# EPITECH 2022 - Technical Documentation Documentation

*Release 1.3.38*

**Maxime CORBIN, Jules CASTÉRAN, Jessy SOBREIRO**

**Nov 13, 2018**

# Contents

**1 Table of contents** **3**
- 1.1 C Basics . . . 3
- 1.2 Makefiles . . . 7
- 1.3 Criterion . . . 9
- 1.4 Criterion - Upcoming assert API . . . 14
- 1.5 CSFML : Graphical Programming . . . 22
- 1.6 Debug : Understanding Valgrind's messages . . . 26
- 1.7 Windows Activation . . . 30
- 1.8 WikiHow . . . 31

**2 Introduction** **33**
- 2.1 Why this documentation ? . . . 33

**Caution:** We are currently writing this documentation. It may be incomplete, imprecise, use it at your own risk.

CHAPTER 1

# Table of contents

## 1.1 C Basics

Just to be sure that you still know the basics.

### 1.1.1 Types

#### Integers

| Type | Minimum | Maximum | Bytes |
|---|---|---|---|
| INT | -2,147,483,648 | 2,147,483,647 | 4 |
| UNSIGNED INT | 0 | 4,294,967,295 | 4 |
| LONG | -9,223,372,036,854,775,808 | 9,223,372,036,854,775,807 | 8 |
| UNSIGNED LONG | 0 | 18,446,744,073,709,551,615 | 8 |
| SHORT | -32,768 | 32,767 | 2 |
| UNSIGNED SHORT | 0 | 65,535 | 2 |

### 1.1.2 Operators

#### Arithmetic Operators

#### Addition : +

Addition is used to add operands

```
int a = 10;
int b = 15;
```

Now a + b = 25

### Bitwise Operators

Bitwise operator perfom bit-by-bit operation.

**Did you just say bit-by-bit operation ?**

A bit-by-bit operation is an operation that apply to each bit of a binary value.

**Binary ?**

Let's take a `short int` as an example which value is 42.

In a computer memory, it takes 2 bytes to store our value. It will look like that

```
0 0 0 0 0 0 0 0  0 0 0 1 0 1 0 1
```

It is important to understand what a logic gate is.

Logic are used in electronic, they have inputs and an output

Look at the first gate:

### AND GATE

| Inputs | | Output |
|---|---|---|
| A | B | A & B |
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

The output is 1 when both of the inputs are 1

### OR GATE

| Inputs | | Output |
|---|---|---|
| A | B | A \| B |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

The output is 1 when at least one of the inputs is 1

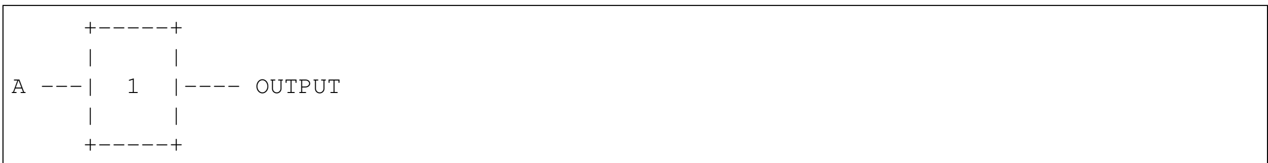
### XOR GATE

```
     +------+
A ---|      |
     |   =1 |---- OUTPUT
B ---|      |
     +------+
```

| Inputs | | Output |
|---|---|---|
| A | B | A ^ B |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

The output is 1 when *only* one of the inputs is 1

### NOT GATE



```
     +------+
     |      |
A ---|  1   |---- OUTPUT
     |      |
     +------+
```

| Inputs | Output |
|---|---|
| A | ~A |
| 0 | 1 |
| 1 | 0 |

The output is the inverse of the inputs

### Operators

Now we are going to look at bitwise operators in C

What operators do is applying logic gate to each one of bits in the variable

### AND : &

AND operator apply AND gate to each bit

```c
short int a = 10; // 00000000 00001010
short int b = 20; // 00000000 00010100
```

To represent the operation:

```
  00000000 00001010
& 00000000 00010100

  00000000 00000000
```

a & b = 0

### OR : |

OR operator apply OR gate to each bit

```
short int a = 10; // 00000000 00001010
short int b = 15; // 00000000 00001111
```

To represent the operation:

```
  00000000 00001010
| 00000000 00001111

  00000000 00001111
```

a | b = 15

### XOR : ^

XOR operator apply XOR gate to each bit

```
short int a = 10; // 00000000 00001010
short int b = 15; // 00000000 00001111
```

To represent the operation:

```
  00000000 00001010
^ 00000000 00001111

  00000000 00000101
```

a ^ b = 5

### NOT : ~

NOT operator applies NOT gate to each bit

```
short int a = 10; // 00000000 00001010
```

To represent the operation:

```
~ 00000000 00001010

  11111111 11110101
```

~a = -11

## 1.2 Makefiles

### 1.2.1 Introduction

A Makefile is a file, read by the `Make` program, which executes all the commands and rules in the Makefile.

---

**Remember**

Remember the first two days of C Pool. You were supposed to use bash and build your own aliases and scripts. `Make` is kind of a custom build script that makes your programmer-life really easier.

---

At EPITECH, you'll be using Makefiles to compile your code, using a `compiler`.

### 1.2.2 Generic Makefile

A simple Makefile is composed of your project source files (the .c files) and some `rules` to make your `Make` command work.

You have to list your source files like this:

```
SRC = ./main.c \
      ./file.c
```

After that, you can use them to build your objects. It will take all `.c` files in `$(SRC)` and compile them into `.o` files.

```
OBJ = $(SRC:.c=.o)
```

For the compilation there is a feature that allow you to compile each `.c` with flags, it's the `+=`. For example let's add verification of errors flags : `-Werror -Wextra` and a flags to find `.h` of your project : `-I./include`. You can call this variable `CFLAGS` for compilation's flags.

```
CFLAGS += -Werror -Wextra -I./include
```

---

**Be careful !**

You don't have to call this variable in your Makefile, he will solo add it to the compilation of your `.c`.

