Piscine — Tutorial D7

version **#v3.2.0**

Assistant C/UNIX 2021 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

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

1 Structures & Enum

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 informations 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 Enumerations

Enumerations are an other kind of derived type, which let you name arbitrary values. For example, if you code 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?[1]

The simple way is to use integers, but which values will you give? We do not care, the only requirement is that the four values are different.

To represent this concept, we can use an enumeration like this one:

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

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:

[1] Forget strings: they are inefficient and not suited for this.

```c
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;
}
```

1.6 Exercise

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

1.6.1 Pairs

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

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

Write the following functions:

```c
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 with 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 GDB(1)

2.1 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.

2.2 Basics

2.2.1 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`

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

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

(continues on next page)

(continued from previous page)

```
    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` or `make debugme1 CFLAGS=-g` and then run it. It crashes.

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

2.2.2 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 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
```

(continues on next page)

(continued from previous page)

```
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.

2.2.3 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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
#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
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.

3 Memory by allocation

3.1 Reminder

You have already seen the basics of dynamic memory allocations in a previous tutorial. We will now explain the different types of memory, and present two other functions of the `C` standard library relating to memory allocation: `calloc(3)` and `realloc(3)`.

3.2 Types of memory

When a `C` program is executed, it can store data in different memory areas. These memory areas belong to different **types of storage**, which are defined by the language standard and have an associated **lifetime**.

There are three different types of storage:

- Static memory: where global variables are stored, or variables defined with the `static` keyword. The static memory persists for the lifetime of the program.

- Automatic memory: allocated on the **stack**, and its lifetime depends on specific scopes (function body, loops, ...).
- Dynamic memory: allocated on the **heap**, whose lifetime is that of the program itself unless explicitly released.

3.3 `calloc(3)`

When allocating memory for an array, you may need a memory chunk initialized to zero, thus setting all elements in your array to zeros. This is what `calloc(3)` does; like `malloc(3)`, it allocates a memory area, but it is filled with zeros. Like `malloc(3)`, the pointer returned will be `NULL` if the memory cannot be allocated, see `man 3 calloc` for more details.

```c
// create an array of ten integer. All elements are set to zero.
int *array = calloc(10, sizeof(int));

if (!array)
    /* handle the error */

for (int i = 0; i < 10; ++i)
    printf("%d ", array[i]);

free(array);
```

The prototype of `calloc(3)` is a bit different from `malloc(3)`. Where `malloc(3)` takes only the size of the whole block as parameter; `calloc(3)`, takes the number of members of an array and the size of each member.

3.3.1 Application: my calloc

```c
void *my_calloc(size_t n, size_t size);
```

Returns a pointer to an allocated memory area capable of holding `n` elements of `size` bytes. The whole memory must be set to zero. Returns `NULL` if the allocation failed. The returned pointer will be freed later using `free(3)`.

Going further...

Keep in mind that `calloc(3)` is way more than a simple function that calls `malloc(3)` and fill the area with *0s*[1].

[1] https://vorpus.org/blog/why-does-calloc-exist/

3.4 realloc(3)

Once you allocated a memory area, you may need to resize it later. For example if your memory is used to hold elements of a dynamic vector, you want to expand this memory when your vector is full, or reduce it when there is only a few elements in it. `realloc(3)` will give you the ability to resize a given memory zone. It takes two parameters: the pointer to an area you already allocated, and the new size. It returns a pointer to the new area.

Be careful!

If the new size is bigger than the previous one, `realloc(3)` may not be able to expand the current region, thus it will allocate another bigger chunk of memory, copy the previous elements in this new memory area, then free the old memory area. This is why `realloc(3)` returns a pointer to a new area, if it was not able to expand the current one.

Tips

`realloc(3)` may return `NULL` if the allocation of the new area failed. According to the man page, the previous area is left untouched in case of failure. Therefore, if you reassigned your pointer to the value returned by `realloc(3)`, you lost the previous pointer and won't be able to free it. The best way to avoid this leak is to use a temporary pointer and to check it before reassigning it to your initial pointer.

```
size_t size = 10;
int *array = calloc(size, sizeof(int)); // create an array of ten integer

if (!array)
    /* handle the error and leave */

/* do some stuff with your array */

size_t new_size = (size * sizeof(int)) * 2;
int *new = realloc(array, new_size);

if (!new)
    /* reallocation failed */
else
    array = new; /* reallocation succeeded, array has been freed */
/* if the allocation succeeded, old values are still there */

free(array);
```

3.4.1 Exercises

3.4.2 my_free

Using `man 3 realloc`, implement an equivalent to `free(3)` function without calling `free(3)` directly.

