SAPIENZA
UNIVERSITÀ DI ROMA

Using compiler plugins for mocking C code

School of Engineering in Computer Science
Master di secondo livello in Engineering in Computer Science

Candidate
Luca Ciavaglia
ID number 1248286

Thesis Advisor
Prof. Camil Demetrescu

Co-Advisors
Dr. Mattias Eriksson
B.Sc. Anders Nilsson

Academic Year 2013/2014

Using compiler plugins for mocking C code
Second level master thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: luca.ciavaglia@gmail.com

Dedicated to
my family

Abstract

For any software it is important that it is possible to write tests to ensure that the software is working as intended; but when the software is large, it is convenient test part of it in isolation. The goal of this project is to make isolated testing of individual C files easy by using GCC plugins. Any function calls that are made from the tested file to an outside function should be mocked automatically, using information that is available in the compiler.

It is possible through plugins, to access at the abstract syntax tree (AST) built when GCC parses the input file. In the AST we can find all the information necessary to mock such functions. To perfom automatic mocking, the external function of the software to test are replaced with the new functions generated via plugin.

In this thesis, we explain how we have developed a system able to perfom automatic mocking.

The thesis has been completely realized at the Baseband Tools department of Ericsson AB in Linköping, Sweden.

Acknowledgments

I would like to thank my advisor Professor Camil Demetrescu to have followed me at distance. I would like to express my special appreciation and thanks to my co-advisor Dr. Mattias Eriksson, you have been a tremendous mentor for me, the best that I have ever had. I would like to thank you for encouraging my research and for allowing me to grow as a software developer. Your advice on both research as well as on my career have been priceless. I would like to thank my co-advisor Anders Nilsson, you give me the possibility to do this wonderful experience in Ericsson.

A special thanks to my family. Words cannot express how grateful I am to my mother and my father for all of the sacrifices that you've made on my behalf, you have allowed me to stay far from home for so many months. I would also like to thank all of my friends who supported me to strive towards my goal. At the end I would like express appreciation to my corridor-mate to support me in all the moments.

Contents

Chapter 1

Introduction

For any software it is important to write tests in order to ensure that the software is working as intended. Software system code has dependencies, especially if the system is large. Dependencies can cause problems as slow system builds and high risk of introducing bugs. If the build time and execution time are long, it is also difficult to run all the tests frequently. To remedy these problems it is convenient to test parts of the system in isolation. In this way we do not have to compile the whole application but just a selected function. To isolate the software's dependencies, we can use a mechanism called stubbing, in which the dependencies are replaced with fake implementations of them, called stubs. Stubs are written manually and this process is very time-consuming, moreover they must be updated whenever the dependence's imitated are also updates. These problems can be solved by using a technique called *mocking*, in which the behavior of the software dependence is controlled at run-time. The dependence's, this time, are replaced with mock functions. In a particular test case, it is possible to set the behavior of the software dependencies. This allows to have different test cases in which all of them can work with different expectations but always using the same *mock functions*.

This thesis project aims at making mocking easier in C by utilizing the plugin mechanism in the GCC compiler. By using the available information in the compiler, the system is able to intercept automatically any function calls which are directed outside of the currently tested C file and redirect them to *mock functions*. The project has been realized at Ericsson AB, Linköping, Sweden.

The System has been developed in two main components: a GCC plugin that detects and redirects the function calls, and a run-time system that checks the behavior of the mock functions.

In this thesis, the software supposed to be tested is called System Under Test (SUT). In Figure 1.1 is shown a SUT with its software dependencies. In this representation, the SUT calls the external functions defined different files: *dependencyA*, *dependencyB* and *dependencyC*. However, we mean to test the SUT in isolation, we suppose to not have such dependencies or files.

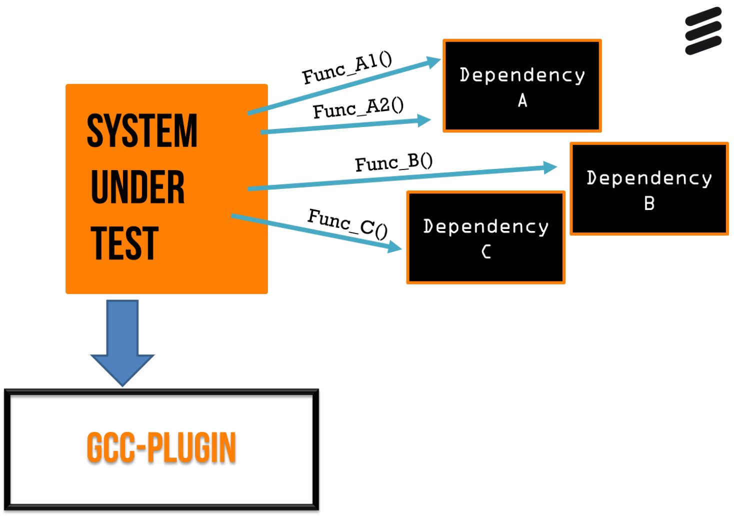

Figure 1.1. The System Under Test intercepts the function calls which are directed outside of the tested file.

This project aims to isolate the SUT, and to achieve this goal, the SUT is given as input to the GCC compiler plugin that detects the functions to be mocked and redirects them to the new mock functions as shown in Figure 1.2.

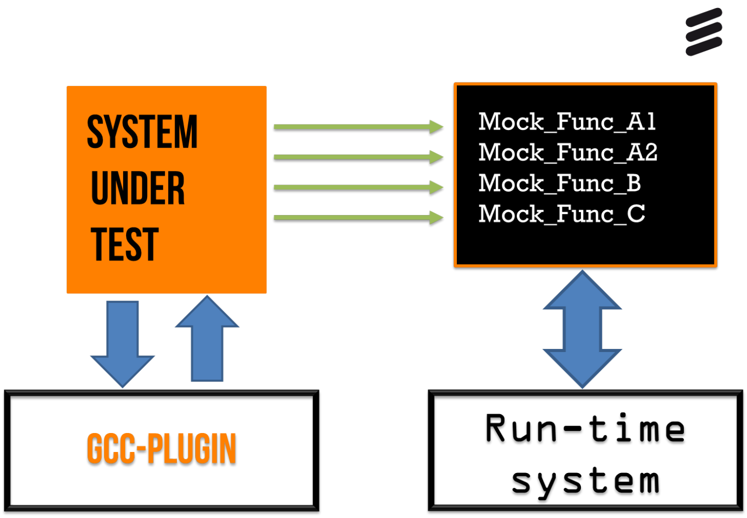

Figure 1.2. The functions are redirected to mock functions that communicate with the run time system.

