Piscine — Tutorial D4

version #

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

Copyright

Contents

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

1 Makefiles

1.1 Context

For now, in order to compile your program you did something like:

```
42sh$ gcc -std=c99 -Wall -Wextra -Werror -pedantic hello_world.c -o hello_world
```

We can all agree that this is inconvenient and fastidious to type[1].

When working on actual projects, you will also have more than one source file, and quickly, your simple one line command can become very long and if you want to move a file it will be complex to edit[2].

And another problem that you will see soon enough: compiling a big project can take a lot of time and be boring. You do not want to recompile the whole project each time you edit a file.

To summarize, we need something that can solve these problems:

- We do not want to type a long command.
- We want to be able to edit our compilation command easily.
- We do not want to recompile our project each time we edit a file.
- We want other people to easily compile our project.

1.2 What is make?

In response to those issues, the POSIX standard contains a tool used to handle project builds: `make`. The implementation of `make` that we will use this semester is GNU make which was released by the FSF as part of the GNU project. `GNU make` implements all the features defined in the POSIX standard for `make` and has some nice extensions as well.

`make`, is a tool used to simplify the task of building programs composed of many source files. It can be configured through a file named `Makefile`, `makefile`, or `GNUmakefile`. `make` will first search for a `GNUmakefile`, then `makefile`, and finally `Makefile` if it did not find the previous ones. However, it is recommended in the official GNU/Make documentation to call your makefile `Makefile`, and we expect you to follow this convention for your submissions.

`make` parses rules and variables from the `Makefile` which are mainly used to build the project. `make` will only re-build things that need to be re-built by comparing modification dates of targets with their dependencies.

A `Makefile` typically starts with a few variable definitions, followed by a set of target entries. Each variable from the `Makefile` is used with `$(VARIABLE)` and can be declared at the top of the file as:

```
VARIABLE = VALUE
```

Finally, each rule follows the convention:

[1] And do not talk to us about your `.bash_history`, it is equally fastidious to search the command line you want.
[2] `fc(1)` is out of the discussion as well.

```
target: dependency1 dependency2 ...
    command
    ...
```

A target is usually the name of the file generated by the Makefile rule. The most common examples of targets are executables or object files. It can also be the name of an action to carry out, for example `all`, `clean`, ...

A dependency (or "prerequisite") is a file that needs to exist in order to create the target. A target can also be a dependency for another target. When evaluating a rule, `make` will analyze its dependencies and if one of them is a target of another rule, `make` will evaluate it too before the one it depends on.

A command (or "recipe") is an action that `make` carries out. Be careful, each command is preceded by a **tabulation** (`\t`). Otherwise, your `Makefile` will not work.

1.3 Invocation

1.3.1 Basics

Let us start easy. First, you need a `hello_world.c` like the following one:

```c
#include <stdio.h>

int main(void)
{
    puts("Hello World !");
    return 0;
}
```

And a `Makefile`:

```
CC=gcc
CFLAGS=-std=c99 -Wall -Wextra -Werror -pedantic

hello_world: hello_world.o
	$(CC) -o hello_world hello_world.o

hello_world.o: hello_world.c
	$(CC) $(CFLAGS) -c -o hello_world.o hello_world.c
```

Be careful!

Remember that the `command` part of a `rule` **must** start with a tabulation.

Now try:

```
42sh$ make hello_world
gcc -std=c99 -Wall -Wextra -Werror -pedantic -c -o hello_world.o hello_world.c
gcc -o hello_world hello_world.o
```

So what is going on here?

First you have two `variables`:

- `CC`, for the *C* Compiler
- `CFLAGS`, for the *C* Flags

Then you have two `rules`. Those are the core of a `Makefile`. Let us look at the first one.

First we want to produce a binary called `hello_world`, this is our target.

In order to build our binary, we need the object `hello_world.o`, we say that our target `hello_world` depends on the existence of the object `hello_world.o`. Thus `hello_world.o` is a dependency of the target `hello_world`.

Now that our `rule` specifies the target it produces and what it depends on, we need to specify how it should be produced. Remember our `variables`? We will need them now. In order to `expand`[3] a Makefile `variable`, you need to put it between parentheses[4] with a dollar before it.