3.4.3 my_malloc

Using `man 3 realloc`, implement an equivalent to `malloc(3)` function without calling `malloc(3)` directly.

3.4.4 realloc changes

Allocate a memory chunk of the size of your choice, then display the address using `printf(3)` with `%p`. Inside a loop, expand the memory area by one using `realloc(3)` and display the new address.

What is happening?

4 Assert

The `assert(3)` **macro** `void assert(int expression);` allows you to perform checks in your program. This macro will always be allowed and can be really useful, so it is in your best interest to know how to use it properly.

Assert is defined in the `assert.h` header, and you can use it as shown below.

- **File:** `assert/example1.c`

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

int div(int a, int b)
{
    assert(b != 0);
    return a / b;
}

int main(void)
{
    printf("div(10, 2) = %d\n", div(10, 2));
    printf("div(5, 0) = %d\n", div(5, 0));
    return 0;
}
```

Let's run the above program. It will abort if the condition is not met.

```
42sh$ gcc -g -Wall -Wextra -Werror -pedantic -std=c99 example1.c -o example1
42sh$ ./example1
div(10, 2) = 5
example1: example1.c:6: div: Assertion `b != 0' failed.
[1]    10722 abort (core dumped)  ./example1
```

The output message is really self-explanatory:

- `abort (core dumped) ./example1` means that the program `./example1` actually aborted.
- `example1: example1.c:6: div:` provides you with the exact location of the crash: in the `example1` process, at line 6 in the `example1.c` file and in the `div` function.

- finally `Assertion 'b != 0' failed.` directly displays the failed assertion.

The condition in the assert statement is printed when the assert fails. You can play with it to get some additional information displayed in the error message. For example, add `'&& mystring'` to your condition:

- **File:** `assert/example2.c`

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

int fact(int n)
{
    int res = 1;

    assert(n >= 0
           && "The factorial function is only defined for positive numbers");
    while (n)
        res *= n--;

    return res;
}

int main(void)
{
    printf("fact(5) = %d\n", fact(5));
    printf("fact(-2) = %d\n", fact(-2));
    return 0;
}
```

```
42sh$ gcc -g -Wall -Wextra -Werror -pedantic -std=c99 example2.c -o example2
42sh$ ./example2
fact(5) = 120
example2: example2.c:9: fact: Assertion `n >= 0 && "The factorial function is only defined
↪for positive numbers"' failed.
[1]    8140 abort (core dumped)  ./example2
```

4.1 Warning

As explained in the `assert(3)` manual, the macro does nothing when `NDEBUG` is defined (with `gcc -DNDEBUG`). This feature enables avoiding the check *overhead* once your code is delivered.

Tips

It should be set in the `CPPFLAGS` variable in your Makefile.

Be careful not to have side effects in your assert statement!

- **File:** `assert/example3.c`

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

```

(continues on next page)

(continued from previous page)

```
int main(void)
{
    int i = 1;
    assert(i++ > 0);
    assert(i++ > 1);
    assert(i++ > 2);
    printf("%d\n", i);

    return (0);
}
```

```
42sh$ gcc -g -Wall -Wextra -Werror -pedantic -std=c99 example3.c -o example3
42sh$ ./example3
4

42sh$ gcc -g -Wall -Wextra -Werror -pedantic -std=c99 example3.c -o example3 -DNDEBUG
42sh$ ./example3
1
```

5 Macro in C

You have already discovered the basics of preprocessing with various preprocessor instructions such as includes, definitions and conditions. Today, you will learn more about preprocessing with the use of *macro*.

Tips

Reminder: to see the gcc preprocessor output you can use the option `-E`.

5.1 Define and use

You have already seen that you can define macros in `C` using the preprocessor directive `#define`. It allows you to give a name to any text (constant, statement, …). Macros are most commonly used to name numeric constants such as:

```
#define ARRAY_SIZE 2048
int arr[ARRAY_SIZE];
```

These two lines define the macro `ARRAY_SIZE` for the number 2048 and then use this macro for the definition of `arr`.

Tips

By convention macros are named in capitals. It helps to make the difference with variables and function names.

The use of a macro for a numeric constant is important to make your code more flexible, sustainable and readable. Assume that the size of an array such as `arr` is used several times in your program (loop, other array declaration, …). The use of the macro named `ARRAY_SIZE` instead of a number is more readable, and if you need to change the size of your array, you only need to change the number in one place and not everywhere it is used.

Macros are a simple way to implement text replacements. They are very simple, but also very powerful (and sometimes dangerous). During the preprocessing step, all macro names are *expanded*. This means that the macro's name is replaced with the text it has been defined to replace.

Tips

Macro names that appear in string literals are not expanded because a string literal is itself a preprocessor token.

In this section you have discovered the basic use of macros, be aware that you can do a lot more.

5.1.1 Macro first use

In this exercise, you should use a macro to define the size of an array in order to understand how it can be useful.

You must create an array of size `N`, assign the value `N` modulo `i + 1` to the element at index `i`, and print it.

5.2 Macro without parameters

To define a macro with no parameters, the syntax is really simple:

```
#define MACRO_NAME replacement-text
```

Whitespace characters before and after the `replacement-text` are not part of the replacement text.

Note that you can define macros on multiple lines.

```
#define MACRO_NAME This is a \
    very long text
```

As a macro is just text replacement, the whitespace between `a` and `\`, and between the newline and `very` are part of the replacement text.

Tips

Reminder:

Use a macro instead of a bare integer (also called magic number) to name the value being used.

5.3 Macro with parameters

You can also define macros with parameters. They are often called function-like macros, and you can define them using the following syntax:

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

When the preprocesser expands such a macro, it incorporates the arguments you specify. Sequences of whitespace 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.

```
#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
```

```
#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
```

5.3.1 Macro operators

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

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

5.4 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.

```
#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!

Macros don't make your code faster. Whenever possible, always prefer writing functions over writing function-like macros.

C programmers often write `static inline` functions in header files as a substitute for function-like macros.

5.5 Static assert

You have already seen `assert` that performs checks at runtime. What if you wanted these checks to be run at compile time? There are multiples ways of doing it.

Here is a small example of how you could check a condition at compile-time with a macro.

```
#define STATIC_ASSERT(Cond) switch(0){case 0:case (Cond):;}

int main(void)
{
    STATIC_ASSERT(sizeof(int) == 5);
}
```

```
42sh$ gcc -g -Werror -Wall -Wextra -pedantic -std=c99 example.c -o example4
example.c: In function ‘main’:
example.c:1:46: error: duplicate case value
    1 | #define STATIC_ASSERT(Cond) switch(0){case 0:case (Cond):;}
      |                                              ^~~~
example.c:5:5: note: in expansion of macro ‘STATIC_ASSERT’
    5 |     STATIC_ASSERT(sizeof(int) == 5);
      |     ^~~~~~~~~~~~~
example.c:1:39: note: previously used here
    1 | #define STATIC_ASSERT(Cond) switch(0){case 0:case (Cond):;}
      |                                       ^~~~
example.c:5:5: note: in expansion of macro ‘STATIC_ASSERT’
    5 |     STATIC_ASSERT(sizeof(int) == 5);
      |     ^~~~~~~~~~~~~
```

If you now change the code above with this `STATIC_ASSERT(sizeof(int) == 4);` it will compile without any problems.

How does it work?

It uses one of the particularities of `switch`. Each `case` must be defined only once. So when the condition is false, the value is 0 and it produces an error since the case 0 is already defined.

Note that it works with `sizeof(int) == 5` because its value is resolved at compile-time. It would also have worked with `3 == 2`, but it would not work with `b == 2` because the value of a variable `b` is not known at compile-time.

