Unit tests – Practical session

SÉBASTIEN VALAT – CPPM - MARSEILLES – 24/01/2022

Plan

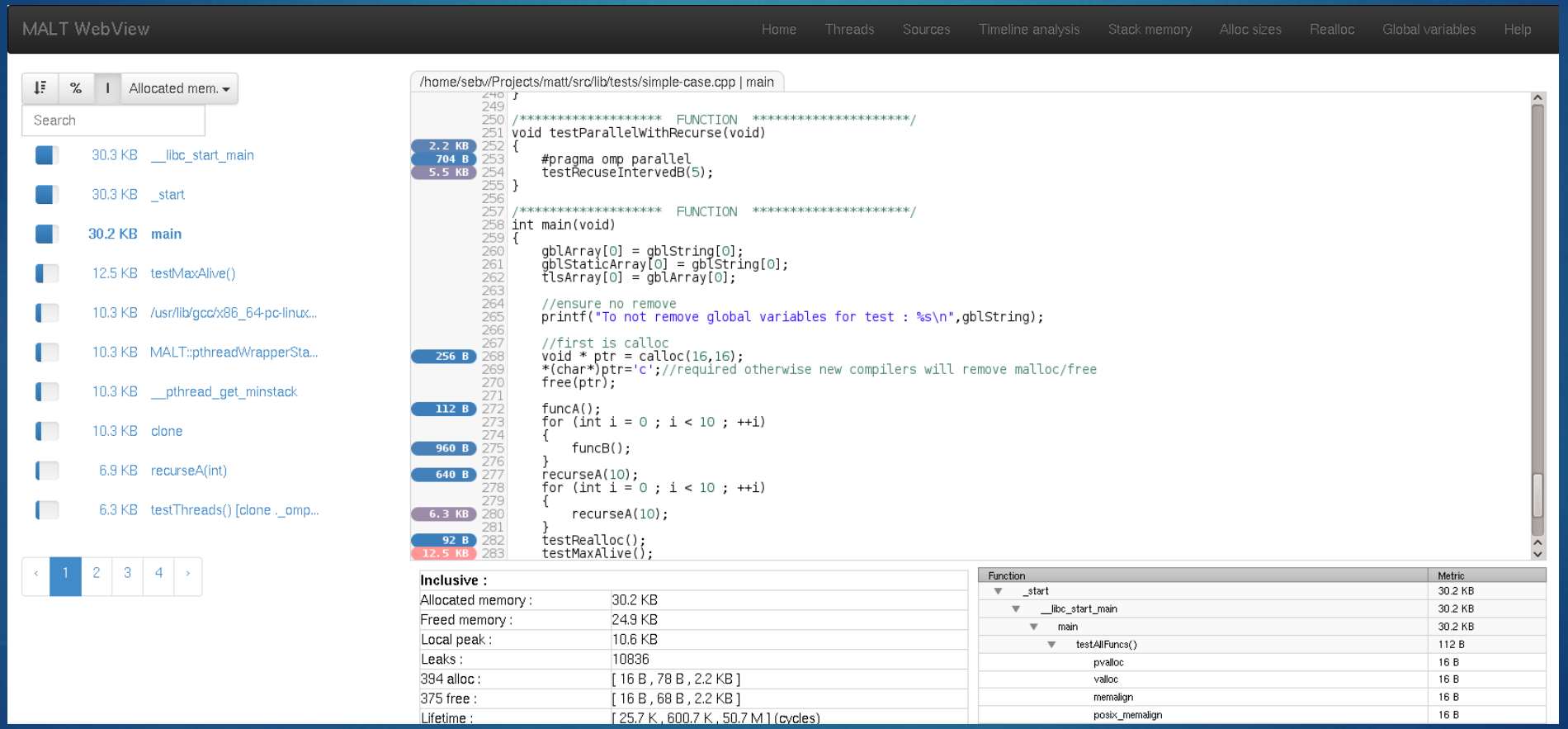
1. Explore one benefits of unit tests (C++)
   - Find a bug in an **unknown product**
   - Time with **integration tests** or **unit tests**
2. Implement a **basic unit test** on a particle (C++/Python)
3. Implement tests by **mocking** on a cache (C++/Python)
4. Implement more realistic tests on a task scheduler (C++/Python)