---

Now, set the name of your final binary using `NAME`, so the AutoGrader can find your binary correctly.

```
NAME = binary_name
```

Then, it is mandatory to create a $(NAME) rule that will execute other rules, and render a binary.

```
$(NAME): $(OBJ)
         gcc -o $(NAME) $(OBJ)

all:     $(NAME)
```

---

**Pro-tip**

When you have a rule like `$(NAME)`, the rules put in the same line will be used as mandatory rules. After those rules have been called, the command on the next lines will be called.

For instance, this will execute the `ls` command without executing any previous rule.

```
list:
      ls
```

---

You can also have some rules that permit you to clean-up your folder without having to do it manually.

For instance, this `clean` will remove `.o` files. Also, `fclean` will remove `.o` files and the binary. `re` will do `fclean` and re-make your binary.

```
clean:
        rm -f $(OBJ)

fclean: clean
        rm -f $(NAME)

re:     fclean all
```

Don't forget to put a `.PHONY`, in order to avoid relinking. Put all the rules you use.

```
.PHONY: all clean fclean re
```

And that's pretty much it ! Your Makefile is now ready to use.

### 1.2.3 Criterion Makefile

At EPITECH, you use `criterion` for unit tests. In order to make it clean there is a approach given by EPITECH.

First of all, you have to add a new rule to your main Makefile, according to EPITECH this rule should be named `tests_run`.

---

**Pro tip**

In order to make it cleaner we recommend you to another Makefile in the tests directory and link to the main. To call a Makefile rule of your tests Makefile just type : make -C tests/ [rule_name] in your main Makefile.

---

The `tests_run` rule should compile your sources files `.c` and your tests files. This rule must launch your binary `./unit-tests`.

---

**Mendatory !**

You never have to put your main function in the source files that you compile for unit tests : Criterion have his own.

---

Your tests must compile with the CFLAG `--coverage` (see Generic Makefile). This flag will create `.gcda` and `.gcno` of your sources files.

---

**Tip**

Make a rule to clean all your `.gcda` and `.gcno` files.

---

Now, when you launch your tests_run rule, your binary of tests should compile again and execute so that you can see if you passed tough your tests. You should see your files from –coverage. You can use the gcov [files] to see how many line were executed when you launch your unit tests.

Clear all `.gcda`, `.gcno` and `.c.gcov` and you can push it to the AutoGrader !

### 1.2.4 Library Makefile

### 1.2.5 Advanced Makefile

## 1.3 Criterion

Criterion is a unit-testing tool that will allow you to test your code efficiently.

### 1.3.1 Introduction

#### Setup

If you haven't installed Criterion yet, please download install_Criterion.sh from the intranet, then run it. Criterion will be installed.

#### What is Criterion ?

> **Warning:** Do not forget to follow the Coding Style !

Criterion lets you create a set of tests that will be executed on your code. Creating a test follows this pattern :

```
Test(suite_name, test_name, ...)
{
        //tests
}
```

The `suite_name` must be the name of a set of tests, which will help you know exactly what part of your code fails. The `test_name` is here to help you know which precise test failed, and thus will be usefull to help you debug your code.

> **Danger:** If you want your tests to work, you must compile your code with Criterion's library using the `-lcriterion` flag. You should also consider running your code with the `--verbose` flag if you want a full summary of the test's results.

### 1.3.2 Asserts

Asserts are Criterion's way of defining tests to run. You will have to define several assets in order to test every bit of your code. Let's see an example using Criterion's most basic assert, `cr_assert`. This asserts takes a condition as a parameter, and the test passes if the condition is true :

```
Test(basics, first_test) {
        cr_assert(1 + 1 == 2);
}
```

Here, as one plus one will always be equal to two, the test will always pass. Let's see the full list of Criterion's asserts.

**Warning:** All asserts are located in the header `<criterion/criterion.h>`. Don't forget to include this header in order to make your tests functionnal.

**Note:** Most asserts here have the `assert` keyword in their name. You should be aware that using those macros will stop the whole test right away if they fail. If you want to run some code after the test, even if it fail, you should really consider changing `assert` with `expect`. For example, in the following code the printf function will not be executed :

```
Test(suite_name, test_name) {
        cr_assert(0);
        printf("I wanted to run this code :/");
}
```

But in this example it will still run :

```
Test(suite_name, test_name) {
        cr_expect(0);
        printf("I will live :D");
}
```

### Basic asserts

**Note:** In addition to the condition tested by the following tests, it is possible to give a string as parameter which will be printed to stderr if the test fails. This optionnal string can take printf-like arguments. For example, you can do something like:

```
Test(suite_name, test_name) {
        int i = 0;
        int j = 2;
        cr_assert(i * 2 == j, "The result was %d. Expected %d", i * 2, j);
}
```

**cr_assert** (condition)

Passes if `condition` is `true`.

**cr_assert_not** (condition)

Passes if `condition` is `false`.

**cr_assert_null** (condition)
**cr_assert_not_null** (condition)

Passes if `condition` is `NULL`, or is not `NULL`.

### Common asserts

**Note:** Please note that the following asserts only work for non-array comparison. If you want to compare arrays, please refer to *cr_assert_str_eq()* or *cr_assert_arr_eq()*.

**cr_assert_eq**(Actual, Reference)
**cr_assert_neq**(Actual, Reference)

Passes if and only if `Actual` is equal (or not equal, if you are using `neq`) to `Reference`.

**cr_assert_lt**(Actual, Reference)
**cr_assert_leq**(Actual, Reference)

Will pass if `Actual` is less than (or less than or equal if you used `leq`) `Reference`.

**cr_assert_gt**(Actual, Reference)
**cr_assert_geq**(Actual, Reference)

Will pass if `Actual` is greater than (or greater than or equal if you used `geq`) `Reference`.

### String asserts

**Note:** Those functions won't allow you to compare the output of your progam with a given reference string. To do so you must use redirections. Check *cr_assert_stdout_eq_str()* for more info.

**cr_assert_str_eq**(Actual, Reference)
**cr_assert_str_neq**(Actual, Reference)

Just like *cr_assert_eq()*, but will check two strings, character by character.

**cr_assert_empty**(Value)
**cr_assert_not_empty**(Value)

Will pass if the string is empty (or is not empty is you used `not_empty`).

