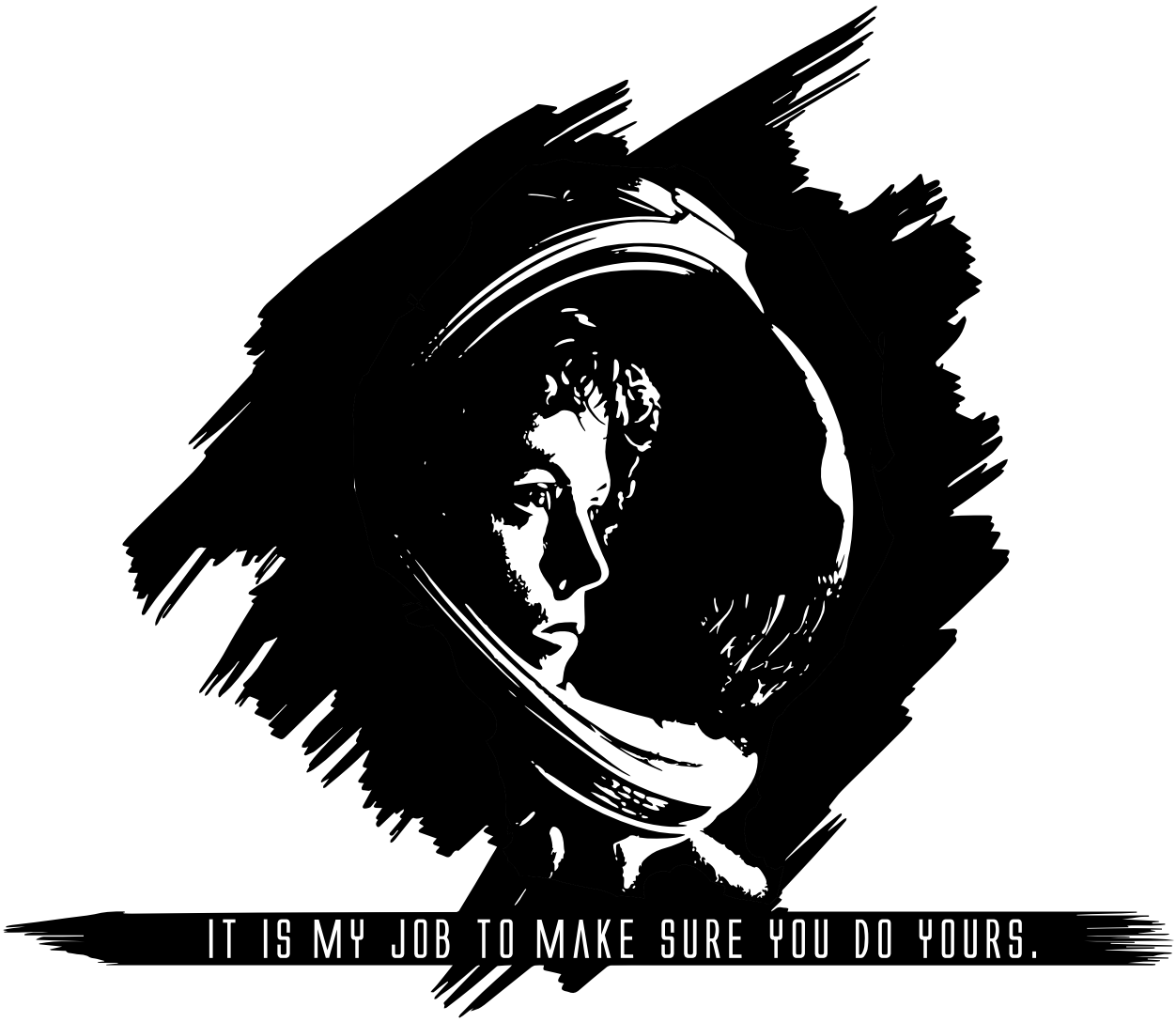
Piscine — Tutorial D3

version #



Assistants C/Unix 2022 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

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

1 Structures

1.1 Problem

Let's say we want to develop a function that takes two points in a plane and computes the distance between them. The prototype of the function could look like this:

```
float distance(float p1_x, float p1_y, float p2_x, float p2_y);
```

A point has only two coordinates and the prototype of this function is already very long. Imagine how long it would be if we were in three dimensions! Another issue is the lack of cohesion between values. For example, `p1_x` is technically not related in any way to `p1_y`. We need a way, to regroup related variables under one name we can use in our code.