The mock functions are generated when the plugin reads the SUT and then they are connected with the runtime-system. Through the plugin, we can access the available information in the compiler. In particular, we can read and manipulate the *abstract syntax tree* (AST) that is built when GCC parses the input file. The mock functions generated can either be done by inserting any missing functions in the AST or by letting the plugin generate a new temporary C file which contains the needed mock functions. In this project we chose to use the method of generating new temporary files.

Ericsson's specifications required that the runtime system provides an API which allows the user to define the behavior of the mock functions as follows:

Consider an external function that is called from the file we want to test:

```
int foo(int x);
```

Controlling the mock function should be done like in the following example:

```
expect_foo(ANY_INT, 5);
expect_foo(INT(2), 1);
```

In which we have specified that we expect *foo* to be called two times. The first time it accepts as parameter in input any integer and will return 5. The second time *foo* accepts only the integer parameter 2 and will return 1. It is requested to further investigate the exact API to be used in this project.

One open question was also how to handle parameters with user-defined types (such as a struct).

1.1 Questions

In Ericsson there is an internal software called Swift. Swift adapts Google Mock to DSP-C (a C dialect) and allow tests to be run on linux. It parses the SUT (written in DSP-C) by reading Ctags and it automatically generates mock objects. A drawback of Swift is that it uses another method of parsing than the real compiler does.

Unfortunately, Swift is an internal project and we cannot describe it in more depth. By having this brief knowledge about Swift, in thesis project we aims to answer at the following questions:

Is it possible to use plugin compilers to automatically create mocks for external functions defined in C language?

By using plugin compilers, we want investigate the possibility to realize a new kind of parser that it is supposed to work as the real compiler does.

Moreover, we can not link C code with C++ code in all the available platforms. In Swift, the SUT's dependence's (in C) are linked with the mock objects (in C++). In addition, not in all platform we have a C++ compiler. An example are the digital signal processor (DSP) platforms, in which it is not possible to link these two languages.

Hence, the next question to investigate is: *Can we create a runtime system that is almost powerful as G oogle mock in pure C?*

1.2 Project scope

The project scope of this thesis is to make mocking easier in C by utilizing the plugin in GCC by following these indications:

- All the external functions must be replaced from new mock functions.
- The plugin should be able to read the primitive data types as: *double*, *float*, *int*, *short*, *long int*; and and the user-defined structures, where they are used in the external function detected.
- The runtime system needs to be supported with an API to set the external function behaviors.

1.3 Development choices

In this section we want to describe the development choices made and their reasons.

First of all, we have chosen GCC as compiler. GCC is one of the most used compilers. It supports many of the present architectures and it is known as a solid and stable tool. Moreover, there is not documentation about how to create plugins in different compilers. For example in Clang[1], the plugin interface is basically undocumented and the preferred method is to directly modify the Clang source code. Summarizing, the main features of GCC are:

- GCC is open-source,
- cross-platform and
- widely-deployed.

The second important choice is that we create a temporary file, in which write the mock functions generated by the plugin. This allow us to read the generated source code; this is a big advantage during the debugging phase. It could be hard to debug the plugin if we manipulate directly the external function in the AST.

Finally, the runtime system is mainly divided in two parts. One part is generated at compilation time by reading the SUT, the runtime system SUT dependent part; it is contained in a temporary file. Whereas the other part is fixed: the runtime system SUT independent part. Both are built to work together.

1.4 Project architecture

This project follows a determined order of compilation steps, all of them are managed through a makefile. We will see now how the project works.

As stated previously, the term System Under Test or briefly SUT, indicates the piece of software that we aim to test. We suppose that it is contained in a file called *sut.c*.

During the first step, the SUT external functions are intercepted and new *mock functions* created. This is done by compiling the *sut.c* file with GCC and the plugin loaded, avoiding the compilation linking phase. The result is firstly, a new generated file (created through the plugin) called *temporary_file.cpp*, it contains all the new mock functions; and secondly the SUT object file.

The next compilation step is to obtain the object file for the temporary file. Therefore, also in this step the compilation is done avoiding the GCC linking phase. What we get is the *temporary_file.o* file.

The behavior of the mock functions are set in a third file called *test_sut.c*, here is the place in which we write the function expectations to set the expected behaviors of the mock functions. This file is written from the software tester by using an API. Such API is provided by including the file *framework.h*.

The expectations are created into the *temporary_file.cpp* together with the mock functions. Hence, what *test_sut.c* does is to call the expectation functions contained into the temporary file by passing the proper arguments. Also this time, *test_sut.c* is compiled without the linking phase to obtain the object *test_sut.o*.

Finally, in the last step, the SUT's external functions have to be replaced with the new mock functions generated. The redirection of the SUT external functions to the new mock functions is performed by linking the *sut.o* with *temporary_file.o* files. In addition, in *test_sut.c* are called both the expectation functions contained into the *temporary_file.o* and the functions that should be tested (in *sut.c*). Hence, the *test_sut.c* file needs to be linked with *sut.o* with *temporary_file.o*. These last two tasks can be done in a linking step by executing the linking between all these three files, in such way we obtain as result, an executable file. Note that the program's *main* is contained into the *test_sut.c*; in this file, it must be called the function supposed to be tested.

By running the executable file, we test the SUT and we can see if it works in accordance with the mock functions' expectations.

Let us suppose to test a piece of software contained into the file *sut.c* and that its source code is shown in Listing 1.1.

Listing 1.1. *sut.c* - Example.

```
#include <stdio.h>

int foo(int y, double x); //external function

void function_to_test()
{
  int result = foo();
  //...

}
```

By compiling the *sut.c* file with GCC and the plugin we obtain two files: the *temporary_file.cpp* and *sut.o*. The temporary file contains both mock functions and expectation functions. It contains also the structures to save the expectation settings through the expectation functions. Listing 1.2 shows a basic source code for the temporary file to clarify how it can be implemented.

Listing 1.2. *temprary_file.c* - Example.

```
#include "helper.h"
struct Mock_foo{
  int switch0; int param0; int (*userfunc0)(int x);
  int switch1; double param1; int (*userfunc1)(double x);
  list<Expectation*> list_expectations;
};
list<Mock_foo*> list_foo;

extern ``C''
void expect_foo(/*parameters*/){
  /*
    Save the expectations
  */
}

extern ``C''
int foo(int y, double x){
  /*
    Check if the expectation
    are respected
  */
}
```

Next, the behavior of the external functions need to be set. This is done with the *test_sut.c* file; and the source code of such file follows in Listing 1.3. In this file is also called the function supposed to test, that for this particular example is *function_to_test*. In general, the tester needs to write only this file.