So if we summarize this rule, it will produce the target `hello_world` that depends on the object `hello_world.o` using the following command line:

```
gcc -o hello_world hello_world.o
```

But the object `hello_world.o` does not exist! This is why we have the second rule! If a dependency does not exist, `make` will search for a rule that can produce it and execute it before the first.

The second rule follows the same principle. You should be able to understand what it does by your own.

Going further...

The default `rule` called when typing `make` is the first one of the file. Thus, here, typing `make` or `make hello_world` will have exactly the same behavior.

If you run `make` again without updating any file, you will have a message like this:

```
42sh$ make hello_world
make: 'hello_world' is up to date.
```

If a target already exists, `make` will rebuild it if its dependencies do not exist or if they have been updated since the last build.

Going further...

You can force `make` to rebuild all targets with the option `-B`.

If you modify the `hello_world.c` and retry `make`, it will rebuild everything.

Another interesting thing to note is `make`'s `-n` option which asks `make` to print the commands it would have run instead of running them.

```
42sh$ cat Makefile
all:
    echo toto
42sh$ make
echo toto
```

(continues on next page)

[3] The `expansion` is the concept of replacing a variable by its value before interpreting the line.
[4] You might see braces instead of parentheses, they work the same. It is as you prefer.

(continued from previous page)

```
toto
42sh$ make -n
echo toto
```

2 The sieve of Eratosthenes

2.1 Makefile

You must provide a makefile with a `all` rule, that must compile your `sieve.c` and produce the `sieve.o` object file.

2.2 Goal

The goal here is to create a sieve of Eratosthenes. It is a simple iterative sieve used to find prime numbers.

It works with an array of numbers from `2` to `n`. The first number, `2`, is identified as a prime number. Then, all multiples of `2` are marked. When encountering a new unmarked number, it is identified as a prime number, and its multiples are marked.

You must write a function taking an integer `n` and printing the list of all prime numbers from `2` to `n`, included.

Prototype:

```c
void sieve(int n);
```

If `n` is lower than `2`, the function will not print anything. This will not be tested with `n` greater than `1000`. The `sieve.c` file must not include a `main` function. You may use a `main.c` file for your tests.

2.3 Example

You can use a `main` to test your program, but it must not be included in your submission. Calling the `sieve` function with the first program argument as parameter, will produce the following output:

```
42sh$ ./sieve 11 | cat -e
2$
3$
5$
7$
11$
42sh$ ./sieve 0 | cat -e
42sh$ ./sieve 42 | cat -e
2$
3$
5$
7$
```

(continues on next page)

(continued from previous page)

```
11$
13$
17$
19$
23$
29$
31$
37$
41$
```

Be careful!

You must **not** include a `main` function.

3 Data structures - Linked lists

3.1 Explanation

Although structures and pointers are useful by themselves, they become even more powerful when tied together. Since a pointer is a value like any other with a fixed size, it can be used as a field in a structure definition.

```c
struct my_struct
{
    int a_field;
    float *another_field; // why not?
};
```

However, it is impossible for a structure to refer to itself.

```c
struct impossible
{
    int foo;
    struct impossible bar; // Impossible!
};
```

To understand why, ask yourself the question: what would the size of this structure be? To get it, it must determine the size of its own fields, but to get the size of `bar` we will need the size of the structure itself.

However, nothing prevents us from using a pointer to a struct of the type being defined because the size of a pointer is known and is always the same.

```c
struct possible
{
    int foo;
    struct possible *bar; // Totally possible: we know the size!
};
```

We just saw a new powerful concept: **recursive** structure definitions, which use a pointer to an instance of the structure in its definition.

A simple example of this concept is the implementation of a singly linked list.

A singly linked list is either:

- an empty list
- an element followed by a list

3.2 Structure declaration

How to represent a linked list in *C*? Let us consider alternatives:

- If a list is empty, it does not hold any information and thus it is useless to allocate memory for it: just use `NULL` to represent an empty list.
- Otherwise, it holds an element and the next node of the list.

```c
struct list
{
    int data;
    struct list *next;
};
```