Copy the source dir


```
# clone the source repo
git clone /home/admin/sebv/unit-test-tp.git
```

Debugging with tests

MALT



https://memtt.github.io/

Build


```
# go in the source direcory
cd part-1-malt-buggy/malt

# create a build directory & move in
mkdir build
cd build

# create a build directory & move in
../configure --enable-debug --prefix=$HOME/usr
make
```

Run

- Forward port via SSH (to redirect the webview)

```
ssh server -L8080:localhost:8080
```

- Profile and launch the webview

```
# profile
~/usr/bin/malt ./tests/test-main
# launch the webview server
malt-webview -i malt-test-main*.json
```

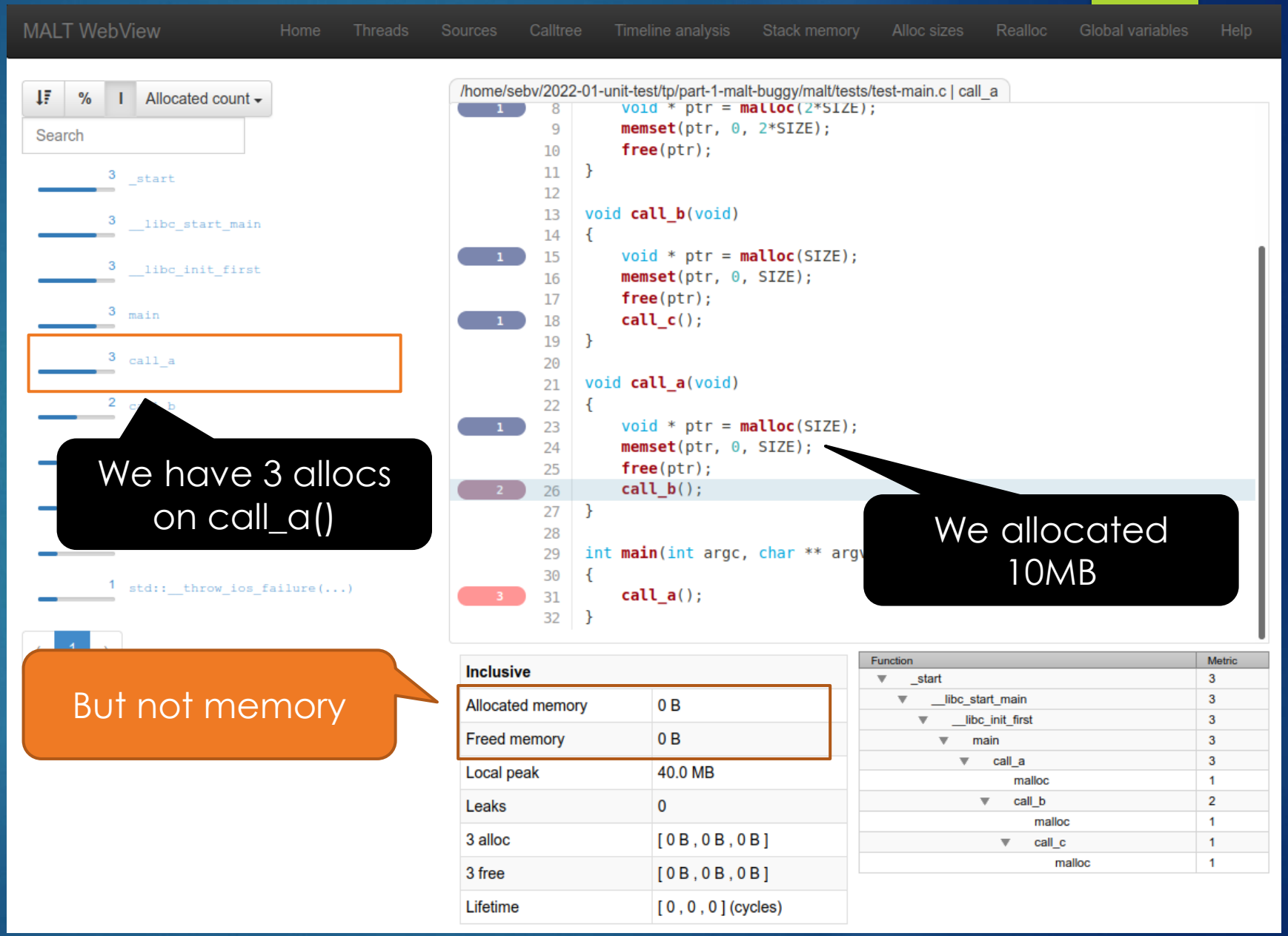
- Open your browser on http://localhost:8080