Listing 1.3. test_sut.c

```
#include ''framework.h''
void test_sut(){
    //Set the expectation
    expect_foo(ANY_INTEGER, DOUBLE(5.6), 9 );

    //Execution: Call the function to test
    function_to_test();
}
int main(int agrc, char** argv){
   test_sut();
   return 1;
}
```

The API used in Listing 1.3 is only an example to show the general idea. We will see in the runtime system chapter how it has been implemented. In this case, we mean that *foo* should be called only once with any integer as first parameter and the *double* value of 5.6 as second parameter; moreover the mocked function *foo* should return an integer value equal to 9.

Now that the files, *temporary_file.cpp* and *test_sut.c* are created, the next step is compile them to obtain their object files. Finally the files *temporary_file.o*, *test_sut.o* and *sut.o* are linked together to obtain an executable file. By running the executable file, that we called simply *result*, we can see if the function works as expected.

All the compilation steps are performed automatically by commands set in a Makefile. Hence, the tester needs just to write the file for set the expectations and run the Makefile. Obviously the file names, as *test_sut.c* or *sut.c*, are only an example. The tester can choice the names of them and modify easily the Makefile. In this paper, we will keep such names so to keep the explanation clear.

Chapter 2

Background

In the last years the use of mocks is spreading widely among software developers thanks to the new agile methodologies such as the Extreme Programming and the Test Driven Development. For such reason, these methodologies are described briefly during this chapter.

We will speak about mock objects since the mocking mechanism is common in an object oriented language. We will also clarify with an example the difference between stubbing and mocking.

We will describe the available mock frameworks in Java, C++ and C; the digital signal processing processors and the general architecture of compilers.

2.1 Software testing

Software testing is any activity aimed at evaluating a software, determining that it meets its required results. The software to test can be executed both alone or together with a service software to evaluate if it respects the requirements. It can be implemented at any time during the software development process. Traditionally software testing is performed when the software has been completed (or almost completed), however lastly, with the introduction of the agile approaches, testing is implemented in an earlier stage, depending on the chosen software development methodology.

There are different methods of testing, one of them is the *box* approach. The box approach is divided in *white box* and *black box* testing. In the first approach, it is verified the internal structures of the software, hence it is necessary to have internal knowledge of the system. The white box approach works by testing a single unit. Whereas, in black box approach, the test is performed only in according on what the software is supposed to do, without any knowledge of the internal system.

Tests are also grouped in levels:

- Unit testing. Unit testing verifies the functionality in a specific function, object or in general a specific section of the code. It uses the white box approach.
- Integration testing. In integration testing, the individual software units are combined and tested together. The tests works following the black box approach, in which the verification is done by knowing what each unit should do.
- System testing. System testing is performed after the unit and integration testing. It aims to test the integrated software components that have passed the integration testing. Hence, it tests the whole system. Also this last level of testing uses the black box approach.

2.2 Test automation

Test automation attempts to automate repetitive tests made on the software under test[17]. Test automation uses a special software and there are a large number of available frameworks. As stated in Meszaros's book [19], the goals of test automation are:

- Tests should help us to improve quality.
- Tests should help us to understand the SUT.
- Tests should reduce (and not introduce) risk.
- Tests should be easy to run.
- Tests should be easy to write and to maintain.
- Tests should require minimal maintenance as the system evolves around them.

2.3 Xunit

xUnit is a family of test automation frameworks. It makes easy for developers to write tests without learning a new programming language. In fact, xUnit is available in most languages. It makes also easy to run one or many test with a single action. xUnit provides a set of assertions to verify the state of the system under test. They are basically functions or macros in which we specify the SUT's behavior.

All xUnit frameworks share the same architecture, with some varied implementation details. xUnit test pattern is composed of four main phases:

- Setup. During the setup phase the necessary variables to execute the test are created and set to a proper state.

- Exercise. The exercise phase calls the part of the code that we want to test.

- Verify. Here, all the assertions are verified.

- Teardown. The teardown phase deletes all the variables created during the setup phase.

2.4 Agile Software Development

Agile software development is a group of software development methodologies based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams[22]. Agile software development is defined in the Manifesto for Agile Software Development[16]. The Agile Manifesto has been written by 17 software developers in February 2001. It is partly reported here:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - Individuals and interactions over processes and tools
> - Working software over comprehensive documentation
> - Customer collaboration over contract negotiation
> - Responding to change over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

Agile methods state a set of key values used to create and describe lighter and faster application development methodologies that are more focused on people, working software, and results. Agile methods prefer a face-to-face communication. When a team works in different locations, they keep in contact daily through videoconferencing, voice, e-mail, etc. One key factor is that the customer is part of the team, in this way the final product is always as expected from him. There are techniques and tools used to improve quality and enhance project agility such as continuous integration, automated or xUnit test, pair programming, test driven development, design patterns, domain-driven design and code refactoring.

2.5 Extreme Programming

Extreme Programming (XP) is one of several popular agile processes[12]. The Extreme Programming key factors are: pairs programming (PP), test-first programming (TF), frequent re-engineering of software, avoiding programming of features until they are actually needed and simplicity in code. But also: expecting changes in the customer's requirements as time passes and the problem is better understood, frequent communication with the customer and among programmers and the continuous verification of the program during the development by test programs. In particular, the Extreme Programming rules about *Testing* are the following:

- All code must have unit tests.
- All code must pass all unit tests, before that it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.

2.6 Test Driven Development

Test Driven Development (TDD) is a software development process in which the real development is preceded by writing automated test case. The history of TDD starts in 1999 with a group of developers who were inspired from a set of Extreme Programming concepts as the test-first programming and code refactoring[1][14]. The process is structured on the repetition of short development cycles (TDD cycles). Each cycle is made of three steps. First the developer writes an (initially failing) automated test case that defines a desired improvement or a new function. Then, the developer produces the minimum amount of code to pass that test; and finally he refactors the new code to acceptable standards.

TDD aims to create software in an incremental way, it improves the quality and the design of the software by simplifying the development process. By running all the previously tests, the software developer is confident to not have inserted into the system changes that can cause unexpected side effects.

[1]Refactoring is an approach to changing the design without changing the behavior of the code.