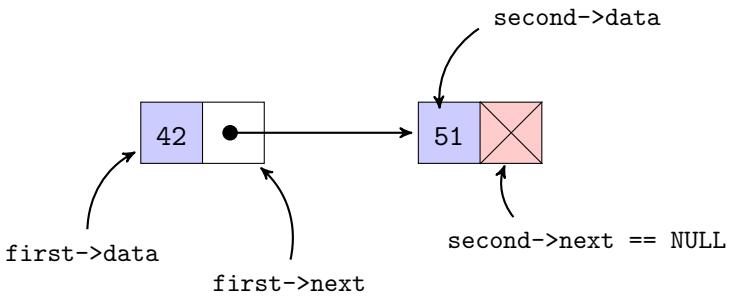
For example, to represent the list `42 -> 51 -> NULL`, we can do:

```c
struct list *first = NULL;
struct list *second = NULL;

second = malloc(sizeof(struct list));
second->data = 51;
second->next = NULL; // The last element is followed by an empty list

first = malloc(sizeof(struct list));
first->data = 42;
first->next = second; // The first element is followed by the second

// Do not forget to free the memory afterwards
```

As you can see, we use the notation `->` to access the content of our variables as they are **pointers** to structures. This is syntactic sugar for:

```c
(*second).data = 51;
second->data = 51; // Easier to use this notation
```

The list represented below:



Be careful!

When manipulating a list do not lose your pointer to the head.

This concept is a bit hard to understand, but with experience, you will learn how to correctly handle it. Time to practice!

3.3 Exercise

3.3.1 Integer linked list

3.4 Goal

A header (`list.h`) containing all the required functions is provided on the intranet.

In this exercise, you will have to implement a linked list, along with its manipulation operations.

- An empty list is represented by a `NULL` pointer.

3.5 list_prepend

```
struct list *list_prepend(struct list *list, int value);
```

This function must insert a node containing `value` at the beginning of the list. The function must return the new list, or `NULL` if an error occured.

3.6 list_length

```
size_t list_length(struct list *list);
```

This function returns the length of the list.

3.7 list_print

```
void list_print(struct list *list);
```

This function displays the elements of the list, separated by a space except the first one, ended by a newline.

If the list is empty, nothing is displayed.

3.8 list_destroy

```
void list_destroy(struct list *list);
```

This function releases all the memory used by `list`.

4 Address Sanitizer

4.1 What is it?

AddressSanitizer, or *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, your compiler 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.

4.2 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++*)

4.3 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.

4.4 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
```

4.5 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 is why the error is reported.

Be careful!

Your code will be systematically corrected with the address sanitizer.

If you do not use it for your projects, you will lose points and make easily avoidable mistakes.

5 Valgrind

5.1 What is it?

Valgrind is a suite of tools for debugging and profiling Linux executables, and an instrumentation framework for building dynamic analysis tools.

One *Valgrind* tool we are particularly interested in is Memcheck: a memory error detector.

Memcheck detects memory-management problems. When a program is run under Memcheck's supervision, all accesses (read or write) to memory are checked, and calls to `malloc`/`new`/`free`/`delete` are intercepted. As a result, Memcheck can detect if your program:

- is using uninitialized variables or pointers
- accesses memory it should not (areas not yet allocated, areas that have been freed, areas past the end of heap blocks, inaccessible areas of the stack)
- does bad frees of heap memory blocks (double frees, mismatched frees)
- leaks memory (areas that should be freed but are not)
- passes overlapping source and destination memory blocks to some standard library functions (like `memcpy(3)`, `strcpy(3)`...). Why? Because the POSIX standards says: "If copying takes place between objects that overlap the behavior is undefined". In other words, when you do a copy using a function related to `memcpy`, memory block of the source and memory block of the destination have to be entirely different (no overlapping allowed), otherwise, the behavior is undefined.

5.2 When to use it?

It depends on the situation, but you should use *Valgrind*:

- regularly, when you make big changes to you program, to ensure there is no memory-management problem (which is often the case)
- when a bug occurs, to get an instant overview of the behavior of your program
- when a bug is suspected and your program is behaving oddly
- in automated tests, to make sure modifications of your code are not raising new memory-management problems

5.3 Valgrind and Memory leaks.

As a reminder, a memory leak is when memory was allocated but is not freed or cannot be freed. When memory leaks happen, valgrind will class them into several categories:

- *Definitely lost*: That means your program is leaking memory – fix those leaks!
- *Indirectly lost*: That means your program is leaking memory in a pointer-based structure. E.g. if the root node of a binary tree is "definitely lost", all the children will be "indirectly lost". If you fix the "definitely lost" leaks, the "indirectly lost" leaks should go away.

- *Possibly lost*: That means your program is leaking memory, unless you are doing unusual things with pointers that could cause them to point into the middle of an allocated block; see the user manual for some possible causes. Use `--show-possibly-lost=no` if you do not want to see these reports.
- *Still reachable*: That means your program is probably ok – it did not free some memory it could have. This is quite common and often reasonable. Do not use `--show-reachable=yes` if you do not want to see these reports.
- *Suppressed*: That means that a leak error has been suppressed. There are some suppressions in the default suppression files. You can ignore suppressed errors.

5.4 How does it work?

To run *Valgrind* on an executable, simply:

```
42sh$ valgrind <program>
```

To specify a tool in particular, just use `--tool` option. Check out the Valgrind *man* for more options. For example, to have details about each leak and show still reachable areas, we will run:

```
42sh$ valgrind --tool=memcheck --leak-check=yes --show-reachable=yes <program>
```

Note that *Memcheck* is the default *Valgrind* tool, so explicitly specifying `--tool=memcheck` is optional.

Because your program runs under *Valgrind* with Memcheck checks, its execution will be slow, **very** slow (about 10 to 30 times slower than normal)! So be careful when using it in extreme conditions.

5.5 Usage example

The diagnostic of *Valgrind* is sometimes difficult to read for beginners. Here is an explained example to help you solve the upcoming exercises:

```c
#include <stdlib.h>

static int *buggy_alloc(void)
{
    int *ptr = malloc(sizeof (char));
    *ptr = 42;
    return ptr;
}

static int test_int_ptr()
{
    int *ptr = buggy_alloc();
    return *ptr == 42;
}

int main(void)
{
    return test_int_ptr();
}
```

```
42sh$ gcc -Wall -Wextra -Werror -pedantic -std=c99 -g example.c -o example
42sh$ valgrind --leak-check=full ./example
==25678== *Memcheck*, a memory error detector
==25678== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==25678== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==25678== Command: ./example
==25678==
==25678== Invalid write of size 4
==25678==    at 0x400550: buggy_alloc (example.c:6)
==25678==    by 0x400568: test_int_ptr (example.c:12)
==25678==    by 0x40058B: main (example.c:18)
==25678==  Address 0x51fc040 is 0 bytes inside a block of size 1 alloc'd
==25678==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==25678==    by 0x400547: buggy_alloc (example.c:5)
==25678==    by 0x400568: test_int_ptr (example.c:12)
==25678==    by 0x40058B: main (example.c:18)
==25678==
==25678== Invalid read of size 4
==25678==    at 0x400571: test_int_ptr (example.c:13)
==25678==    by 0x40058B: main (example.c:18)
==25678==  Address 0x51fc040 is 0 bytes inside a block of size 1 alloc'd
==25678==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==25678==    by 0x400547: buggy_alloc (example.c:5)
==25678==    by 0x400568: test_int_ptr (example.c:12)
==25678==    by 0x40058B: main (example.c:18)
==25678==
==25678==
==25678== HEAP SUMMARY:
==25678==     in use at exit: 1 bytes in 1 blocks
==25678==   total heap usage: 1 allocs, 0 frees, 1 bytes allocated
==25678==
==25678== 1 bytes in 1 blocks are definitely lost in loss record 1 of 1
==25678==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==25678==    by 0x400547: buggy_alloc (example.c:5)
==25678==    by 0x400568: test_int_ptr (example.c:12)
==25678==    by 0x40058B: main (example.c:18)
==25678==
==25678== LEAK SUMMARY:
==25678==    definitely lost: 1 bytes in 1 blocks
==25678==    indirectly lost: 0 bytes in 0 blocks
==25678==      possibly lost: 0 bytes in 0 blocks
==25678==    still reachable: 0 bytes in 0 blocks
==25678==         suppressed: 0 bytes in 0 blocks
==25678==
==25678== For counts of detected and suppressed errors, rerun with: -v
==25678== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

Let us proceed step by step. Every line starts with the identifier of the process, here ==25678==.

```
==25678== *Memcheck*, a memory error detector
==25678== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==25678== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==25678== Command: ./example
```