**Hint:** There are also `str_lt`, `str_gt`, etc... macros that will check the lexicographical values of the two sting given, just like your `my_strcmp` would do (if you've done it well :D).

### Array asserts

**cr_assert_arr_eq**(Actual, Expected, Size)
**cr_assert_arr_neq**(Actual, Expected, Size)

Compares each element of `Actual` with each of `Expected`.

**Caution:** While not documented in Criterion's official documentation, `Size` is mandatory, otherwise the test will be marked as failed.

### Redirections

**Tip:** To use the following assertions, you must include `<criterion/redirect.h>` along with `<criterion/criterion.h>`. `redirect.h` allows Criterion to get the content of stdout and stderr and run asserts on it. You also need to create a function that calls the `cr_redirect_stdout()` function.

**`cr_assert_stdout_eq_str`**(Value)
**`cr_assert_stdout_neq_str`**(Value)

Compares the content of `stdout` with `Value`. This assertion behaves similarly to *`cr_assert_str_eq()`*.

**`cr_assert_stderr_eq_str`**(Value)
**`cr_assert_stderr_neq_str`**(Value)

Compares the content of `stderr` (a.k.a. "error output") with `Value`.

Here is a sample usage of this assert.

```
#include <criterion/criterion.h>
#include <criterion/redirect.h>

void redirect_all_stdout(void)
{
        cr_redirect_stdout();
        cr_redirect_stderr();
}

int error(void)
{
        write(2, "error", 5);
        return(0);
}

Test(errors, exit_code, .init=redirect_all_stdout)
{
        error();
        cr_assert_stderr_eq_str("error", "");
}
```

**Note:** Note that you MUST include criterion.h and redirect.h in this order, otherwise you tests won't work.

## 1.3.3 Test options

### Options reference

It is possible for you to provide additional parameters to a test. Here is a full list of thos parameters and what you can do with them.

**`.init`**

This parameter takes a function pointer as an argument. Criterion will execute the function just before running the test.

Note that the function pointer should be of type `void (*)(void)`.

Here is a sample usage of this parameter.

```
void my_func(void)
{
        my_putstr("Here is the beginning of my test\n");
}

Test(suite_name, test_name, .init = my_func)
{
        //tests
}
```

**.fini**

This parameter takes a function pointer to a function that will be executed after the tests is finished.

It takes the same pointer type as the `.init` parameter, and also has the same usage.

**.signal**

> **Warning:** In order to use this parameter, you must include the header <signal.h>

If a test receives a signal, it will by default be marked as a failure. However, you can expect a test to pass if a special kind of signal is received.

```
#include <stddef.h>
#include <signal.h>
#include <criterion/criterion.h>

Test(example, will_fail)
{
        int *ptr = NULL;
        *ptr = 42;
}

Test(example, will_pass, .signal = SIGSEGV)
{
        int *ptr = NULL;
        *ptr = 42;
}
```

In the above example, the first test will fail while the second one will not.

You can find a full list of handled signals by checking signal.h's documentation here : http://pubs.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html.

**.exit_code**

By default, Criterion will mark a test as failed if it exits with another exit code than 0.

If you want to test your error handling, you can use the `.exit_code` parameter so the test will be marked as passed if the given exit code is found.

Here is a sample usage of this parameter :

```
#include <unistd.h>

int error(void)
{
        write(2, "error", 5);
```

(continues on next page)

(continued from previous page)

```
        exit(0);
}

Test(errors, exit_code, .exit_code = 84)
{
        error();
        cr_assert_stderr_eq("error", "");
}
```

**.disabled**

If `true`, the test will be skipped.

**.description**

This parameter must be used in order to give extra definition of the test's purpose, which can be quite helpful if your suite_name and test_name aren't as explicit as you would like.

**.timeout**

This parameter takes a `double`, representing a duration. If your test takes longer than this duration to run, the test will be marked as failed.

#### Suite configuration

If you want to set a test for all of a suite's members (for example, setting the exit code of all your error handling tests), you can, using the `TestSuite` macro.

```
#include <criterion/criterion.h>

TestSuite(suite_name, [params...]);

Test(suite_name, test_1)
{
        //tests
}

Test(suite_name, test_2)
{
        //other tests
}
```

As you can see, you can set some params to all the tests with the same suite_name at once.

## 1.4 Criterion - Upcoming assert API

### 1.4.1 Introduction

In this tutorial we will take a look at Criterion's upcoming assert API. If you're not familiar with Criterion, you may start with the first part of our Criterion documentation.

### Setup

Before using the new assert API, it is necessary to download and build the development version of Criterion, as we are working with unreleased functionality.

Start by downloading the code with this command :

```
$ git clone --recursive git@github.com/Snaipe/Criterion.git
```

**Note:** Take note of the *–recursive* option, it is necessary to compile the code.

Once the code is downloaded, let's build it.

```
$ mkdir build
$ cd build
$ cmake ..
$ cmake --build .
$ sudo make install
```

Now, to use any of the features described in this page, you will need to include the following headers to your test files.

```
#include <criterion/criterion.h>
#include <criterion/new/assert.h>
```

## 1.4.2 Assertions

With the old Criterion, there was a multitude of assertion functions, covering a lot of cases. This led to a few useless repetitions, such as *cr_assert_eq*, *cr_assert_str_eq*, *cr_assert_arr_eq*… The new API is much more straightforward, there are only 5 assertion functions :

### Reference

**cr_assert** (Criterion)

Passes if `Criterion` is true, abort if not.

**cr_expect** (Criterion)

Passes if `Criterion` is true, fails if not.

**Note:** The difference between *cr_assert* and *cr_expect* is the behavior when the test fails. An assert will stop the test right away while an expect will continue until the end of the test.

**cr_fail** ()

Marks the test as failed. Can be used when testing manually with conditions for example.

**cr_fatal** ()

Marks the test as failed then abort.

**cr_skip** ()

Marks the test as skipped then abort.

**Note:** All of those macros can take an optional printf-like string which will be printed if the test fails (see below for an example).