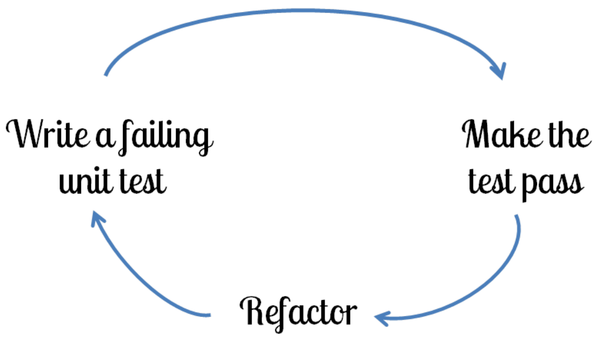

Figure 2.1. The TDD cycle.

According to McConnel's book [18], there is a research at Hewlett-Packard, IBM, Hughes Aircraft, TRW, and other organizations in which it has been found that detecting an error at the beginning of the software construction allows to solve the error in 10 to 100 times less expensively than when it is solved in the last part of the process.

Hence, TDD could probably be the right approach to use; instead than other common software engineering approach as for example the waterfall model, in which the software testing phase is performed when the implementation phase of the software is finished.

However, there are not researches that establish TDD better than others software engineering approach, and it is impossible to test completely a software.

2.7 Mocking in an Object Oriented language

In this paragraph, we will describe the mocking technique in an object oriented language. We will show how to implement mocking in C code in the next chapters.

Nearly all of the publications concerning Extreme Programming or Test Driven Development use an object oriented language, since they have based on concepts of polymorphism and modularity. Mock objects came out of the XP community and their use is widely spreading thanks to TDD. For these reasons, we prefer introduce the concepts with example in an object oriented language.

2.7.1 Test Doubles

According to Meszrous [19]: in an object oriented language, when a class does *not* depend on any other classes, testing it is relatively straightforward and the following techniques described here are not necessary. When a class *does* depend on other classes, we have two choices: we can test it together with all the other classes in which it depends on, or we can try to isolate it from the other classes so that we can test it by itself.

When the system under test depends on other classes, it is always preferable isolate it. This has a big advantage: we are sure that our SUT is not conditioned by possible bugs present on its dependent classes.

The dependencies are isolated with particular kind of objects called test doubles; and continuing to follow Meszrous [19], there are four different types of test doubles:

- *Dummy object.* A dummy object is a placeholder object that is passed to the SUT as an argument (or an attribute of an argument).
- *Test stub.* A test stub is an object that replaces a real component on which the SUT depends. Consequently, the test can control the SUT's input directed to the dependent object.
- *Mock object.* A mock object is an object that replaces a real component on which the SUT depends. In this way, the test can set expectations and verify the entire dependency's behavior.
- *Fake object.* A fake object is an object that replaces the functionality of the real dependency with an alternative implementation of the same functionality.

2.7.2 Examples

The purpose of these examples is to clarify the concepts of *stubbing* and *mocking*, since they are often misunderstood. The examples are created from Martin Fowler[15]; they are developed in Java and they use the JUnit framework[10]. JUnit follows the typical xUnit test pattern made of four main phases: setup, exercise, verify and teardown.

Test via Stubbing

In this example, we aim to test is the *fill* function present in the Order object. Order maintains information about a product and its quantity. The *fill* function works together with the Warehouse object. Warehouse holds inventories of different products. In testing terms, Warehouse is called collaborator and Order is the system under test.

The behavior of Order.*fill* is: if Warehouse has enough quantity of the same product defined in Order, the order becomes filled and the warehouse's amount of product is properly reduced. Otherwise, the Order is not filled and the state of Warehouse does not change.

The situation described above is tested by using the code in Listing 2.1.

Listing 2.1. This code is from Martin Fowler[15].

```
public class OrderStateTester extends TestCase {
  private static String TALISKER = "Talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Warehouse warehouse = new WarehouseImpl();

  protected void setUp() throws Exception {// Setup phase
     : Part 1.
    warehouse.add(TALISKER, 50);
    warehouse.add(HIGHLAND_PARK, 25);
  }
  public void testOrderIsFilledIfEnoughInWarehouse() {
    Order order = new Order(TALISKER, 50); // Setup phase
       : Part 2.
    order.fill(warehouse);                 // Exercise
       phase.
    assertTrue(order.isFilled());          // Verify
       phase.
    assertEquals(0, warehouse.getInventory(TALISKER));
  }
  public void testOrderDoesNotRemoveIfNotEnough() {
    Order order = new Order(TALISKER, 51); //Setup phase:
        Part 2.
    order.fill(warehouse);                 //Exercise
       phase.
    assertFalse(order.isFilled());         //Verify phase
       .
    assertEquals(50, warehouse.getInventory(TALISKER));
  }
}
```

The setup phase is divided in two parts. The first one is implemented into the *setUp* method, whereas the second part depends on the kind of test: *testOrderIsFilledIfEnoughInWarehouse* or *testOrderDoesNotRemoveIfNotEnough*.

With the first test, we want to check if the *fill* method has removed the right quantity in warehouse; instead with the second we want to know if the quantity of product does not change.

As we can see by using the stubbing method, we are going to verify (with the assert methods provided from JUnit) if the collaborator (Warehouse) is in some acceptable state after the SUT's execution.

This is the *traditional* way to test, known as *state verification*. In state verification, first we exercise the SUT; then we examine the SUT's state or anything returned by the SUT (the collaborators), through assertions. What we cannot verify is how the SUT interacts with the collaborators. We examine only the object's state and we use only direct method calls as our observation points[19]. The last "limitation" is resolved by using mock objects.

Test via Mocking

This time, we will test the same objects as before but by using the mocking technique. In this example shown in Listing 2.2, Mock objects are created with the JMock[9] framework.

Listing 2.2. This code is from Martin Fowler[15].

```java
public class OrderInteractionTester extends
   MockObjectTestCase {
  private static String TALISKER = "Talisker";

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new Order(TALISKER, 50);
    Mock warehouseMock = new Mock(Warehouse.class);

    //setup - expectations
    warehouseMock.expects(once()).method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once()).method("remove")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");

    //exercise
    order.fill((Warehouse) warehouseMock.proxy());

    //verify
    warehouseMock.verify();
    assertTrue(order.isFilled());
  }
```

```java
public void testFillingDoesNotRemoveIfNotEnoughInStock
    () {
  //setup - data
  Order order = new Order(TALISKER, 51);
  Mock warehouse = new Mock(Warehouse.class);
  //setup - expectations
  warehouse.expects(once()).method("hasInventory")
    .withAnyArguments()
    .will(returnValue(false));
  //exercise
  order.fill((Warehouse) warehouse.proxy());
  //verify
  assertFalse(order.isFilled());
}
```

As for the stubbing example, we set up the environment by creating the collaborators that we need and set them in the proper state. Note that this time, the collaborator is not a real Warehouse object, but it is a mock object of the Warehouse class. Moreover we also set the Mock's expectations. In the verification phase the mocks object compare the real received calls with those defined in the expectations. Through expectations we are able to check the behavior of the collaborator object.