At this point in time, the only types you have seen are **primitive** data types: they are types built in the language and can be used as basic data (integer, float...). You will see by practicing, that programming problems and concepts will become complex. So, the question is: *How to represent these concepts in our type system?*

1.2 Syntax

Let's go back to our cartesian point. One point is composed of:

- A float `x`, the field which represents the abscissa.
- A float `y`, the field which represents the ordinate.

In *C* , we can define a new type for this concept as follows:

```
/* A point is both: */
struct point
{
    float x; /* an "x" field...      */
    float y; /* ...and a "y" field  */
}; /* Don't forget the trailing semicolon! */

struct point p; /* p is a variable of type struct point */
p.x = 42;       /* The x field of p is 42 */
p.y = 51;       /* The y field of p is 51 */
```

Therefore, we have defined a new type, `struct point`, which can be used like any other type:

```
void point_print(struct point p)
{
    printf("This point is (%f, %f)\n", p.x, p.y);
}
```

Our method `distance` could then be written like this:

```
float distance(struct point p1, struct point p2);
```

The great force of the structure definition, is that we can use it to create even more derived types! For example, let's create a segment, composed of two `struct point`.

```c
struct segment
{
    struct point p1;
    struct point p2;
};
```

You can also declare a structure along with a variable of this type in one go using the following syntax:

```c
struct foo
{
    int bar;
    // other fields
} foo_var;
foo_var.bar = 42;
```

1.3 Size of a structure

It can be interesting to compute the size of a structure. For example, what is the size of this structure?

```c
struct trap
{
    int i;
    char c;
};

printf("%zu\n", sizeof(struct trap)); // What will this print?
```

At first we could say that the size of a structure is equal to the sum of its fields. Here, if we suppose that `sizeof(int) == 4`, the size of the structure will be 5 bytes (the size of a char is **always** one byte). Try this, and you will see that this is wrong!

Be careful!

You may notice that the size of this structure is actually 8. It is due to compiler's optimization that will favor memory access on *aligned addresses*. Thus, the compiler will add padding bytes to make fields' addresses aligned.

If it is not clear for you, just remember that **the size of a structure is at least the sum of its fields**, but can be greater for performance reasons.

1.4 Array of structures

Like we said in the first part, a structure is used to store information about one particular object. But if we need to have more structures of this particular object, then an array of structures is needed.

For example:

```c
struct student
{
    char name[24];
```

(continues on next page)

(continued from previous page)

```
    int age;
    int average_grade;
} students[100];
```

Here, students[0] stores the information of the first student, students[1] stores the informations of the second student and so on.

In order to access the age field of the third student, you just have to do:

```
int age = students[2].age
```

If you need to initialize an array of struct, it is done the same way as primitive types. Also, in a structure initializer, you can specify the name of a field by adding ".fieldname =" before the element value:

```
struct student
{
    char name[24];
    int age;
    int average_grade;
};

struct student students[] = {
    { .name = "Antoine", .age = 21, .average_grade = 19 },
    { .name = "Paul", .age = 21, .average_grade = 19 },
};
```

1.5 Exercise

The use of structures is simple. We advise you to take the next exercise seriously, as it will prove useful for the rest of your *C* studies.

1.5.1 Pairs

1.6 Goal

In this exercise, we will use a simple pair structure containing integers:

```
struct pair
{
    int x;
    int y;
};
```

Write the following functions:

```
struct pair three_pairs_sum(const struct pair pair_1, const struct pair pair_2,
                            const struct pair pair_3);
```

`three_pairs_sum` takes three `pairs` and sums each x field together, and each y field together. The summed x and y fields are stored into a new structure which is returned by the function.

```c
struct pair pairs_sum(const struct pair pairs[], size_t size);
```

`pairs_sum` takes an array of `size` `pairs` and sums each x field together, and each y field together. The summed x and y fields are stored into a new structure which is returned by the function.