### Example

```
Test(my_suite, my_test) {
    FILE *myfile = fopen("myfile.txt", "r");
    if (myfile == NULL) {
        cr_fatal("Test mytest failed, file could not be opened : %s\n",␣
↪strerror(errno));
    } else {
        // Do something...
    }
}
```

## 1.4.3 Criteria

As you can see, there aren't any macros that compare values. This new API does not expect you to write the comparison functions yourself, you may use a set of predefined Criteria (or _Criterions_) which are the recommended parameters of *cr_assert* and *cr_expect*.

### Reference

### Logical Criteria

Logical criteria are simple helpers to test multiple criteria at once.

**`not`** (Criterion)

Evaluates to *!Criterion*.

**`all`** (...)

Takes a sequence of Criteria as parameters. Will be true if all Criteria given are true. (equivalent to an &&)

**`any`** (...)

Takes a sequence of Criteria as parameters. Will be true if any Criteria given is true. (equivalent to an ||)

**`none`** (...)

A combination of a *not* over each criteria and an *all*.

### Tagged Criteria

Those are the real useful testing macros.

**Note:** Do not worry with the following *Tag* parameters, the complete list of tags are described in the nex section.

### General Purpose

**eq** (Tag, Actual, Expected)

Tests if Actual is equal to Expected.

---

**Note:** This function may only use the operator == if the tag specifies numeric values. It is also able to the the equality between strings if given the tag *str*.

---

**ne** (Tag, Actual, Expected)

Tests if Actual is not equal to Expected.

**lt** (Tag, Actual, Expected)

Tests if Actual is less than Expected.

**le** (Tag, Actual, Expected)

Tests if Actual is less than or equal to Expected.

**gt** (Tag, Actual, Expected)

Tests if Actual is greater than Expected.

**ge** (Tag, Actual, Expected)

Tests if Actual is greater than or equal to Expected.

### Floating point

**Warning:** The following criteria only work with the following tags : *flt*, *dbl* and *ldbl*.

### Epsilon

**Warning:** This method of comparison is more accurate when comparing two IEEE 754 floating point values that are near zero. When comparing against values that aren't near zero, please use ieee_ulp_eq instead.

It is recommended to have Epsilon be equal to a small multiple of the type epsilon (FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON) and the input parameters.

**epsilon_eq** (Tag, Actual, Expected, Epsilon)

Tests if Actual is almost equal to Expected, the difference being the Epsilon value.

**epsilon_ne** (Tag, Actual, Expected, Epsilon)

Tests if Actual is different to Expected, the difference being more than the Epsilon value.

#### ULP

**Warning:** This method of comparison is more accurate when comparing two IEEE 754 floating point values when Expected is non-zero. When comparing against zero, please use epsilon_ne instead.

A good general-purpose value for Ulp is 4.

**epsilon_eq** (Tag, Actual, Expected, Epsilon)

Tests if Actual is almost equal to Expected, by being between Ulp units from each other.

**epsilon_ne** (Tag, Actual, Expected, Epsilon)

Tests if Actual is different to Expected, the difference being more than Ulp units from each other.

#### Example

All the following tests pass :

```
Test(strings, eq) {
    cr_assert(eq(str, "Hello", "Hello"));
}
Test(strings, ne) {
    cr_assert(ne(str, "Hello", "Hella"));
}
Test(integers, eq) {
    cr_assert(eq(int, 8, 8));
}
Test(logical, all) {
    cr_assert(all(eq(int, 8, 8), eq(str, "Hello", "Hello")));
}
```

### 1.4.4 Tags

The tags are Criterion-defined macros that represent standard C types.

#### Predefined Tags

Here is the complete list of all predefined tags :

int8_t **i8**

int16_t **i16**

int32_t **i32**

int64_t **i64**

uint8_t **u8**

uint16_t **u16**

uint32_t **u32**

uint64_t **u64**

size_t **sz**

void * **ptr**

intptr_t **iptr**

uintptr_t **uptr**

char **chr**

int **int**

unsigned int **uint**

long **long**

unsigned long **ulong**

unsigned long long **ullong**

float **flt**

double **dbl**

long double **ldbl**

complex double **cx_dbl**

complex long double **cx_ldbl**

See below for details about the implementation of this structure.

const char * **str**

const wchar_t * **wcs**

String of wide characters

const TCHAR * **tcs**

Windows character string.

### Mem struct

Here is the definition of the cr_mem structure :

struct **cr_mem**

const void * **data**

data is a pointer to the data to test

size_t **size**

size is the size of data to test.

### User-Defined Type

You can use the following macro to use you own types as Tags :

**type** (UserType)

You may then use you type the same way as any other tag :

```
cr_assert(eq(type(my_type), var1, var2));
```

However there are some functions to implement in order to use this type in your code.

> **Warning:** Due to implementation restrictions, UserType must either be a structure, an union, an enum, or a typedef.
>
> For instance, these are fine:
>
> ```
> type(foo)
> type(struct foo)
> ```
>
> and these are not:
>
> ```
> type(foo *)
> type(int (&foo)(void))
> ```
>
> in these cases, use a typedef to alias those types to a single-word token.

#### In any case

The type must be printable, and thus should implement a “to-string” operation. These functions are mandatory in any case.

##### C

```
char *cr_user_<type>_tostr(const <type> *val);
```

For example if you have a *character* type, you must implement the *char *cr_mem_character_tostr(const character *val);*

##### C++

```
std::ostream &operator<<(std::ostream &os, const <type> &val);
```

#### eq, ne, le, ge

These functions are mandatory in order to use the operators *eq*, *ne*, *le* or *ge* with your type. They should return 1 (or true) if the two parameters are equal, 0 (or false) otherwise.

##### C

```
int cr_user_<type>_eq(const <type> *lhs, const <type> *rhs);
```

##### C++

```
bool operator==(const <type> &lhs, const <type> &rhs);
```

#### lt, gt, le, ge

These functions are mandatory in order to use the operators *lt*, *gt*, *le* or *ge* with your type. They should return 1 (or true) if the first parameter is less than the second, 0 (or false) otherwise.