Using expectations, we can define which parameter a function should receive or what it should return, how many times it should be called; if the functions should follow a particular order between them; and many other kind of expectations.

With the Mocking technique we are not doing anymore state verification, but rather *behavior* verification.

2.8 Mock Frameworks in Java, C++ and C

In the past, the mocks were implemented manually and it was not very pleasant. It is a lot of work to create them by hand; and moreover they are not enough generic to be reused in multiple tests. As result many people avoided mocks. However, it is perfectly reasonable to create mocks by using a good library or framework, thanks at the computer capacity to do repetitive things.

Actually, for many languages we have some fine frameworks to create mocks. In Java, two popular frameworks are JMock[9] (as we have seen in the examples before) and EasyMock[3]. They use reflection of the language to dynamically create mock objects at run time.

For C++ there is Google Mock[8]. In C++, mocking is possible by using classes, inheritance and template programming.

Unfortunately in C, the powerful properties of the object-oriented languages are not available and it can be harder to implement mocks. However there are frameworks available also in C. One example is CMock[2] that allows us to create Mocks, but it has a drawback: the functions to be mocked are found by parsing the SUT with regular expression through a script written in Ruby [20]. This means that it uses a different parsing method than the real C compiler does.

At Ericsson, there has been an investigation on the CMock framework before creating the internal software Swift; and from this investigation, GMock was chosen over Cmock as mocking framework. In fact CMock has been considered less powerful then GMock. Moreover, as already stated, CMock uses Ruby and this is another big disadvantage.

Another interesting C library is TestDept[11]. This latter is a unit test framework for C with stubbing that allows us to change at runtime the function with another one defined by the user.

There are other C frameworks available, but in general they are not as powerful as GMock or Jmock.

Finally, mocking C code is slightly different from the use of Mock objects as in an object oriented language. In fact, in C we want be able to *mock* functions. However, the idea is still the same: mock functions need to be created in some way and their related expectations need to be set for each of them.

2.9 Compilers

According to Aho's book [13]: a compiler is a program that can read a program in one language, the *source* language, and translate it into an equivalent program in another language, the *target* language. A compiler has also the role to report any error in the source program that it detects during the translation process. The target program is an executable machine-language program. It is used by the user to process input and receive output. In contrast with the *interpreter*, in which it executes the operations specified in the source program including directly also the user's input. A target program produced by a compiler has the advantage to be usually much faster if compared with an interpreter.

The compiler is not the only program necessary to create an executable target program. In fact, a source program may be divided in separate files or it may have shorthands called macros. These two last tasks are performed by a separated program called *preprocessor*. The source code generated by the preprocessor is given in input to the compiler in which in turn, it produces in output an assembly-language program. An assembly language has the advantage to be easier to be produced and to be debugged. An *assembler* program processes the assembler language in order to obtain a relocatable machine code. Since programs are often large, the relocatable machine code may need to be linked together with other relocatable object files and library files. This last operation is solved through the *linker*. The linker resolves external memory addresses in which the code in one file refer to a location in another file.

2.9.1 The structure of a Compiler

All compilers are mainly divided in two parts: *analysis* and *synthesis*.

The analysis part, also called *front end* of the compiler, has the purpose to create a grammar structure of the source code and translate this grammar structure in an intermediate representation of the source code. Moreover, information about the source program are stored into a data structure called *symbol table*. The symbol table is then read in the synthesis part. The analysis part checks also if the source code is syntactically and semantically correct. In the contrary case, informative messages are provided to the user.

The synthesis part, or *back end* of the compiler, constructs the target program by reading the intermediate representation and the symbol table.

A machine-independent optimization part is introduced in some compilers between the front end and the back end. It performs optimization on the intermediate representation in order that the back end can produce a better target program. This part is called often *middle end* [13].

Intermediate Code Generation

As stated previously, an intermediate representation is built in the process in order to translate a source code to a target language.

Intermediate representation saves a considerable quantity of effort. In fact, by knowing that there are several different languages n; and m of different machines/-platforms, with an intermediate representation it is enough to build n front end plus m back end; instead to have n*m compilers.

A well defined intermediate representation should be easy to produce and easy to translate into the target machine.

In a compiler, there are different intermediate representations, mainly there are the high-level representations that are close to the source code; and a low-level representations close to the target machine. High level representations are the syntax trees in which they reflect the natural hierarchy of the source code, whereas low level representation are three-address codes, more suitable for tasks like register allocation.

2.10 Digital Signal Processor (DSP)

According to the W. Smith's book [21], digital signal processors (DSP) are microprocessors specifically designed to handle Digital Signal Processing tasks. They move samples into devices, perform the mathematical operations and give in output the processed data.

The DSP market is growing fast during the last years. DSPs are used in everything from cellular to advanced scientific instruments. The reason for this grown is that they are faster if compared with traditional microprocessor. Traditional microprocessors are built to work with *data manipulation*, such as word processing and database management; whereas DSP concerns on *mathematical calculation* used in science, engineering, and digital signal processing.

DSPs are usually programmed both in C or assembly languages and DSP programs are shorter than traditional software. In DSP, the execution speed is a critical part of the application. In DSP, it is important to consider the code size and memory footprint. The critical parts are programmed in assembly that allows to achieve the maximal efficiency. In the other side, C affords more flexibility and fast development, moreover in it is always possible to add assembly code for the critical section that must execute quickly.

Chapter 3

The GCC compiler

In this chapter, we will describe the GCC architecture and how we can create a GCC plugin. We will also explain how access and read the *abstract syntax tree*, generated when GCC parses a file.