2 Enumerations

2.1 Problem

Let us imagine you are coding a video game, and you wish to represent a direction in which a character is moving: it can go to the north, south, east, or west. How can we represent this?

Possible solutions here could be using either strings or numerical values to represent each direction, but these have inherent issues:

- If we choose to go with strings, we end up having to use much more memory than really needed to store every character in every possible string, and comparing strings to check if they are identical is non-trivial and takes time. Think about `strcmp`: in the worst-case scenario you have to go through the whole string to realize they differ at the last character. This might not be much, but if you do a lot of comparisons with multi-character strings, the number of operations will sum up over time.
- If we choose to go with numerical values, all these optimization concerns go away but at one cost: readability. Now you have to remember that 1 means "north", 2 means "west" and so on. Introducing such *magic values* makes the code harder to read, and if it is harder to read it is harder to debug and more prone to bugs. Magic values should be avoided at all cost.

Fortunately for us, the *C* language has what we need: **enumerations**.

2.2 Definition

Enumerations are a user defined data type which can be used to name arbitrary values in order to make code easy to read and maintain.

If we go back to our video game example, we could use an enumeration like this one:

```c
/* A direction is either: */
enum direction
{
    NORTH, /* this value    */
    SOUTH, /* OR this value */
    EAST,  /* OR this one   */
    WEST   /* OR that one   */
}; /* Again, trailing semicolon! */
```

Tips

As with macros, we usually name enumerations in capitals. It helps differentiate them from variables and function names.

Similarly to an integer having multiple possible values {0, 1, 2, 3, ...}, `enum direction` is a type that has four possible values: {`NORTH`, `SOUTH`, `EAST`, `WEST`}. Their usage is very simple:

```
void print_direction(enum direction dir)
{
    /* A switch is also commonly used to work with enums */
    if (dir == NORTH)
        puts("I'm facing north!");
    else if (dir == SOUTH)
        puts("I'm facing south!");
    /* ... */
}

int main(void)
{
    enum direction my_dir = SOUTH;
    print_direction(my_dir);
    return 0;
}
```

Enumerations are actually named integers so handling them is mostly like handling `int` values: comparing them takes no time, and they do not take much space memory-wise, with the added benefit that the code is clear and easily understandable.

The consequence is that when defining an `enum`, it is possible to assign specific numerical values to each enum value, which can be useful for instance when working with error codes.

For example, we could use the following enum to represent error codes for `ls`:

```
enum error_code
{
    OK = 0,
    MINOR_ERR = 1,
    MAJOR_ERR = 2
};

int main(void)
{
    enum error_code err = OK;

    // Main code goes here, change err value if needed...

    switch (err)
    {
    case OK:
       puts("No problem detected.");
       break;
    case MINOR_ERR:
       puts("Minor problem detected.");
       break;
    case MAJOR_ERR:
       puts("Major problem detected.");
       break;
    }
```

(continues on next page)

(continued from previous page)

```
    // Here the enum is implicitly converted to an int, which is the
    // function return type.
    return err;
}
```

Going further...

Enumerations actually serve quite a similar purpose to macros: avoiding magic values. That being said, when dealing with multiple possible values or states for a similar concept, one should prefer using enumerations over macros due to the added benefit of typing. Enumerations define a custom type which can be used to express more meaning in program, whereas macros are only a shortcut to a value and are not typed per se.

For example: when doing a switch on an enum, the compiler is smart enough to be able to determine whether all possible values are handled, even without having a default case. This proves quite useful as if we were to add a possible value to a given enum, all switches on this enum would trigger a warning until that new case is taken into account. This provides better code safety, and is really useful for maintaining code.

When no integer values are associated with the enumerations, the default value of the first one is guaranteed to be 0, and the following values are increments of their predecessors.

```
enum only_default
{
    A,
    B,
    C
};

enum non_default
{
    D,
    E = 41,
    F
};

int main(void)
{
    printf("%d\n", A); // prints 0, A has default value of 0
    printf("%d\n", B); // prints 1, B follows A so its value is 0 + 1 = 1
    printf("%d\n", C); // prints 2, C follows B so its value is 1 + 1 = 2
    printf("%d\n", D); // prints 0, D has default value of 0
    printf("%d\n", E); // prints 41, E has a specific value
    printf("%d\n", F); // prints 42, F follows E so its value is 41 + 1 = 42

    return 0;
}
```

Going further...