5.6 The `stringify` operator

The unary operator `#` is called the `stringify` operator because it converts a *macro argument* into a string. If a parameter appears with a prefixed `#`, the preprocessor places the argument between `"`, with slight changes:

- Any sequence of whitespace characters between tokens in the argument value is replaced with a single space character.
- A \ is prefixed to each `"` in the argument.
- A \ is prefixed to each \ that occurs in the argument, except if it introduces a universal character (ex: `\u03B1`).

```
#define PRINT_EXP(Exp) printf(#Exp " = %lf\n", Exp)

PRINT_EXP(     4.0     *    10.0      );
```

The previous block is expanded as:

```
printf("4.0 * 10.0" " = %lf\n", 4.0 * 10.0);
```

5.7 The token-pasting operator

The binary operator `##` joins its left and its right operands together into a single token. Whitespace characters that appear before and after `##` are removed along with the operator itself.

```
#define JOIN_INT(A, B) A ## B

JOIN_INT(123   ,    456)
```

The previous block is expanded as:

```
123456
```

Be aware that if you want to append 2 strings, the `##` cannot be used directly. And if you need to mix `#` and `##`, you cannot do:

```
#define JOIN_STR(A, B) #A ## #B
```

This creates 2 strings and then tries to append the strings, thus it fails. You need to do it like this:

```
#define STR(A) #A
#define JOIN_STR(A, B) STR(A ## B)
```

5.7.1 Macro declare and set

Create a macro `DECLARE_AND_SET(TYPE, NAME, VALUE)`, that declares and sets 3 variables.

- the first one must be a variable with the given type, name and value.
- the second one must be a pointer on the first variable, with the same name as the first one, preceded by *ptr_*.
- the third one must be a string of the given value, with the same name as the first variable, preceded by *str_*.

`DECLARE_AND_SET(int, foo, 42);` should create `int foo = 42;`, `int *ptr_foo = &foo;` and `char *str_foo = "42";`

5.8 Macros using macros

After argument substitution and execution of the `#` and `##` operations, the preprocessor examines the resulting replacement text and expands any macros it contains. But macros cannot be expanded recursively.

```
#define TEXT_1 "Hello"
#define MSG(A) puts(TEXT_ ## A)

MSG(1);
```

The previous block is expanded as:

```
puts("Hello");
```

5.9 Scope and redefinition

You cannot declare two macros with the same name, unless the new replacement text is identical to the existing macro definition. If the macro has parameters, the new parameter names must also be identical to the old ones.

The following sample is valid.

```
#define BAR(X) X
#define FOO(X) X
#define BAR(X) X
```

But not this one.

```
#define BAR(X) X
#define FOO(X) X
#define BAR(x) x
```

However, it can happen that you need to change the meaning of a macro (but it is not a good idea). To change the meaning of a macro, you first need to cancel its current definition using the following directive:

```
#undef MACRO_NAME
```

After that, the identifier `MACRO_NAME` is available for a new macro definition. If `MACRO_NAME` is not a name of a macro, the preprocessor ignores the directive.

The scope of a macro ends with the first `#undef` directive with its name, or if no `#undef` are found, it ends at the end of the translation unit in which it is defined.

5.10 Predefined macros

Compilers that conform to the ISO C standard must define the following macros (and some others). Each of these macro names begins and ends with two underscore characters:

- __DATE__

  The replacement text is a string literals containing the compilation date in the format “Mmm dd yyyy” (example: Sep 13 2017). If the day of the month is a single-digit number, an additional space character fill the empty space.

- __FILE__

  A string literal containing the name of the current source file.

- __LINE__

  An integer constant whose value is the number of the line in the current source file that contains the __LINE__ macro reference, counting from the beginning of the file.

- __TIME__

  A string literal that contains the time of compilation, in the format “hh:mm:ss” (example: “00:04:02”).

Tips

Certain macros are predefined only under certain conditions (example: __STDC_NO_COMPLEX__ defined to 1 if the implementation does not support complex numbers, that is if the header complex.h is absent).

Going further...

You can define macros during compilation aswell with the `-D` flag:

```
gcc -Dwhile=if main.c -o main
```

It can actually be useful with other preprocessor directives like `#ifdef`

```
// some code
#ifdef DEBUG
  printf("I'm debuging!\n");
#endif
// some code
```

```
42sh$ gcc -DDEBUG main.c -o main
42sh$ ./main
   ...
I'm debuging!
   ...
```

6 Address Sanitizer

6.1 What is it?

AddressSanitizer aka **ASan** is a memory error detector tool. It can only run upon programs built with a dedicated compiler option.