3.1 GCC architecture

GNU Compiler Collection (GCC) is a collection of compilers for different programming languages. It is a driver program because it invokes the proper compilation programs in according to the language of the source file. For example, if the source file is written in C , the programs called by GCC are: the preprocessor, the compiler *cc1*, the assembler *as* and the linker *collect2*. GCC is composed by three part: front end, middle end and back end. GCC compiles one file at a time going through all three components one after another. The front end reads the source file, parses it and creates an *abstract syntax tree* (AST) representation. The AST may be different depending on the front end. The AST is then converted into a common form called GENERIC. The GENERIC's form is given in input to the middle end. This latter converts the GENERIC's form in a lower level representation form, called GIMPLE. In turn, the GIMPLE form is converted in the *static single assignment* (SSA) representation; and again it is converted back in GIMPLE. The middle end uses the GIMPLE and the SSA representation to perform optimizations on the source code. Next, GIMPLE is converted in a *register-transfer language* (RTL), an hardware-based representation. It corresponds to an abstract target architecture with an infinite number of registers. Also in the RTL representation are performed some optimization steps. Finally, the back end generates the assembly code for the target architecture by reading the RTL.

3.2 GCC plugin

Compiler plugins allow the developers to add new features to the compiler without needing to modify the compiler itself. As reported in the GCC plugin internal site page[7], the plugin's functionalities can be:

Plugin shortens the time needed to build and test new features. Only the code needed to implement the new functionality needs to be compiled.

Plugin allows the development and maintenance of compiler features that for one reason or another are not suitable for inclusion in the main GCC distribution.

It simplifies the job of developers who need to modify GCC but do not have the time or inclination to delve too much into the compiler internals.

Plugins can be useful to create an additional optimization pass, to transform code, or to analyze information. Plugin support is available in GCC since the version 4.5.0 where it can be built only in C. Since the version 4.7.0, it is possible to build plugins both in C or in C++; and since the version 4.8.0, plugins are built only in C++.
In this project we have used the GCC version 4.7.2 and we have written the plugin in C++.

3.2.1 Standard plugin template

A standard plugin template is shown in Listing 3.1. It contains the main parts that each plugin should have.

Listing 3.1. General plugin scheme

```
#include "gcc-plugin.h"
#include <iostream>
int plugin_is_GPL_compatible;

void cb_plugin_finish(void *gcc_data, void *user_data){

}

int plugin_init (plugin_name_args* info,
   plugin_gcc_version*)
{
  int r (0);
  // Register callbacks.
  register_callback(info->base_name,
             PLUGIN_FINISH,
             &cb_plugin_finish,
             0);
  return r;
}
```

All the GCC plugins have to import the *gcc-plugin.h* header file in order to access at the plugin API. They have also to declare the variable *int plugin_is_GPL_compatible* to assert that it has been licensed under a GPL-compatible license. In fact, GCC policy forces plugins to be under GPL license and if we do not declare such variable, the compiler will emit a fatal error. Note that the source code generated by plugins can have any license.

Every plugin should export the function `plugin_init` and it is called right after the plugin is loaded. It is mainly used to register the callbacks required by the plugin, to parse any input arguments, or to do any other required initialization.

3.2.2 Callbacks

The callbacks can be seen as *hook points*, where we engage our source code into GCC's execution flow.

According to the plugin API [6], callback functions have the following prototype:

Listing 3.2. Callback prototype

```
/* The prototype for a plugin callback function.
   gcc_data  - event-specific data provided by GCC
   user_data - plugin-specific data provided by the
      plugin.
*/
typedef void (*plugin_callback_func)(void *gcc_data, void
    *user_data);
```

Callbacks receive different parameters depending on the event at which they have been associated. In particular, we will see that some callbacks receive an *abstract syntax tree* node through the *gcc_data* parameter.

There are several pre-determined events in which the callbacks can be invoked, some of them are shown in Listing 3.3:

Listing 3.3. Callback events [6].

```
/* To hook into pass manager.*/
```

```
PLUGIN_PASS_MANAGER_SETUP,
/* After finishing parsing a type.*/
PLUGIN_FINISH_TYPE,
/* After finishing parsing a declaration.*/
PLUGIN_FINISH_DECL,
/* Useful for summary processing. */
PLUGIN_FINISH_UNIT,
/*Allows to see low level AST in C and C++ frontends.*/
PLUGIN_PRE_GENERICIZE,
/* Called before GCC exits.  */
PLUGIN_FINISH,
/* Information about the plugin. */
PLUGIN_INFO,
/* Called at start of GCC Garbage Collection. */
PLUGIN_GGC_START,
/* Extend the GGC marking. */
PLUGIN_GGC_MARKING,
/* Called at end of GGC. */
PLUGIN_GGC_END,
/* Register an extra GGC root table. */
PLUGIN_REGISTER_GGC_ROOTS,
/* Register an extra GGC cache table. */
PLUGIN_REGISTER_GGC_CACHES
  ....
```

The complete list of events can be found in the plugin API page [6].

Events are associated to function callbacks by calling the *register_callback* function with the following arguments:

- `char *name`. It represents the plugin's name.
- int event. It specifies one of the event code contained in Listing 3.3.
- plugin_callback_func callback: The function that handles the event.
- void *user_data. It is a pointer to the plugin-specific data.

In particular, during the next chapter, there will be described in depth three particular events:

- `PLUGIN_FINISH_DECL`,
- `PLUGIN_PRE_GENERICIZE` and

- `PLUGIN_FINISH`.

They will allow us to detect the external functions to mock.

3.2.3 How to compile a plugin

Plugins are compiled through the GCC compiler. What we obtain, after the compilation, is a shared object (.so), also called loadable module.

Assuming that our plugin is contained into the *plugin.cxx* file, we can obtain its respective shared object file by executing the command in Listing 3.4.

Listing 3.4. How to compile a plugin

```
g++ -I‘g++ -print-file-name=plugin‘/include -fPIC
-shared plugin.cxx -o plugin.so
```

With the -I flag, we import the path in which the plugin header files are installed. We get the path directory with the command: `g++ -print-file-name=plugin`. Note that *plugin.cxx* is written in C++ and we have used the GCC version 4.7.2. If we use a version below 4.7.0, we must add the line *external “C”* both before including the plugin header *gcc-plugin.h* and also before all the other function definitions, as for example, the callbacks and the *plugin_init* function.

3.2.4 Compile a file with a loaded plugin

It is possible to compile a file (for example *hello_world.c*), with a loaded plugin, through the command shown in Listing 3.5.

Listing 3.5. Load a plugin into gcc.

```
gcc -fplugin=./plugin.so hello_world.c -o hello_world
```

Since we have supposed that our plugin works with C files, the command above uses *gcc* instead of *g++*. In case the plugin was built to work for c++ files, we can replace *gcc* with *g++*.