It is possible for two values of an enumeration to have the same numerical value. The following code is valid:

```c
enum my_enum
{
    ZERO,
    ONE,
    ALSO_ONE = 1
};

int main(void)
{
    printf("%d, %d, %d\n", ZERO, ONE, ALSO_ONE); // prints 0, 1, 1

    return 0;
}
```

3 Macro Genericity

3.1 What is genericity?

Generic code is code defining behaviour without taking specific types into account. A generic function behaves the same no matter the type of its arguments. This allows to write an algorithm once that will work for every possible type as input, which greatly reduces the need for code duplication.

Although generic code is arguably not the main focus of the *C* language, there are a few ways to write generic code in *C*, which we will see progressively.

The first and most straight-forward way is using macros.

3.2 Macro expansion

The main mechanism behind macros is called *macro expansion*. The idea is that macros will be replaced by their value at compile-time. Although this is already useful to eliminate *magic values* as you have seen earlier this week, this also allows to write generic code thanks to parameter macros.

When a macro with parameters is expanded, its whole body is inserted where it is called, no matter the type of its parameters. Hence, the same macro can be used for multiple parameter types: it is **generic**.

3.3 Macro with parameters

Macros with parameters, often called function-like macros, can be defined using the following syntax:

```c
#define MACRO_NAME([[Parameter, ]* Parameter]) replacement-text
```

When the preprocessor expands such a macro, it incorporates the arguments you specify. Sequences of whitespaces before and after each argument are not part of the argument.

The parameter list is a comma-separated list of identifiers for the macro's parameters. When you use a function-like macro, you must use as many arguments as there are parameters in the macro definition.

Moreover, make sure that there is no whitespace between the macro's name and the `(`. If there is a whitespace, it will be expanded as a macro without parameters.

As the following examples illustrate, you should generally enclose your parameters and your replacement text in parentheses.

```c
#define SUM ((2) + (2))
#define UNSAFE_SUM (2) + (2)

int foo = 10 * SUM;        // foo = 10 * ((2) + (2)) = 40
int bar = 10 * UNSAFE_SUM; // bar = 10 * (2) + (2) = 22
```

```c
#define MULT(A, B) ((A) * (B))
#define UNSAFE_MULT(A, B) A * B

int foo = MULT(10 + 1, 10);        // foo = ((10 + 1) * (10)) = 110
int bar = UNSAFE_MULT(10 + 1, 10); // bar = 10 + 1 * 10 = 20

float baz = MULT(-3.0, 2.0); // this generic macro can be used with multiple types
```

3.3.1 Macro operators

3.4 Goal

You must write to a file named `macro.h` the following macros:

```c
#define MIN(A, B) ...
#define MAX(A, B) ...
```

3.5 Macro with parameters (cont.)

Moreover, a *function-like* macro is not a function. It is still replacement text. Therefore, be careful of calls with side-effects. For example, take the absolute function-like macro.

```c
#define ABS(A) (((A) < 0) ? -(A) : (A))

int main(void)
{
    int i = -42;
    int j = ABS(++i);  // j = 40
}
```

As the macro is just text replacement, `++i` is called twice. This is not the same behavior as a function.

Be careful!

The point of macros is **not** to make your code faster. Although depending on the context they *might* make it faster, this will not always be the case, depending on how the compiler optimizes your code. Macros do come with disadvantages, mostly that they are not strongly typed, which is useful when you need genericity but can make them hard to debug.

Whenever possible, always prefer writing functions over writing function-like macros.

Going further...

C programmers often write `static inline` functions in header files as a substitute for function-like macros. You will see this in more detail later on.

4 GDB(1)

4.1 Introduction

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.

