function

Try to solve the previous exercise with the help of these two new functionalities.

2.4.9 Value Optimized Out

- **File:** `debugme6.c`

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

static int eval_exp(char **str)
{
    int res = 0;

    do
    {
        res = res * 10;
        ++(*str);
    } while (**str <= 'a');

    return 18;
}

int main(void)
{
    char *exp[] = { "Antoine", "ACU Love You" };

    for (int i = 0; i < 2; ++i)
        printf("%d\n", eval_exp(exp + i));

    return 0;
}
```

Specific compiler options allow to optimize the produced code, but mislead the debugger which does not know how to handle it:

```
42sh$ gcc -Wall -Wextra -Werror -std=c99 -Wvla -pedantic -g -O2 -o debugme6 debugme6.c
```

Run this program without arguments and study its expected behavior by executing step by step with the *step* command. Its behavior is completely erratic, and we try to access some variables (like *i* in the *main* function), GDB returns things like:

- No symbol `str` in current context
- str = `<synthetic pointer>`.

The reason for this is simple: by using the `-O2` compiling option, we are asking the compiler to highly optimize the program, involving things like instructions removal, addition, applying special treatment to certain variables (move to registers, etc.). Just keep in mind that debugging an optimized code is extremely difficult and unadvised.

Just compile without optimization (without `-O` option or by setting `-O0` to explicitly disable all optimizations).

Going further...

If you still want to debug an optimized binary, then you may leave the source code level and switch to the *assembly* view, using the adapted *disassemble*, *nexti* and *stepi* commands.

3 Advanced

3.1 Watchpoints

With *watchpoints*, you can ask GDB to watch an expression or variable, and stop the execution whenever it is updated.

```
(gdb) watch len
Hardware watchpoint 2: len
(gdb) continue
...
Hardware watchpoint 2: len

Old value = 0
New value = 36
```

The execution stops when there is a write access on the *len* variable. If you want to watch the read accesses too, you can use *awatch* (for *access watchpoint*):

```
(gdb) awatch len
Hardware access (read/write) watchpoint 2: len
(gdb) next
...
Hardware access (read/write) watchpoint 2: len

Old value = 0
New value = 36
main (argc=1, argv=0x7fffffffdab8) at debugme1.c:22
22          return 0;
```

You also have *rwatch* to only watch read accesses but not write accesses.

Tips

`-location` or `-l` option will give the line of code causing the change.

watch <expr> [-l]	Watch modifications of '*<expr>* and stop the program execution
rwatch <expr> [-l]	Watch the read accesses to '*<expr>* and stop the program execution
awatch <expr> [-l]	Watch the read/write accesses to '*<expr>* and stop the program execution

Try it by yourself with the `debugme4.c`!

3.2 Pretty Print

GDB provides variables that control how arrays, structures and symbols are printed. For example, setting the *print pretty* variable to *on* updates the way structures are printed:

```
(gdb) print mystruct
$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, meat = 0x54 "Pork"}

(gdb) set print pretty on
(gdb) print mystruct
$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
    sour = 1
  },
  meat = 0x54 "Pork"
}
```

Same goes for the arrays, with *print array* and *print array-indexes* variables:

```
(gdb) print my_array
$2 = {1, 2, 3, 4}

(gdb) set print array on
(gdb) print my_array
$4 =   {1,
  2,
  3,
  4}

(gdb) set print array-indexes on
(gdb) print my_array
$5 =   {[0] = 1,
  [1] = 2,
  [2] = 3,
  [3] = 4}
```

Tips

Check out the GDB documentation to get more information about available variables.

set print pretty on	Pretty printing of structures
set print array on	Column view of arrays
set print array-indexes on	Print arrays indexes

Let's pratice! Try to pretty print the first element of the array and then to pretty print the whole array.

- **File:** `debugme7.c`

```c
#include <stdio.h>

#define ARRAY_SIZE 6

enum shirt_size
{
    XS,
    S,
    M,
    L,
    XL
};

struct ACU
{
    char *name;
    char *role;
    enum shirt_size size;
    unsigned int age;
};

int main(void)
{
    struct ACU ACU2024[ARRAY_SIZE] = {
        { .name = "Adrien", .role = "Piscine", .size = M, .age = 69 },
        { .name = "Antoine", .role = "Piscine", .size = XL, .age = 2 },
        { .name = "Ethan", .role = "Piscine", .size = M, .age = 42 },
        { .name = "Nigel", .role = "Piscine/HTTPD", .size = M, .age = 20 },
        { .name = "Pauline", .role = "Piscine/SQL", .size = M, .age = 22 },
        { .name = "Thuraya", .role = "Piscine/SQL", .size = S, .age = 21 }
    };

    for (size_t i = 0; i < ARRAY_SIZE; ++i)
        printf("name: %s\n", ACU2024[i].name);

    return 0;
}
```

3.3 Coredumps

When testing a program outside of GDB, a **segfault** or **abort** can happen unexpectedly. In that case, it can be tedious to launch the program again in GDB and try to reproduce the cause of the crash, so that you can properly study it.

However, when such an unexpected crash occur, the kernel generates a snapshot of the program's state and writes it to a file, this is what we call a **coredump**. GDB understands coredumps and can start a debugging session directly from it, so that you can inspect the program's state at the time of the crash.

On the environment you are working on, coredumps are completely handled by **systemd**, there are two ways to use them.