3.2.5 GCC MELT

As stated in the GCC MELT site page[5]:

> GCC MELT is a high-level domain specific language for extending or customizing the Gnu Compiler Collection. It is implemented as a GCC plugin (for GNU/Linux systems, with GCC 4.7, 4.8 or later) in a free software (under GPLv3+ license and FSF copyright). Coding in MELT a GCC extension is much easier than manually developing a GCC plugin

in C or C++ (these are efficient languages, but not well fit to easily work on compiler's internal representations).

In addition to be a tool to simplify the development of GCC extensions, GCC MELT may be considered as a stable interface between GCC and the new GCC extensions. Plugins can access at the GCC internal API, but this API may change with new GCC versions. Hence, GCC MELT allows at GCC plugin developers to create GCC extensions to work with all the GCC versions, without the necessity to modify the extension for a specific GCC version.

However for this thesis, we have chosen to investigate GCC plugins to have full access at the GCC internals, in particular at the GCC *abstract syntax tree* (AST); considering that changes at the GCC AST API are rares.

3.3 The Abstract Syntax Tree (AST)

The *abstract syntax tree* (AST) is a tree representation of the source code written in a programming language. The AST does not include certain elements, such as inessential punctuation, braces, semicolons, parentheses, etc. In addition to being a representation of the source code, the AST contains also extra information about the program, due to the consecutive stages of analysis by the compiler. For example, the AST maintains the position of an element in the source code, an information that can be used in case of an error in the code, to notify the user the error's location.

For the GCC C front end, the AST is a number of connected nodes of type `struct tree`. Tree node are extended, with additional fields, to represent different things as constants, variables, functions, etc. To avoid AST tree corruption, the access at its nodes is done by using macros.

All nodes have a field called *tree code*, to indicate what the node represents. The macro `TREE_CODE` is used to refer at the node's tree code and it return an integer. Tree codes are defined in files tree.def and c-common.def. By using the global variable `tree_code_name` (array of strings) we are able to read the three code names. Some example of tree code names are: `TYPE_DECL` (a type declaration), `VAR_DECL` (a variable declaration), `ARRAY_TYPE` (an array type) or `RECORD_TYPE` (a struct type).

During the plugin development, it has been useful to know how we can read the node's tree code. This is shown with the source code in Listing 3.6.

Listing 3.6. Print a node tree code.

```
tree node = ...;
int tree_code = TREE_CODE(node) ;
```

```
cout <<''The tree code is: ''<< tree_code_name[tc] <<
    endl;
```

The AST nodes can be generalized dividing them in two big class as declaration nodes and type nodes. We can distinguish them because declaration nodes have suffix _DECL as tree code names, whereas type node have suffix _TYPE.

Both declaration and type nodes have two fields in common. The first is a TREE_CHAIN pointer that can be used as a singly-linked list to other trees. The other field is TREE_TYPE. Many trees store the type of an expression or a declaration in this field.

3.3.1 Declaration nodes

Since this research project concerns to detect external functions, we will start with the node that represents function: the FUNCTION_DECL node. In Figure 3.1 follows an example of function node.

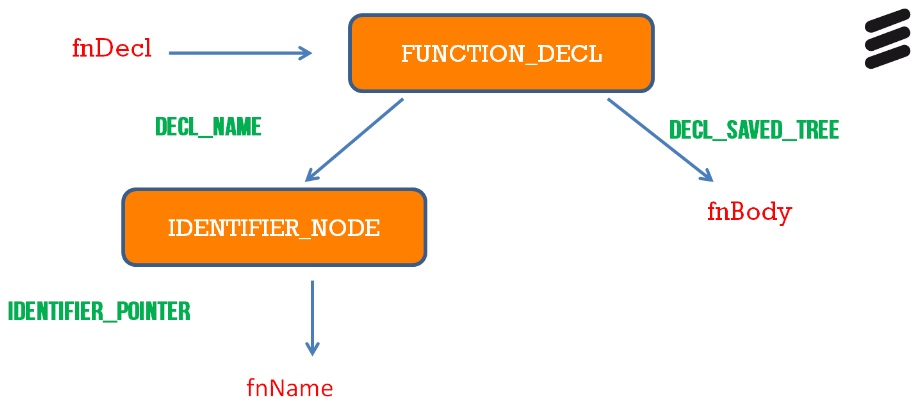

Figure 3.1. A representation of a function tree node.

By supposing to have a FUNCTION_DECL node, we can use the macro DECL_NAME to access at a node with IDENTIFIER_NODE as tree code, at which in turn, by using the macro IDENTIFIER_POINTER, we can read the function's name. In Listing 3.7 is shown how we can use these two last macros.

Listing 3.7. How to get the function's name.

```
tree* fnDecl = ... ; // a FUNCTION_DECL node
char* function_name = IDENTIFIER_POINTER(DECL_NAME(fnDecl
    ));
```

Whereas, to access at the instructions contained into the function's body, we use in turn, two macros: before DECL_SAVED_TREE and then BIND_EXPR_BODY as follows:

```
tree* function_body = BIND_EXPR_BODY( DECL_SAVED_TREE(
    fnDecl) );
```

It is common that functions have more than one instruction. If such situation is true, in the last example, the `function_body` tree node will be a `STATEMENT_LIST` node to gather nodes that represent the function's instructions.

In general, the `STATEMENT_LIST` node is another way to maintain a list of nodes. We can navigate such node through the API shown in Table 3.1.

Table 3.1. API statement node navigation.

Functions	Purpose
tsi_start(stmt_list)	get an iterator pointing at list head
tsi_end_p(iter)	is end of list?
tsi_stmt(iter)	get current element
tsi_next(&iter)	next element of the list

An example to navigate `STATEMENT_LIST` node is shown In Listing 3.8.

Listing 3.8. Statement node navigation.

```
// ...
tree_stmt_iterator stm_iterator = tsi_start(
    statement_list);
while(!tsi_end_p(stm_iterator)){
    tree statement = tsi_stmt(stm_iterator);
    debug_tree(statement);
    manage_tree_node(statement_node);
    tsi_next(&stm_iterator);
 }
```

In the example above, we have used the function `debug_tree()` to print the name, the tree code and all the fields of a node. It is provided with the plugin API and it has been often used during the project development for debugging reasons. `manage_tree_node()` is our function to manage each node obtained from the statement list navigation.

There are a wide variety of nodes to manage. For example we can have a `CALL_EXPR` node, if there is a function called. A `COND_EXPR` node for an "if" construct. A `GOTO_EXPR` node when there is a "for" construct. A `DECL_EXPR` node to represent a declaration of a variable like `int x = 4 + y`. It is a big task to manage all these different types.