A small example


```c
#include <stdlib.h>

void call_c(void) {
    void * ptr = malloc(SIZE);
    free(ptr);
}

void call_b(void) {
    void * ptr = malloc(SIZE);
    free(ptr);
    call_c();
}
```

```c
void call_a(void) {
    void * ptr = malloc(SIZE);
    free(ptr);
    call_b();
}

int main(void) {
    call_a();
    return EXIT_SUCCESS;
}

# main > call_a > call_b > call_c
```

The bug


MALT WebView
Home
Threads
Sources
Calltree
Timeline analysis
Stack memory
Alloc sizes
Realloc
Global variables
Help
Allocated count
Search
3 _start
3 __libc_start_main
3 __libc_init_first
3 main
3 call_a
2
1 std::__throw_ios_failure(...)
/home/sebv/2022-01-unit-test/tp/part-1-malt-buggy/malt/tests/test-main.c | call_a
8 void * ptr = malloc(2*SIZE);
9 memset(ptr, 0, 2*SIZE);
10 free(ptr);
11 }
12
13 void call_b(void)
14 {
15 void * ptr = malloc(SIZE);
16 memset(ptr, 0, SIZE);
17 free(ptr);
18 call_c();
19 }
20
21 void call_a(void)
22 {
23 void * ptr = malloc(SIZE);
24 memset(ptr, 0, SIZE);
25 free(ptr);
26 call_b();
27 }
28
29 int main(int argc, char ** argv
30 {
31 call_a();
32 }
We have 3 allocs on call_a()
We allocated 10MB
But not memory
Inclusive
Allocated memory 0 B
Freed memory 0 B
Local peak 40.0 MB
Leaks 0
3 alloc [0 B , 0 B , 0 B]
3 free [0 B , 0 B , 0 B]
Lifetime [0 , 0 , 0] (cycles)
Function
Metric
_start 3
__libc_start_main 3
__libc_init_first 3
main 3
call_a 3
malloc 1
call_b 2
malloc 1
call_c 1
malloc 1

Search the bug - integration test

- You can run an integration test

```
ctest
```

- In vebose mode & only this one

```
Ctest -V
ctest –V –R integration-test-func-stats
```

Try for **10 minutes !**

With unit tests

- Enable the unit tests

```
../configure --enable-debug --enable-tests
```

- Build and run the tests

```
make
make test
```

Select the test, run it in **verbose**
and go in the test **source code**
Try to find the bug !

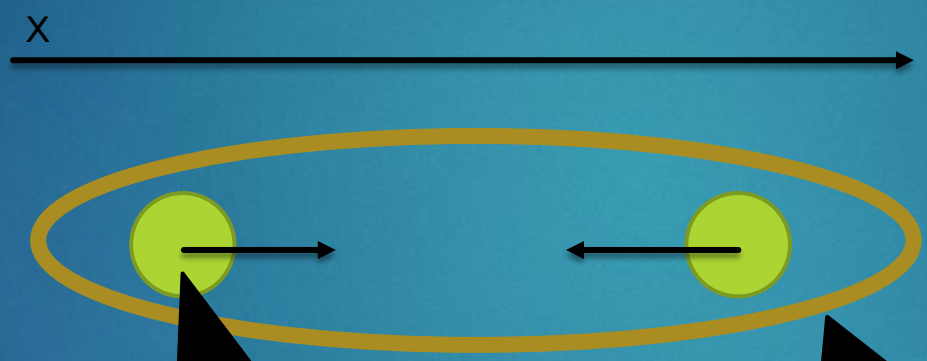
About the frameworks

Some framework

Language	Test framework	Mocking
Python	**unittest**	**unitest.mock**
C++	**Google test** *Catch2* *Boost test library* *cppunit* …	**Google mock** *FakeIT*
C	*Google test* *Criterion*	
Bash	*bats*	
Rust	*[native]*	*mockall*
Go	*[native]*	*gomock*