##### C

```
int cr_user_<type>_lt(const <type> *lhs, const <type> *rhs);
```

##### C++

```
bool operator<(const <type> &lhs, const <type> &rhs);
```

### 1.4.5 User-defined type example

```
#include <stdio.h>
#include <criterion/criterion.h>
#include <criterion/new/assert.h>


typedef struct vector3d {
    int x;
    int y;
    int z;
} vector3d;

/*
    Defines the string representation of a vector3d
*/
char *cr_user_vector3d_tostr(const vector3d *val)
{
    char *str = malloc(sizeof(char) * (12 + 3 * 9));

    if (str == NULL) {
        return "";
    }
    sprintf(str, "X:%d; Y:%d: Z:%d\n", val->x, val->y, val->z);
    return str;
}

/*
    Defines an equality between two vector3d
*/
int cr_user_vector3d_eq(const vector3d *lhs, const vector3d *rhs)
{
    if (lhs->x == rhs->x && lhs->y == rhs->y && lhs->z == rhs->z) {
        return 1;
    }
    return 0;
}

/*
    Creates a new vector with predefined values
```

(continues on next page)

(continued from previous page)

```
*/
vector3d *create_vector(void)
{
    vector3d *vect = malloc(sizeof(vector3d));

    vect->x = 8;
    vect->y = 7;
    vect->z = 2;
    return vect;
}

Test(usertype, test) {
    vector3d *test_vect = malloc(sizeof(vector3d));
    vector3d *vect = create_vector();

    test_vect->x = 8;
    test_vect->y = 7;
    test_vect->z = 2;

    cr_assert(eq(type(vector3d), *vect, *test_vect));
}
```

## 1.5 CSFML : Graphical Programming

### 1.5.1 Table of contents

A soon-to-be complete and human-readable CSFML API reference.

### 1.5.2 Reference

#### Window

#### Window managing

##### sfRenderWindow

This is the most basic element of any graphic program : a window.

*sfRenderWindow* ***sfRenderWindow_create**(sfVideoMode *mode*, const char **title*, sfUint32 *style*, const sfContextSettings **settings*)

This function will setup a new window, based on the parameters given.

**Parameters**

- **mode** `sfVideoMode` - The video mode to use (width, height and bits per pixel of the window).
- **title** `const char *` - The title of the window.
- **style** `sfUint32` - You must specify at least one of the following :
  - `sfNone` : None of the other style options will be used.

- `sfResize` : Will add a "resize" button that will allow you to change the size of the window.
- `sfClose` : Will add a "close" button that will allow you to close your window.
- `sfFullscreen` : Will make your window full screen.

If you want to add two or more parameters, simply separate them using pipes (see example below).

**settings** `sfContextSettings *` - Those are advanced render settings you can add. NULL to use default values.

**Return** `sfRenderWindow *` - New render window.

*sfRenderWindow* ***sfRenderWindow_createUnicode**(sfVideoMode *mode*, const sfUint32 **title*, sfUint32 *style*, const sfContextSettings **settings*)

This function takes the same parameters as `sfRenderWindow_create()`, but this will take an array of integers as title, allowing you to have a unicode title.

```
sfVideoMode mode = {800, 600, 32};
window = sfRenderWindow_create(mode, "My awesome window", sfResize | sfClose, NULL)
```

void **sfRenderWindow_destroy**(*sfRenderWindow* **window*)

This function allows you to destroy an existing window.

**Parameter**

**window** `sfRenderWindow *` - The render window to destroy.

void **sfRenderWindow_close**(*sfRenderWindow* **window*)

This function will close a render window (while not destroying it's internal data.)

**Parameter**

**window** `sfrenderWindow *` - The RenderWindow to close.

void **sfRenderWindow_clear**(*sfRenderWindow* **window*, sfColor *color*)

This function will clear all pixels in the window, replacing them with the given color.

**Parameters**

**window** `sfRenderWindow *` - The RenderWindow to clear.

**color** `sfColor` - The color to which all pixel will change.

void **sfRenderWindow_display**(*sfRenderWindow* **window*)

This function will display all sprites on screen.

**Parameter**

**window** `sfRenderWindow *` - The RenderWindow to display.

### Getting window data

There are a lot of data that can be obtained from a `sfRenderWindow` object. As I don't want to spend my whole life writing this paragraph while knowing that no one will ever read this, I'll only list a bunch of useful functions. I'll probably add the others sooner or later.

sfVector2u **sfRenderWindow_getSize** (const *sfRenderWindow* *window*)

This function allows you to know the size of a given render window.

**Return** `sfVector2u` - Contains the size of the window in pixels.

sfBool **sfRenderWindow_hasFocus** (*sfRenderWindow* *window*)

**Return** `sfBool` - Will be `sfTrue` if the window has focus (i.e. it can receive inputs), `sfFalse` otherwise. This function can be useful if you want to pose your program if the window doesn't have focus.

sfBool **sfRenderWindow_isOpen** (*sfRenderWindow* *window*)

Tell whether or not a render window is open.

**Return** `sfBool` - Will be `sfTrue` if the window is open, `sfFalse` otherwise.

### Other useful options

Here are all other functions that I (Oursin) find useful for our first projects.

sfBool **sfRenderWindow_pollEvent** (*sfRenderWindow* *window*, sfEvent *event*)

This function will check the event queue and pop one. As such, if you want your program to take all inputs into account, it is highly advised to empty the event queue at the start of your game loop.

**Parameters**

- **window** `sfRenderWindow *` - The target window.
- **event** `sfEvent *` - This must take a pointer to the sfEvent object which will be filled.

**Return** A `sfBool` will be returned, indicating whether or not an event was returned.

void **sfRenderWindow_setFramerateLimit** (*sfRenderWindow* *window*, unsigned int *limit*)

This function allows you to set your program's framerate.

**Parameters**

- **window** `sfRenderWindow` - Target render window.
- **limit** `unsigned int` - Defines the maximum number of frames your program should display each second.

void **sfRenderWindow_setKeyRepeatEnabled** (*sfRenderWindow* *window*, sfBool *enabled*)

This function enables or disables automatic key-repeat for keydown events. If enabled, a key pressed down will create new `sfEvent` objects all time until it is released.