Valgrind starts by printing the used tool (*Memcheck* here), some information about the **Valgrind** ver-

sion, and the command it used to run the program: `./example` here.

```
==25678== Invalid write of size 4
==25678==    at 0x400550: buggy_alloc (example.c:6)
==25678==    by 0x400568: test_int_ptr (example.c:12)
==25678==    by 0x40058B: main (example.c:18)
==25678==  Address 0x51fc040 is 0 bytes inside a block of size 1 alloc'd
==25678==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==25678==    by 0x400547: buggy_alloc (example.c:5)
==25678==    by 0x400568: test_int_ptr (example.c:12)
==25678==    by 0x40058B: main (example.c:18)
```

This part is a first error report. The first line indicates that our program wrote 4 bytes of data at an invalid memory location. The second line shows that the write was made by an instruction at line 6 in the `example.c` file, in the `buggy_alloc` function. The report is here giving us a *backtrace*, like we have seen with GDB.

The fifth line tells us that the address of the write is corresponding to the start (*0 bytes inside*) of an allocated block of size 1. The last lines give us another backtrace showing us where this memory block has been allocated (in the `test_int_ptr` function with a call to `malloc`).

```
==25678== Invalid read of size 4
==25678==    at 0x400571: test_int_ptr (example.c:13)
==25678==    by 0x40058B: main (example.c:18)
==25678==  Address 0x51fc040 is 0 bytes inside a block of size 1 alloc'd
==25678==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==25678==    by 0x400547: buggy_alloc (example.c:5)
==25678==    by 0x400568: test_int_ptr (example.c:12)
==25678==    by 0x40058B: main (example.c:18)
```

This other report tells us that our program read 4 bytes of data, but on an allocated block of 1 byte only. We can notice that it is the same memory block as before.

```
==25678== HEAP SUMMARY:
==25678==     in use at exit: 1 bytes in 1 blocks
==25678==   total heap usage: 1 allocs, 0 frees, 1 bytes allocated
==25678==
==25678== 1 bytes in 1 blocks are definitely lost in loss record 1 of 1
==25678==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==25678==    by 0x400547: buggy_alloc (example.c:5)
==25678==    by 0x400568: test_int_ptr (example.c:12)
==25678==    by 0x40058B: main (example.c:18)
```

Valgrind then gives us a summary of the heap usage. We learn, (thanks to the `--leak-check=full` option) that a memory leak occurred: 1 byte corresponding to a memory block was “definitely lost”. It means that the last pointer that referenced this block was destroyed, meaning that we lost the block’s address and that we cannot free it. As usual, the *backtrace* allows us to see how this block was allocated. We notice that it is, again, the same block as before.

```
==25678== LEAK SUMMARY:
==25678==    definitely lost: 1 bytes in 1 blocks
==25678==    indirectly lost: 0 bytes in 0 blocks
==25678==      possibly lost: 0 bytes in 0 blocks
==25678==    still reachable: 0 bytes in 0 blocks
```

(continues on next page)

(continued from previous page)

```
==25678==         suppressed: 0 bytes in 0 blocks
==25678==
==25678== For counts of detected and suppressed errors, rerun with: -v
==25678== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 0 from 0)
```

Finally, *Valgrind* gives us a summary of the errors it encountered during execution.

The problem is at line 5 in our code. We ask via `malloc` for a memory zone of one char size (1 byte), but we use an `int *` to point to it, meaning the rest of our code will see this zone as a 4 bytes memory zone (an int is 4 bytes long on this computer). It does not produce a *segfault* because the memory zone allocated by malloc is actually larger (we will see why later this semester), but by dereferencing our `int *`, we are actually accessing 3 more bytes than we should, making us read or write an invalid value. This is a dangerous and fundamentally invalid use of memory. Valgrind detects it and informs us with its "Invalid read" report.

About the memory leak, it can be fixed like this:

```c
static int test_int_ptr()
{
    int *ptr = buggy_alloc();
    int ret = *ptr == 42;
    free(ptr);
    return ret;
}
```

Tips

If Valgrind output is too auster, you can take a look at colour-valgrind. If you want to install it run `pip install --user colour-valgrind`.