Implement your first unit tests

The model



Define a Particle in **1D** space with : **pos_x**, **vx**, **mass**

Define Physiscs interactoin: **elastic_collision()**, **collide()**

Go in « **part-2-simple-ut-particle** »

- For python

```
# move in
cd python
# run tests
pytest-3
```

- For C++

```
# move in
cd cpp
# create build dir & move in
mkdir build && cd build
# build
cmake .. && make
# run tests
ctest -V
```

TODO BUILD WHAT TO DO TO USE IT

- For python
- Nothing to do except importing
  - **unittest.TestCase**
  - **Unittest.mock**
- For C++
- In cmake:

```
find_package(Gtest REQUIRED)
include_directories(${GTEST_INCLUDE_DIRS})

Add_executable(test ...)
target_link_libraries(test Gtest::Gtest
Gtest::Main)
```

- In sources:

```
#include <gtest/gtest.h>
Using namespace testing;
```

A basic test example

- Python

```
class TestParticle(TestCase):
    def test_func(self):
        res = func(10);
        self.assertTrue(res)
```

- C++

```
TEST(TestParticle, func)
{
    bool res = func(10);
    ASSERT_TRUE(res);
}
```

Implement the test

- Python

```
# assert a boolean
self.assertTrue(variable)

# assert a value
self.assertEqual(100, variable)

# for boolean you can
self.assertAlmostEqual(1.0, variable)
```

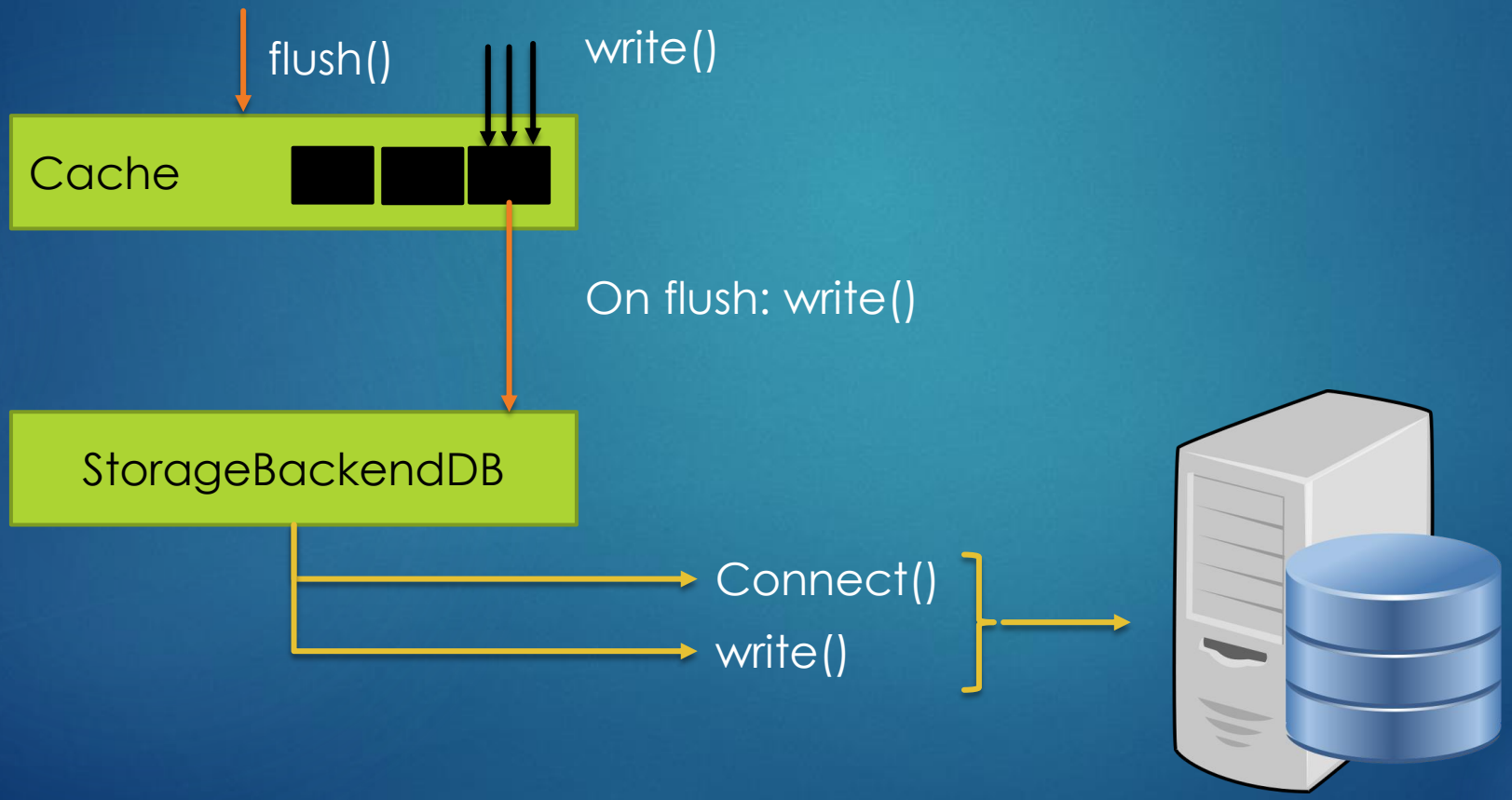
- C++

```
# assert a boolean
ASSERT_TRUE(variable)
EXPECT_TRUE(variable)

# assert a value
EXPECT_EQ(100, variable)

# for boolean you can
ASSERT_DOUBLE_EQ(1.0, variable)
ASSERT_NEAT(1.0, variable, error)
```