**Parameters**

- **window** `sfRenderWindow` - Target render window.
- **enabled** `sfBool` - `sfTrue` to enable, `sfFalse` to disable.

void **sfRenderWindow_setMouseCursorVisible** (*sfRenderWindow* *window*, sfBool *show*)

This function allows you to hide the mouse cursor on a render window.

**Parameters**

- **window** `sfRenderWindow` - Target render window.
- **show** `sfBool` - `sfTrue` to show, `sfFalse` to hide.

### Drawing

In CSFML, there are 4 types of objects that can be displayed, 3 of them beign ready to be used : sprites, text and shapes. The other one, vertex arrays, is designed to help you create your own drawable entities, but you would probably not use it for now.

void **sfRenderWindow_drawSprite** (*sfRenderWindow* **window*, const sfSprite **sprite*, sfRenderStates **states*)

**Parameters**

- **window** `sfRenderWindow *` - The window to draw to.
- **sprite** `sfSprite *` - The sprite to draw.
- **states** `sfRenderStates *` - This can be used to use advanced render options, such as shaders, transfomations etc…

void **sfRenderWindow_drawText** (*sfRenderWindow* **window*, sfText **text*, sfRenderStates **states*)

**Parameters**

- **window** `sfRenderWindow *` - The window to draw to.
- **sprite** `sfText *` - The text object to display on screen. Note that this is not a char *, but an sfText object, which must be created first.
- **states** `sfRenderStates *` - This can be used to use advanced render options, such as shaders, transfomations etc…

void **sfRenderWindow_drawShape** (*sfRenderWindow* **window*, sfShape **shape*, sfRenderStates **states*)

**Parameters**

- **window** `sfRenderWindow *` - The window to draw to.
- **sprite** `sfShape *` - The shape object to display on screen.
- **states** `sfRenderStates *` - This can be used to use advanced render options, such as shaders, transfomations etc…

## 1.5.3 Examples

### RenderWindow

Here is a sample example of most functions related to `sfRenderWindow`. For the following example, we will assume that `sprite`, `text` and `shape` were already created, and are variables of type `sfSprite`, `sfText` and `sfShape`, respectively.

We will also assume that `event` is of type `sfEvent`.

```
sfVideoMode mode = {1080, 720, 32};
sfRenderWindow *window;

window = sfRenderWindow_create(mode, "window", sfClose, NULL);
sfRenderWindow_setFramerateLimit(window, 60);
while (sfRenderWindow_isOpen(window) && sfRenderWindow_hasFocus
                                        (window)) {
        while (sfRenderWindow_pollEvent(window, &event)) {
                if (event.type == sfEvtClosed)
                        sfRenderWindow_close(window);
```

(continues on next page)

(continued from previous page)

```
        }
        sfRenderWindow_clear(window, sfBlack);
        sfRenderWindow_drawSprite(window, sprite, NULL);
        sfRenderWindow_drawShape(window, shape, NULL);
        sfRenderWindow_drawText(window, text, NULL);
        sfRenderWindow_display(window);
}
sfRenderWindow_destroy(window);
```

## 1.6 Debug : Understanding Valgrind's messages

Valgrind is a very useful debug tool, which happens to be already installed on EPITECH's dump.

### 1.6.1 Introduction

#### What is Valgrind ?

Valgrind is an "instrumentation framework for building dynamic analysis tools", according to Valgrind's official documentation.

Valgrind comes with a bunch of tools, but in this page we will only focus on one of those tools : Memcheck.

Memcheck is a memory error detector. As such, it will detect and show you every memory error your code produces. It will also show you your program's memory leaks.

#### How to use it ?

To use Valgrind to debug your program, you can simply add *Valgrind* in front of your program's name and arguments. It should look like this

```
$ valgrind [valgrind\'s options] ./program [program\'s arguments]
```

Valgrind will now lauch your program and report any error it detects.

### 1.6.2 Valgrind's messages

**Warning:** Valgrind will give you more information about where your errors come from if your code has been compiled using GCC's `-g` flag.

#### Invalid read/write

One of the most common errors you will encounter are invalid reads or writes.

#### Invalid write

First, let's write a simple C program.

```c
int main(void)
{
        char *str = malloc(sizeof(char) * 10);
        int i = 0;

        while (i < 15) {
                str[i] = '\0';
                i = i + 1;
        }
        free(str);
        return (0);
}
```

Yes, this code is absolutely useless, but still, let's compile it then run it with valgrind.

```
$ gcc main.c -g
$ valgrind ./a.out
==18332== Memcheck, a memory error detector
==18332== Copyright (C) 2002-2017, and GNU GPL\'d, by Julian Seward et al.
==18332== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==18332== Command: ./a.out
==18332==
==18332== Invalid write of size 1
==18332==    at 0x400553: main (test.c:7)
==18332==  Address 0x521004a is 0 bytes after a block of size 10 alloc\'d
==18332==    at 0x4C2EB6B: malloc (vg_replace_malloc.c:299)
==18332==    by 0x400538: main (test.c:3)
==18332==
==18332==
==18332== HEAP SUMMARY:
==18332==     in use at exit: 0 bytes in 0 blocks
==18332==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated
==18332==
==18332== All heap blocks were free'd -- no leaks are possible
==18332==
==18332== For counts of detected and suppressed errors, rerun with: -v
==18332== ERROR SUMMARY: 5 errors from 1 contexts (suppressed: 0 from 0)
```

So, what happened ? Well, Valgrind detected an invalid write error in our program. But what does it mean ?

"Invalid write" means that our program tries to write data in a memory zone where it shouldn't.

But Valgrind tells you way more than that. It first tells you the size of the written data, which is 1 bytes, and corresponds to the size of a character. Then the line `at 0x400553: main (test.c:7)` tells you at which line your error occured. Line 7, which corresponds to `str[i] = '\0'`.

At the line `Address 0x521004a is 0 bytes after a block of size 10 alloc\'d`, it also tells you that the invalid adress is located right after a block of ten bytes allocated. What this means is that a 10 bytes (so probably 10 characters) long memory zone was allocated, but we tried to write an eleventh byte.