4.2 Basics

4.2.1 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`

```
/*
** debugme1.c
*/
#include <stdio.h>
#include <string.h>
```

(continues on next page)

(continued from previous page)

```c
#include <stdint.h>

void reverse(char input[], uint16_t index)
{
    if (index >= 0)
    {
        putchar(input[index]);
        reverse(input, --index);
    }
    else
        putchar('\n');
}

int main(void)
{
    char str[] = "foobar";
    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
```

4.2.2 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.

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.
```

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
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.

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.

4.2.3 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.

run

Let's run our program with the `run` command. It can take the arguments that you would normally give the program. If you wanted to give the arguments when launching `gdb`, 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 "foobar", index=65535) at debugme1.c:12
12          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 "foobar", index=65535)`
- Location in the source code, file and line: `at debugme1.c:12`
- The corresponding line of code: `12 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.

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`.

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 "foobar", index=65535) at debugme1.
↪c:12
#1  0x000055555555519a in reverse (input=0x7fffffffe7b1 "foobar", index=65535) at debugme1.
↪c:13
#2  0x000055555555519a in reverse (input=0x7fffffffe7b1 "foobar", index=0) at debugme1.c:13
#3  0x000055555555519a in reverse (input=0x7fffffffe7b1 "foobar", index=1) at debugme1.c:13
#4  0x000055555555519a in reverse (input=0x7fffffffe7b1 "foobar", index=2) at debugme1.c:13
#5  0x000055555555519a in reverse (input=0x7fffffffe7b1 "foobar", index=3) at debugme1.c:13
#6  0x000055555555519a in reverse (input=0x7fffffffe7b1 "foobar", index=4) at debugme1.c:13
#7  0x000055555555519a in reverse (input=0x7fffffffe7b1 "foobar", index=5) at debugme1.c:13
#8  0x00005555555551f1 in main (argc=1, argv=0x7fffffffe8a8) at debugme1.c:23
```

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.

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 8`, you will walk up the call stack and position gdb's view point in frame 8 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 8
#8  0x00005555555551f1 in main (argc=1, argv=0x7fffffffe8a8) at debugme1.c:23
23      reverse(str, len);
(gdb) info locals
str = "foobar"
len = 6
```

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 "foobar", index=65535) at debugme1.
↪c:12
No locals.
#1  0x000055555555519a in reverse (input=0x7fffffffe7b1 "foobar", index=65535) at debugme1.
↪c:13
```

(continues on next page)

(continued from previous page)

```
No locals.
[...]
#8  0x00005555555551f1 in main (argc=1, argv=0x7fffffffe8a8) at debugme1.c:23
    str = "foobar"
    len = 6
```

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.

Application

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

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.

4.3 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.

4.3.1 break

A *breakpoint* is a marker placed at a specific location in the code where 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 0x4005e8: file debugme1.c, line 12.
```

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 "foobar", index=6) at debugme1.c:12
9           putchar(input[index]);
```

4.3.2 Print information

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

```
(gdb) print index
index = 6
```

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

```
(gdb) info args
input = 0x7fffffffe330 "foobar"
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 stop (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   0x0000000000400652 in reverse at debugme1.c:12
        breakpoint already hit 1 time
2       breakpoint     keep y   0x00000000004006a1 in main at debugme1.c:22
3       breakpoint     keep y   0x0000000000400652 in reverse at debugme1.c:11
```

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

4.3.3 Application

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

```c
/*
** debugme2.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;
}
```

Use your new skills to debug `debugme2`.

Tips

- You can see the ways of setting a *breakpoint* by typing `help break`.
- You can get the backtrace at any time during execution, not only when your program crashed.
- You should try `help breakpoints` to see few useful commands to control the flow of your program when using breakpoints.
- If you want to navigate in a much more fancy view with you source code and position at the top of the window and the command line at the bottom, just type the `tab` command. In `tab` mode, you can press `<C-x>o` to switch focus between the code window and the command line window.

- If you have many breakpoints set and do not want to exit `gdb`, recompile, re-start `gdb` with the new executable, and reset all the breakpoints, you can invoke `make` within `gdb` to rebuild the executable. However, some of your breakpoints may not be where you think they are if you added or removed lines of your code.

It is my job to make sure you do yours.