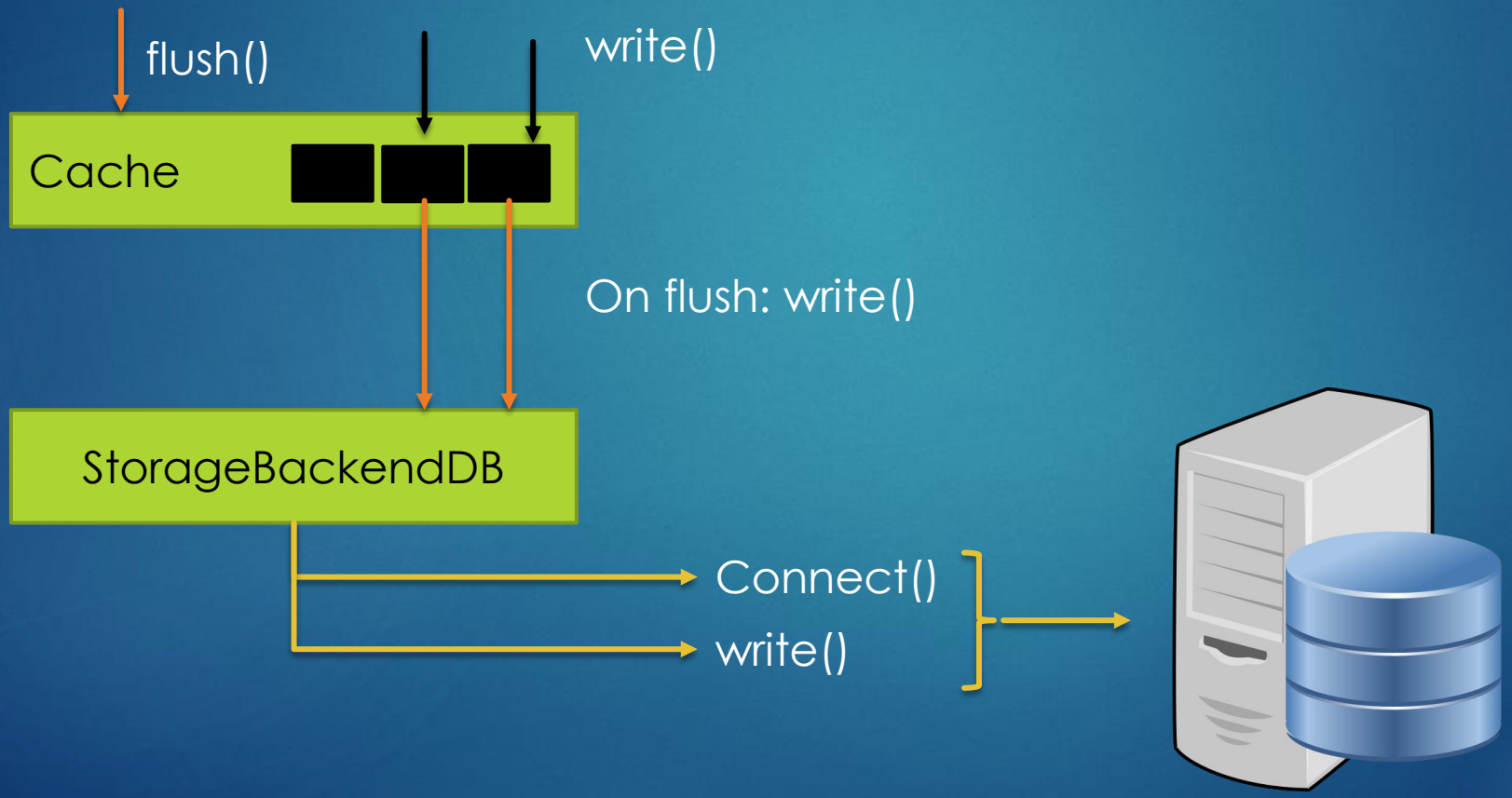
Implement test for:
Particle: **constructor()**, **move()**,
Phsyics: **collide()**, **elastic_collision()**