### Invalid read

This other code will produce a Invalid read error :

```c
int main(void)
{
        int i;
```

(continues on next page)

(continued from previous page)

```
        int *ptr = NULL;

        i = *ptr;
        return (0);
}
```

If we compile and run this code, Valgrind will produce this error :

```
==26212== Invalid read of size 4
==26212==    at 0x400497: main (test.c:8)
==26212==  Address 0x0 is not stack\'d, malloc\'d or (recently) free\'d
```

It means that we tried to read 4 bytes, starting at adress 0x0 (for those of you who don't know it yet, NULL is actually a pointer to adress 0x0, so we tried to read 4 bytes starting from NULL).

As before, Valgrind also tells us that the error occured at line 8 of our code, which corresponds to this instruction : `i = *ptr`.

### Conditional jumps

Let's create a new C program :

```
int main(void)
{
        int i;

        if (i == 0) {
                my_printf("Hello\n");
        }
        return (0);
}
```

Valgrind will produce this error :

```
==28042== Conditional jump or move depends on uninitialised value(s)
==28042==    at 0x4004E3: main (test.c:5)
```

This message may be a bit harder to understand.

Well, a jump is a computer instruction similar to a `goto` in C. There are several types of jumps. Some are unconditionnal, meaning the jump will always occur. Some other are conditionals, which means that the jump will be taken if a previous test was successful, and will not otherwise.

In this case, our program had a conditional jump, because one of the values that were test was not initialized, it led to unexpected behaviour. It means that the outcome of the test may change. For example it could work as intented on your computer, but could fail during the autograder's tests.

**Note:** This type of error could happen if you do some tests involving a recently malloc'd block. (Note that malloc will *never* initialize your data).

### Syscall param points to unadressable bytes

Here is our program :

```
int main(void)
{
        int fd = open("test", O_RDONLY);
        char *buff = malloc(sizeof(char) * 3);

        free(buff);
        read(fd, buff, 2);
}
```

`read` will try to read at the adress pointed to by `buff`. But this adress has already been free'd, so Valgrind will show us this error :

```
==32002== Syscall param read(buf) points to unaddressable byte(s)
==32002==    at 0x4F3B410: __read_nocancel (in /usr/lib64/libc-2.25.so)
==32002==    by 0x400605: main (test.c:11)
==32002==  Address 0x5210040 is 0 bytes inside a block of size 3 free\'d
==32002==    at 0x4C2FD18: free (vg_replace_malloc.c:530)
==32002==    by 0x4005EF: main (test.c:10)
==32002==  Block was alloc\'d at
==32002==    at 0x4C2EB6B: malloc (vg_replace_malloc.c:299)
==32002==    by 0x4005DF: main (test.c:8)
```

Here there is a lot of information that will help you debug your code. First, we know that we gave an invalid pointer to a system call, `read` in our case.

Then Valgrind tells us that this pointer is "0 bytes inside a block of size 3 free'd". In fact, we allocated a 3 bytes block, then free'd it. "0 bytes inside" means that our pointer points to the very first byte of this block.

Valgrind tells us where the error occured, where the block was free'd and also where is was malloc'd.

### Invalid/mismatched frees

#### Invalid free

Another error you may encounter is the "Invalid free" one. It means that we tried to free a pointer that cannot be free'd. Here is an example :

```
int main(void)
{
        char *buff = malloc(sizeof(char) * 54);

        free(buff);
        free(buff);
        return (0);
}
```

Yes, I agree, this error is obvious. But it does happen that the same pointer is twice free'd, or that some programmer tries to free something that wasn't allocated. There are plenty of reasons for an invalid free to happen. Let's look at Valgrind's message :

```
==755== Invalid free() / delete / delete[] / realloc()
==755==    at 0x4C2FD18: free (vg_replace_malloc.c:530)
==755==    by 0x400554: main (test.c:10)
==755==  Address 0x5210040 is 0 bytes inside a block of size 54 free\'d
==755==    at 0x4C2FD18: free (vg_replace_malloc.c:530)
==755==    by 0x400548: main (test.c:9)
```

(continues on next page)

(continued from previous page)

```
==755==  Block was alloc\'d at
==755==    at 0x4C2EB6B: malloc (vg_replace_malloc.c:299)
==755==    by 0x400538: main (test.c:7)
```

Valgrind tells use that there is a problem with a free, a delete, a delete[] or a realloc, but since delete is a C++ instruction, and we're not allowed to use realloc at EPITECH, you will probably only use free.

As before, Valgrind tells us that the error occured because we tried to use free on an adress that belongs to an already free'd block.

#### Mismatched free

Another error you can encounter is this one :

```
==3073== Mismatched free() / delete / delete []
==3073==    at 0x4C2FD18: free (vg_replace_malloc.c:530)
==3073==    by 0x400613: main (in /home/oursin/a.out)
==3073==  Address 0xa09a5d0 is 0 bytes inside a block of size 368 alloc\'d
==3073==    at 0x4C2F1CA: operator new(unsigned long) (vg_replace_malloc.c:334)
==3073==    by 0x4E5AB0F: sfSprite_create (in /usr/local/lib/libc_graph_prog.so)
==3073==    by 0x400603: main (in /home/oursin/a.out)
```

Here, I created a CSFML sprite using `sfSprite_create`, then I tried to free this sprite, resulting in this error.

In fact, `sfSprite_create` does allocate some memory, but it does not use our dear friend `malloc`, but it's C++ brother, `new`. And the problem is that something that has been allocated using `new` must be free'd using `delete`, not `free`. As `delete` does not exist in C, you should use CSFML's `sfSprite_destroy` function.

#### Fishy values

The last type of error you may see is this one :

```
==29010== Argument 'size' of function malloc has a fishy (possibly negative) value: -1
==29010==    at 0x4C2EB6B: malloc (vg_replace_malloc.c:299)
==29010==    by 0x4004EA: main (in /home/oursin/a.out)
```

It simply means that you gave a impossible value to a system call. In this case I called `malloc` with argument `-1`.

## 1.7 Windows Activation

### 1.7.1 The magic command

For this, you need to be on the IONIS Wireless Network

Just press Windows + X, choose "Windows PowerShell (Admin)" In it, copy paste this command:

```
slmgr /ipk W269N-WFGWX-YVC9B-4J6C9-T83GX
```

And reboot. You're done!