5.6 Other tools

As said at the beginning, *valgrind* is a tool suite, which means it is not only a memory management tools. Here is an overview of other tools provided by valgrind:

- `Memcheck`: memory-management problems as we saw.
- `Helgrind`: a thread debugger (deadlocks, data races ...).
- `DRD`: another thread debugger.
- `Massif`: a heap profiler.
- `CacheGrind`: a cache profiler. It will performs detailed simulation of your cache CPU.
- `Callgrind`: A CacheGrind++. It does cache profiling, provide a call tree of your program and many useful insights about your program execution.

These are the main tools of valgrind. You can find more on the valgrind website about them and third party tools.

5.7 Exercise

We give you five Valgrind usage examples. For each example, compile it with debugging symbols, and run it under Valgrind.

```
42sh$ for id in 1 2 3 4 5; do # Compile all exercises.
    make CC=gcc CFLAGS=-g valgrind_${id};
done
```

Without looking at the code, find the memory errors, get their location, and then fix them.

6 Int linked list advanced

6.1 Goal

A header (`list.h`) containing all the required functions is provided.

In this exercise, you will have to implement a linked list, along with its manipulation operations.

- An empty list is represented by a `NULL` pointer.

6.2 list_prepend

- **Authorized functions:** `malloc(3)`

```
struct list *list_prepend(struct list *list, int value);
```

This function must insert a node containing `value` at the beginning of the list. The function must return the new list, or `NULL` if an error occurred.

6.3 list_length

- **Authorized functions:** none.

```
size_t list_length(struct list *list);
```

This function returns the length of the list. If the list is empty, the function must return `0`.

6.4 list_print

- **Authorized functions:** `printf(3)`, `puts(3)`, `putchar(3)`

```
void list_print(struct list *list);
```

This function displays the elements of the list, separated by a space, ended by a newline.

If the list is empty, nothing is displayed.

6.5 Example

For a program called `list_print_example`, calling `list_print` on a list containing `2 -> 1 -> 3`:

```
42sh$ ./list_print_example | cat -e
2 1 3$
```

6.6 list_destroy

- **Authorized functions:** `free(3)`

```
void list_destroy(struct list *list);
```

This function releases all the memory used by `list`.

6.7 list_append

- **Authorized functions:** `malloc(3)`

```
struct list *list_append(struct list *list, int value);
```

This function must insert a node containing `value` at the end of the list `list`. The function must return the new list, or `NULL` if an error occurred.

6.8 list_insert

- **Authorized functions:** `malloc(3)`

```
struct list *list_insert(struct list *list, int value, size_t index);
```

This function inserts a node containing `value` at the position `index` (zero-based) in the list `list`. The function must return the new list.

If `index` is greater than the list size, the behavior is the same than `list_append`.

The function must return `NULL` if an error occurred.

6.9 list_remove

- **Authorized function:** `free(3)`

```
struct list *list_remove(struct list *list, size_t index);
```

This function deletes the node at the position `index`. If `index` is invalid, the list is unchanged.

The function must return `NULL` if an error occurred.

6.10 list_find

- **Authorized function:** none

```
int list_find(struct list *list, int value);
```

This function returns the position (zero-based) of the first node containing `value`. If none is found, the function returns -1.

For the simplicity of the function, the return value is an `int`, even though the position should be a `size_t`. This means that calling `list_find` on a list bigger than the maximum size of an `int` value is not going to be tested.

6.11 list_concat

- **Authorized function:** none

```
struct list *list_concat(struct list *list, struct list *list2);
```

This function concatenates the list `list2` to the list `list` and returns the new list. If `list` is `NULL`, then the new list is `list2`.

6.12 list_sort

- **Authorized functions:** `malloc(3)`, `realloc(3)`, `free(3)`

```
struct list *list_sort(struct list *list);
```

```
struct list *list_sort(struct list *list);
```

This function sorts in ascending order the list `list` and returns the new list.

6.13 list_reverse

- **Authorized functions:** `malloc(3)`, `free(3)`

```
struct list *list_reverse(struct list *list);
```

This function inverts the order of the elements of `list` and returns the new list.

6.14 list_split

- **Authorized functions:** `malloc(3)`, `free(3)`

```
struct list *list_split(struct list *list, size_t index);
```

This function splits `list` in two at the position `index`. `list` keeps the first part, and the function returns the second part. The element at the `index` position belongs to the first part. If `list` is `NULL` or if `index` is invalid, the function returns `NULL`.

It is my job to make sure you do yours.