Mocking

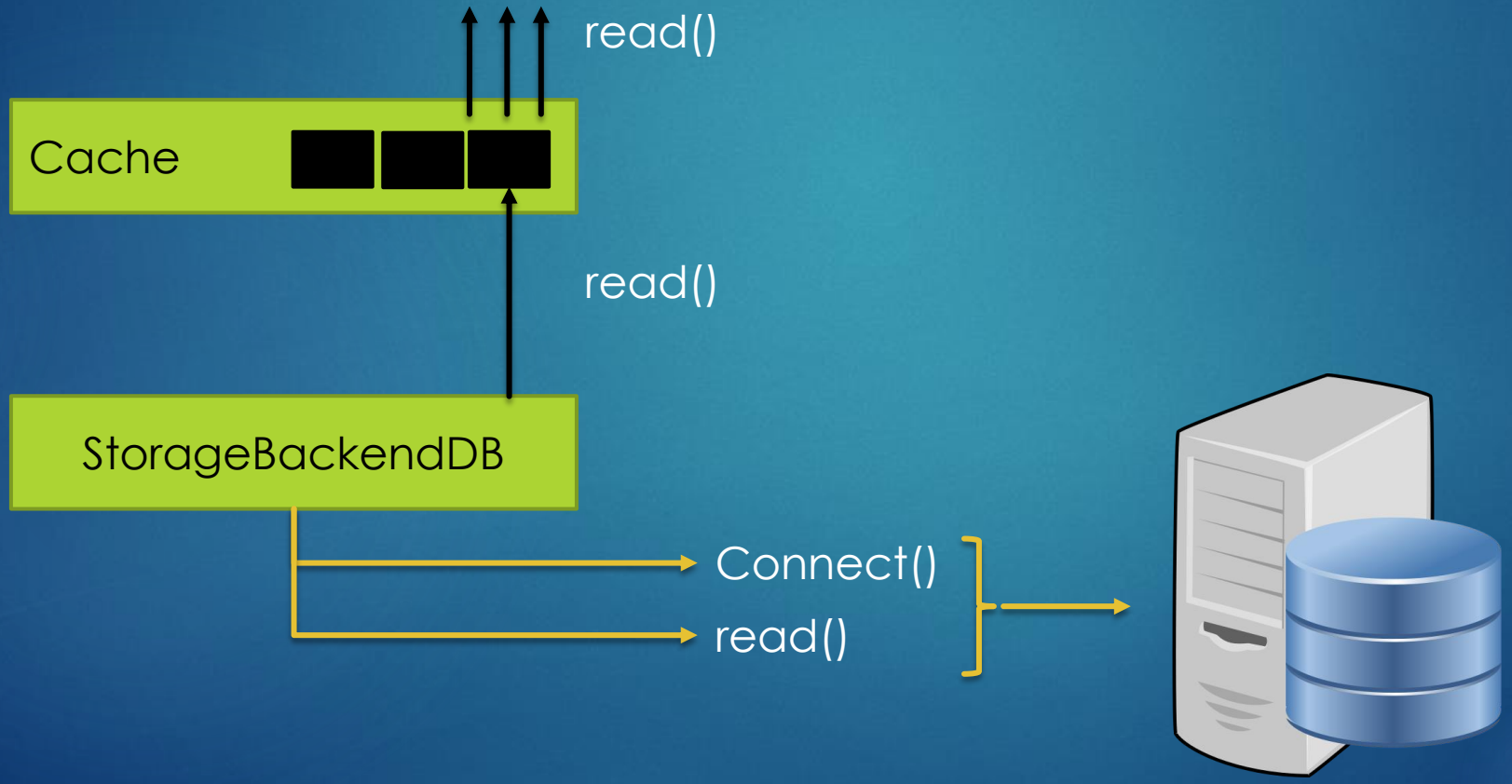
Implementation of a cache



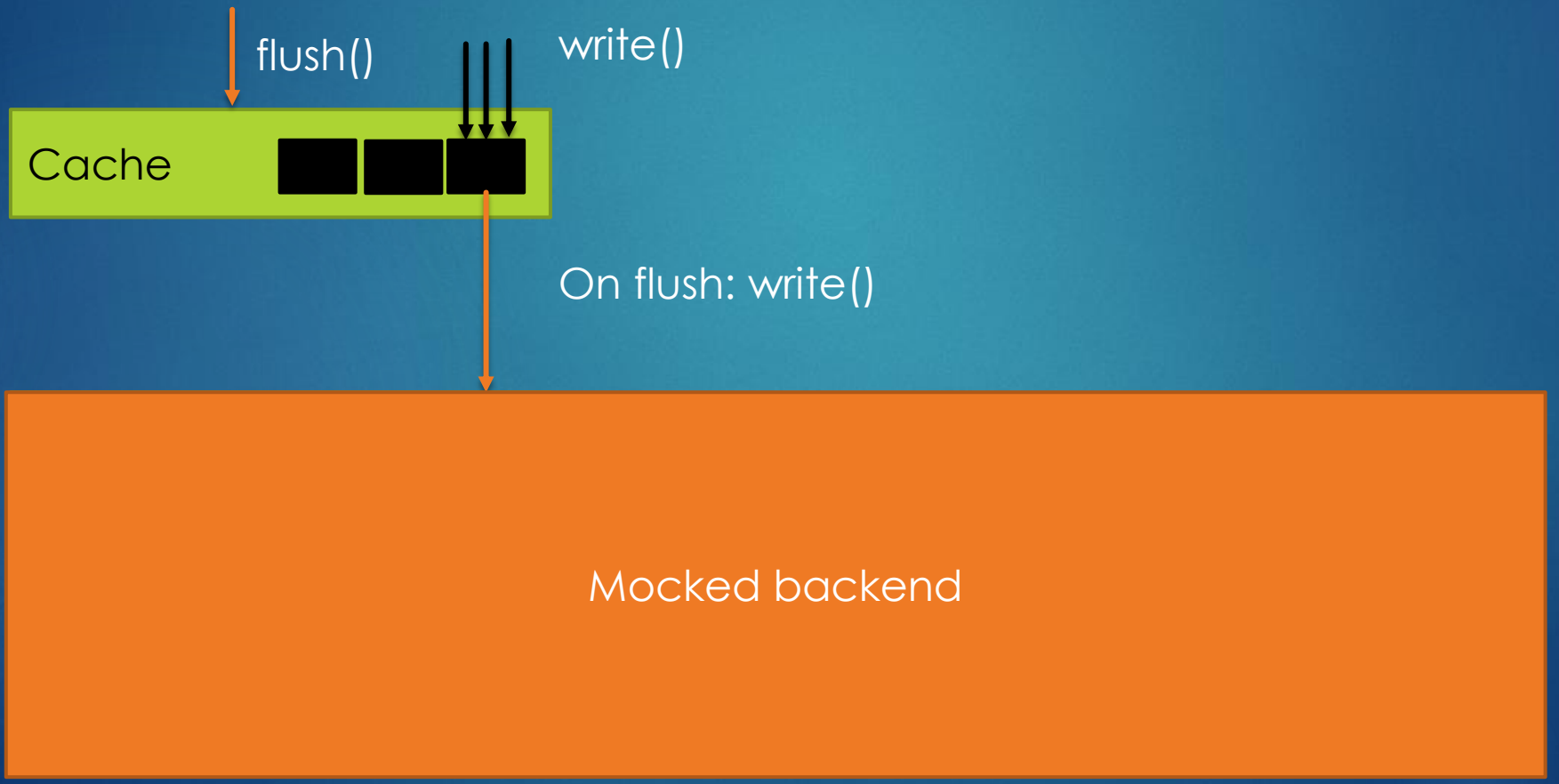
Implementation of a cache



Implementation of a cache



Implementation of a cache



2 approches to mock

- **Manually**
- **Fake implementation** of the storage backend
- Use a **mocking framework**
- In **python** we will use **unitest.mock**
- In **C++** we will use **Google Mock**

Go in « **part-3-cache-mock** »

- For python

```
# move in
cd python
# run tests
pytest-3
```

- For C++

```
# move in
cd cpp
# create build dir & move in
mkdir build && cd build
# build
cmake .. && make
# run tests
ctest -V
```

Basic usage

- For python

```
# build backend
backend = StorageBackend

# build the cache
cache = Cache(backend)

# read/wrote
cache.pwrite(b"Hello", 0)
cache.pwrite(b"World", 5)

# read/wrote
data = cache.pread(0, 10)
# data should contain
# b"HelloWorld"
```

- For C++