You must compile and link your code with the `-fsanitize=address` flag to enable **AddressSanitizer**. It has been available since **GCC** 4.8 and **CLANG** 3.1.

```
CFLAGS += -fsanitize=address
LDFLAGS += -fsanitize=address
```

When **ASan** is enabled, **GCC** will link your binary to `libasan.so`. It will replace the `malloc(3)` family of functions, among other things.

In addition to replacing standard library functions, **ASan** performs compiler instrumentation: it adds safety checks to your code during compilation. When a check fails, an error message is displayed.

For further informations refer to the official documentation on https://github.com/google/sanitizers/wiki/AddressSanitizer.

6.2 What does it detect?

ASan can detect several kind of errors such as:

- variable used after free
- variable used after return
- variable used after scope
- heap buffer overflow
- stack buffer overflow
- global buffer overflow
- memory leaks
- initialization order bugs (Those only concern **C++**)

6.3 When to use it?

ASan should always be activated during the development of a project.

It brings massive error detection capabilities for little performance cost.

Be careful!

Because it makes your code slower, **ASan** must not be activated when your code is in production. It is a debug feature.

6.4 How to use it?

Most error detection features are enabled by default, therefore, you just have to compile your code with `-fsanitize=address` and launch it. In some cases, you must set the variable `ASAN_OPTIONS` before launching your binary in order to enable some additional checks. It is the case for "variable used after return" detection; you must set it to `detect_stack_use_after_return=1` - see below:

```
42sh$ gcc -fsanitize=address -g use_after_return.c -o use_after_return
42sh$ ASAN_OPTIONS='detect_stack_use_after_return=1' ./use_after_return
```

6.5 Usage example

Here an example of the output of **ASan** in case of a heap overflow.

```c
#include <stdlib.h>

#define SIZE 100

int main(void)
{
    int *array = malloc(SIZE * sizeof (int));
    int res = array[SIZE];
    free(array);
    return 0;
}
```

```
42sh$ gcc -fsanitize=address -g heap_overflow.c -o heap_overflow
42sh$ ./heap_overflow
=================================================================
==18295==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x61400000ffd0 at pc␣
↪0x0000004007c9 bp 0x7ffc1f4cacf0 sp 0x7ffc1f4cace0
READ of size 4 at 0x61400000ffd0 thread T0
    #0 0x4007c8 in main /home/assistant/address_sanitizer/heap_overflow.c:8
    #1 0x7f3f92ae782f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
    #2 0x4006a8 in _start (/home/assistant/address_sanitizer/a.out+0x4006a8)

0x61400000ffd0 is located 0 bytes to the right of 400-byte region [0x61400000fe40,
↪0x61400000ffd0)
allocated by thread T0 here:
```

(continues on next page)

(continued from previous page)

```
    #0 0x7f3f92f29602 in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x98602)
    #1 0x400787 in main /home/assistant/address_sanitizer/heap_overflow.c:7
    #2 0x7f3f92ae782f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)

SUMMARY: AddressSanitizer: heap-buffer-overflow /home/assistant/address_sanitizer/heap_
↪overflow.c:8 main
Shadow bytes around the buggy address:
  0x0c287fff9fa0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9fb0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9fc0: fa fa fa fa fa fa fa fa 00 00 00 00 00 00 00 00
  0x0c287fff9fd0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c287fff9fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c287fff9ff0: 00 00 00 00 00 00 00 00 00 00[fa]fa fa fa fa fa
  0x0c287fffa000: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fffa010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fffa020: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fffa030: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fffa040: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Heap right redzone:      fb
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack partial redzone:   f4
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
==18295==ABORTING
```

Most of the time, you will only be interested by the error description presented in the `ERROR` paragraph. You will find additional information in `SUMMARY` paragraph.

Find below the explanation of the report:

`array` points to a heap allocated space of 400 bytes size: range address [0x61400000fe40 ; 0x61400000ffd0 [.

`res` is assigned to the value pointed to by `array + 100` which address is 0x61400000ffd0. However this value is out of bounds, that's why the error is reported.

Be careful!

Your code will be systematically corrected with the address sanitizer.

If you don't use it for your projects, you will lose points and make easily avoidable mistakes.

The only way out is through