```
42sh$ cat fail.c
#include <assert.h>

int main(void)
{
    assert(0 && "what an unexpected crash!");
    return 0;
}

42sh$ make CC=gcc CFLAGS="-Wall -Wextra -Werror -Wvla -pedantic -std=c99 -g" fail
gcc -Wall -Wextra -Werror -Wvla -pedantic -std=c99 -g    fail.c   -o fail

42sh$ ./fail
fail: fail.c:5: main: Assertion `0 && "what an unexpected crash!"' failed.
Aborted (core dumped)
```

By retrieving the coredump first, then launching gdb with it:

```
42sh$ coredumpctl dump fail > fail.core
    [...]
42sh$ gdb -q fail fail.core
Reading symbols from fail...done.
[New LWP 22877]

Core was generated by `./fail'.
Program terminated with signal SIGABRT, Aborted.
#0  0x00007fe1fd3db5f8 in raise () from /usr/lib/libc.so.6
(gdb) backtrace
#0  0x00007fe1fd3db5f8 in raise () from /usr/lib/libc.so.6
#1  0x00007fe1fd3dca7a in abort () from /usr/lib/libc.so.6
#2  0x00007fe1fd3d4417 in __assert_fail_base () from /usr/lib/libc.so.6
#3  0x00007fe1fd3d44c2 in __assert_fail () from /usr/lib/libc.so.6
#4  0x0000000000400523 in main () at fail.c:5
(gdb)
```

By using `coredumpctl`'s utility to directly launch gdb on the coredump:

```
42sh$ coredumpctl gdb fail
    [...]
Reading symbols from /home/dettorer/work/temp/fail...done.
```

(continues on next page)

(continued from previous page)

```
[New LWP 22877]

warning: Could not load shared library symbols for linux-vdso.so.1.
Do you need "set solib-search-path" or "set sysroot"?
Core was generated by `./fail'.
Program terminated with signal SIGABRT, Aborted.
#0  0x00007fe1fd3db5f8 in raise () from /usr/lib/libc.so.6
(gdb) backtrace
#0  0x00007fe1fd3db5f8 in raise () from /usr/lib/libc.so.6
#1  0x00007fe1fd3dca7a in abort () from /usr/lib/libc.so.6
#2  0x00007fe1fd3d4417 in __assert_fail_base () from /usr/lib/libc.so.6
#3  0x00007fe1fd3d44c2 in __assert_fail () from /usr/lib/libc.so.6
#4  0x0000000000400523 in main () at fail.c:5
(gdb)
```

Keep in mind that while the latter is a quicker way, it does not save the coredump in the current directory. If you create more coredumps, it may be difficult to access this particular one later.

3.4 Reverse Debugging

GDB since its **7.0** version added a new interesting feature named *reverse debugging*, allowing to replay the execution path. Imagine: you are debugging step by step, watching variables you have interest in, and suddenly, a wild *segfault* appears! You would have to print the value a variable some lines above. But you need to run you program again....

With *reverse debugging*, you will just need to do some `reverse-next` commands to reverse the execution path, replaying the changes, to continue your debugging session.

Enabling *reverse debugging* capabilities is done with the `record` command. We ask GDB to record the changes made by our program so we can replay them if needed.

Tips

To be able to *record*, your program must be running.

```
42sh$ gdb debugme1
(gdb) break myfunc
Breakpoint 1 at 0x4005df: file debugme1.c, line 9.
(gdb) run
Breakpoint 1, myfunc ( ... ) at debugme1.c:9
9       int a = 2;
(gdb) record
```

If you want to `record` from the start of the `main` function, simply use `start` command, this will set a *temporary breakpoint* on `main` and then run the program.

You can then continue the execution with `continue`:

```
42sh$ gdb debugme1
(gdb) start
Temporary breakpoint 1, main (argc=1, argv=0x7fffffffdab8) at debugme1.c:19
19          uint16_t len = strlen(argv[0]);
```

(continues on next page)

(continued from previous page)

```
(gdb) record
(gdb) continue
```

Reverse debugging is now enabled. To exploit it, simply use the new `reverse-next` and `reverse-step`.

Be careful!

Be careful of the limitations of this system. Indeed, if your program does modifications to your system, like writing in a file or sending a network packet for example, GDB will not be able to reverse those changes.

record	Start recording execution
reverse-next	Reverse of *next*
reverse-step	Reverse of *step*
reverse-continue	Reverse of *continue*
reverse-finish	Reverse of *finish*
reverse-search	Reverse of *search*

Try by yourself with the `debugme3.c`.

3.5 Checkpoint

In addition to **reverse debugging** capabilities, GDB allows you to set **checkpoints** to save a snapshot of a program's state and come back to it later. Indeed, it is like going back in time to the moment when the checkpoint was saved. Constraints are the same as **reverse debugging**.

To set a `checkpoint`, make sure your program is running and just use:

```
(gdb) checkpoint
checkpoint 1: fork returned pid 6637.
```

To come back to a checkpoint, use the *restart <checkpoint_id>* command:

```
(gdb) restart 1
```

To get the list of set checkpoints, use `info checkpoints`, The star `*` indicates which process GDB is currently debugging.

```
(gdb) info checkpoint
  1 process 6637 at 0x4006bd, file debugme1.c, line 20
* 0 process 6399 (main process) at 0x4006bd, file debugme1.c, line 20
```

checkpoint	Set a *checkpoint* at the current location
restart <checkpoint_id>	Come back to <checkpoint_id>
info checkpoints	Get the list of set *checkpoints*

As usual try to practice with the `debugme3`.

I must not fear. Fear is the mind-killer.