```
# build backend
StorageBackend backend;

# build the cache
Cache cache(&backend);

# read/wrote
cache.pwrite(Buffer("Hello"), 0)
cache.pwrite(Buffer("World"), 5)

# read/wrote
data = cache.pread(0, 10)
# data should contain
# Buffer("HelloWorld")
```

Manual approach

- You build a class inheriting from StorageBackend
- The implementation should always return a static value.

```
Class MockBackend(StorageBackend):
    def pread(…):
    def pwrite(…):
```

```
Class MockBackend : public StorageBackend
{
    public:
        virtual pread(…) override {…};
        virtual pwrite(…) override {…};
}
```

Implement two simple tests

- Instanciate a mock backend
- Instanciate a cache
- Call read
- Check the result of read

- Instanciate a mock backend
- Instanciate a cache
- Call write
- Check the result of write

Basic usage

- For python

```
# build backend
backend = StorageBackend

# build the cache
cache = Cache(backend)

# read/wrote
cache.pwrite(b"Hello", 0)
cache.pwrite(b"World", 5)

# read/wrote
data = cache.pread(0, 10)
# data should contain
# b"HelloWorld"
```

Create a **manual of backend**
Make a **read test**
Make a **write test**

Using a mocking framework

- We have a class to mock

```
class ToBeMocked:
    def func(value):
        # do complex stuff
        return value + 10
```

Create a **mocked backend**
Fill the tests

- Mock it

```
obj = ToBeMocked()
obj.func = mock.MagicMock(return_value = 20)

self.assertEqual(20, obj.func)

obj.func.assert_called_once_with(10)
```

Using a mocking framework

- We have a class to mock

```
class ToBeMocked:
    def func(value):
        # do complex stuff
        return value + 10
```

- Mock it

```
obj = ToBeMocked()
obj.func = mock.MagicMock(return_value = 20)

self.assertEqual(20, obj.func)

obj.func.assert_called_once_with(10)
```