## 1.8 WikiHow

Have you ever wondered how to properly malloc an integer variable ? Here you'll find the answers to all your questions !

> **Warning:** Please do not consider this page as a reliable source of knowledge. If you really believe anything here, you're even dumber than you look.

### 1.8.1 How to malloc an int

First things first : "Why should I use malloc on an int variable ?"

> **To ensure you have enough space to hold every possible numeric value.**

Now that this is clear, let's speak a bit about malloc. Malloc uses a special variable type, called size_t to take its value. Usually this type of variable can be obtained by using the *sizeof* function. But have you ever wondered what happens if you use sizeof on the size_t type ? Yeah, you've guessed it. Infinity.

Now that we know that, here's a sample code to help you create you integer that can hold infinity.

```
int integer = malloc(sizeof(size_t));
```

> **Warning:** Do not EVER try to malloc infinity onto a void variable. DO YOU REALLY WANT TO KNOW WHAT HAPPENS WHEN YOU PUT AN INFINITY INTO NOTHING ? I do not. So please be mercyful and spare our lives.

### 1.8.2 How to set the value of all elements of an array to 42

You must be thinking "Hey, it's useless, i've already malloc'd my array, and by default all elements are set to 42, why should i want this ?".

Well, the answer is simple. You want to reset you array. Bonus point : If it is a char array, it'll look like a hidden password when you'll print it. And, let's face it, that's cool.

Now i'll show you how to set all values to 42.

> Step 1: Creating a counter variable.
>
> You want to edit every single element of your char array, so you'll need the help of a counter variable in order to do this.

```
char *my_reset_array(char *array)
{
        int i = my_strlen(array);
}
```

> Step 2: Iterate though the array
>
> Because you want to edit every element, you'll need a loop structure.

```
char *my_reset_array(char *array)
{
        int i = my_strlen(array);

        while (i >= 0) {
                i = i - 1;
        }
}
```

Step 3: Edit every Value

Now all you need to do is editing all values, then returning the array.

```
char *my_reset_array(char *array)
{
        int i = my_strlen(array);

        while (i >= 0) {
                array[i] = 42;
                i = i - 1;
        }
        return (array);
}
```

# CHAPTER 2

# Introduction

The purpose of this documentation is to help you during your EPITECH course. You will find concise documentation on main topics studied at EPITECH.

## 2.1 Why this documentation ?

Since EPITECH is built on DIY-learning, you may lack of precise documentation, especially if you missed some topics. This RTD is built to give you a support on which you can rely with confidence.

We really encourage people who find errors in the documentation to send us an e-mail:

- Jessy SOBREIRO (jessy.sobreiro@epitech.eu), author, IT systems manager.
- Maxime CORBIN (maxime.corbin@epitech.eu), author, main contributor.
- Jules CASTÉRAN (jules.casteran@epitech.eu), author.
- Corentin RONDIER (corentin.rondier@epitech.eu), for Windows Activation

# Index

## Symbols

.description (C member), 14
.disabled (C member), 14
.exit_code (C member), 13
.fini (C member), 13
.init (C member), 12
.signal (C member), 13
.timeout (C member), 14

## A

all (C function), 16
any (C function), 16

## C

chr (C macro), 19
cr_assert (C function), 10, 15
cr_assert_arr_eq (C function), 11
cr_assert_arr_neq (C function), 11
cr_assert_empty (C function), 11
cr_assert_eq (C function), 11
cr_assert_geq (C function), 11
cr_assert_gt (C function), 11
cr_assert_leq (C function), 11
cr_assert_lt (C function), 11
cr_assert_neq (C function), 11
cr_assert_not (C function), 10
cr_assert_not_empty (C function), 11
cr_assert_not_null (C function), 10
cr_assert_null (C function), 10
cr_assert_stderr_eq_str (C function), 12
cr_assert_stderr_neq_str (C function), 12
cr_assert_stdout_eq_str (C function), 12
cr_assert_stdout_neq_str (C function), 12
cr_assert_str_eq (C function), 11
cr_assert_str_neq (C function), 11
cr_expect (C function), 15
cr_fail (C function), 15
cr_fatal (C function), 15
cr_mem (C type), 19
cr_skip (C function), 15
cx_dbl (C macro), 19
cx_ldbl (C macro), 19

## D

data (C member), 19
dbl (C macro), 19

## E

epsilon_eq (C function), 17, 18
epsilon_ne (C function), 17, 18
eq (C function), 17

## F

flt (C macro), 19

## G

ge (C function), 17
gt (C function), 17

## I

i16 (C macro), 18
i32 (C macro), 18
i64 (C macro), 18
i8 (C macro), 18
int (C macro), 19
iptr (C macro), 19

## L

ldbl (C macro), 19
le (C function), 17
long (C macro), 19
lt (C function), 17

## N

ne (C function), 17
none (C function), 16
not (C function), 16

## P

ptr (C macro), 18

## S

sfRenderWindow (C type), 22
sfRenderWindow_clear (C function), 23
sfRenderWindow_close (C function), 23
sfRenderWindow_create (C function), 22
sfRenderWindow_createUnicode (C function), 23
sfRenderWindow_destroy (C function), 23
sfRenderWindow_display (C function), 23
sfRenderWindow_drawShape (C function), 25
sfRenderWindow_drawSprite (C function), 25
sfRenderWindow_drawText (C function), 25
sfRenderWindow_getSize (C function), 23
sfRenderWindow_hasFocus (C function), 24
sfRenderWindow_isOpen (C function), 24
sfRenderWindow_pollEvent (C function), 24
sfRenderWindow_setFramerateLimit (C function), 24
sfRenderWindow_setKeyRepeatEnabled (C function), 24
sfRenderWindow_setMouseCursorVisible (C function), 24
size (C member), 19
str (C macro), 19
sz (C macro), 18

## T

tcs (C macro), 19
type (C function), 19

## U

u16 (C macro), 18
u32 (C macro), 18
u64 (C macro), 18
u8 (C macro), 18
uint (C macro), 19
ullong (C macro), 19
ulong (C macro), 19
uptr (C macro), 19

## W

wcs (C macro), 19