In Listing 3.9 follows the example of how `manage_tree_node()` may be implemented. For each type of node, we call an ad-hoc function, that in turn it must use the proper macro depending at the node's type.

Listing 3.9. Example of manage_tree_node() function.

```
manage_tree_node(tree my_treenode){
  switch(TREE_CODE(my_treenode)){
    case CALL_EXPR:
      manage_call_expr(my_tree,i); //manage function
         calls
      break;

    case COND_EXPR:
      manage_cond_expr(my_tree,i);
      break;

    // ...
    default:
      break;
  }
}
```

3.3.2 Type nodes

The other class of nodes is the type node. GCC uses these nodes to represent types of variables. The type node class can be divided mainly in three categories: fundamental types, user-defined types and derived types.

Fundamental types are the following:

- `VOID_TYPE`,
- `BOOLEAN_TYPE`,
- `INTEGER_TYPE` and
- `REAL_TYPE`.

As it can be obvious that `VOID_TYPE` and `BOOLEAN_TYPE` represents respectively a void and a Boolean variable; it is less obvious that the *char* variables are represented with an `INTEGER_TYPE` tree code. The macro `TYPE_STRING_FLAG` returns 1 if the `INTEGER_TYPE` node represents a "char" variable. Otherwise we need to clarify which kind of integer: *short*, integer or long. This is done through the macro

`TYPE_PRECISION` that returns the respectively sizes: 16, 32, 64. These last values are platform dependent, in this case we have used an 64 bit linux platform. Also for the `REAL_TYPE` case, there is the `TYPE_PRECISION` macro to distinguish between the real type as *float*, *double* or *long double*.

The second category is the user-defined types. A node that belongs to this category represents a variable created ad hoc from the user. It can be an *enum*, an *union* or a *struct*. They are defined with the tree code names as follow:

- `RECORD_TYPE`,
- `UNION_TYPE` and
- `ENUMERAL_TYPE`.

The `REAL_TYPE` represents the definition of a *struct*. Let us suppose to have an user struct as follows in Listing 3.10.

Listing 3.10. The user struct *person*.

```
typedef struct person{
  char* firs_name;
  char* last_name;
  int   age;
};
```

By supposing that we have got the struct's node *user_struct_node*, we call *debug_tree(user_struct_node)* and it print in output the following result in Listing 3.11:

Listing 3.11. The *debug_tree()*'s output with the user struct *person* node in input

```
<record_type 0x40c05ae0 person sizes-gimplified
   asm_written type_0 BLK
    size <integer_cst 0x40b35b60 type <integer_type 0
       x40b47060 bitsizetype> constant 96>
    unit size <integer_cst 0x40b35b7c type <integer_type
       0x40b47000 sizetype> constant 12>
    align 32 symtab 1086343712 alias set -1 canonical
       type 0x40c05ae0
    fields <field_decl 0x40c080b8 firs_name
        type <pointer_type 0x40b4ed20 type <integer_type
           0x40b47240 char>
            sizes-gimplified asm_written unsigned SI
            size <integer_cst 0x40b3555c constant 32>
            unit size <integer_cst 0x40b35578 constant 4>
            align 32 symtab 1086171680 alias set -1
               canonical type 0x40b4ed20 context <
               translation_unit_decl 0x40be7ec4 D.1836>
            pointer_to_this <pointer_type 0x40bf1f00>
               reference_to_this <reference_type 0
               x40b523c0>>
        unsigned SI file system_under_test.h line 16 col
           9 size <integer_cst 0x40b3555c 32> unit size <
           integer_cst 0x40b35578 4>
        align 32 offset_align 128
        offset <integer_cst 0x40b35594 constant 0>
        bit offset <integer_cst 0x40b35620 constant 0>
           context <record_type 0x40c05ae0 person>
        chain <field_decl 0x40c08114 last_name type <
           pointer_type 0x40b4ed20>
            unsigned SI file system_under_test.h line 17
               col 9 size <integer_cst 0x40b3555c 32>
               unit size <integer_cst 0x40b35578 4>
            align 32 offset_align 128 offset <integer_cst
                0x40b35594 0> bit offset <integer_cst 0
               x40b3555c 32> context <record_type 0
               x40c05ae0 person> chain <field_decl 0
               x40c08170 age>>> context <
               translation_unit_decl 0x40be7ec4 D.1836>
    chain <type_decl 0x40be7f30 D.1820>>
```

As we can see, the `RECORD_TYPE` node, of the struct *person*, has a tree chain in *fields* to contain all the struct's fields. It maintains also the struct's dimension with the *size* field.

We are able to navigate and process the fields with a similar code shown in Listing 3.12:

Listing 3.12. How to navigate the struct *person* node

```
tree node_field_chain = TYPE_FIELDS(user_struct_node);
int number_param = 0;
 while(node_field_chain !=NULL ){
   tree field = TREE_TYPE(node_field_chain);
   //To print in output the field node call: debug_tree
```

```
         (field)
      // Process field node...
      node_field_chain = TREE_CHAIN(node_field_chain);
      number_param ++;
}
```

As the `RECORD_TYPE` nodes, also the `UNION_TYPE` and `ENUMERAL_TYPE` nodes are handling approximately in the same way. For example, to navigate an `UNION_TYPE` node instead to use the `TYPE_FIELDS` macro we adopt `TYPE_VALUES` macro.

The third category of types, the derived types, are contained:

- `POINTER_TYPE`,
- `REFERENCE_TYPE` and
- `ARRAY_TYPE`.

Chapter 4

Plugin development

The chapter explains how we can use a GCC plugin to intercept calls to external functions in a C file. The plugin detects the external functions by reading the AST tree nodes. AST tree nodes are generated when GCC parses a file.

Afterwards, the plugin writes all the new mock functions in a temporary file. Since the temporary file is part of the runtime system it will be described in the next chapter.

4.1 Intercept external functions

It is possible to intercept external functions through a GCC plugin. We can access the GCC intermediate representation, the *abstract syntax tree* (AST) through particular plugin events. The plugin's events used to achieve this purpose are: *PLUGIN_FINISH_DECL*, *PLUGIN_PRE_GENERICIZE* and *PLUGIN_FINISH*.

The first event, `PLUGIN_FINISH_DECL` makes possible to access at AST nodes that represent all the declarations in a C file. It is raised each time the parser reads a *declaration*. The second event instead, `PLUGIN_PRE_GENERICIZE` is raised up each time the parser reads a *function definition*, or in another worlds, a function with a body. Finally, the last event, *PLUGIN_FINISH_DECL* is used to write in a temporary file the new mock functions.

Let us clarify the difference between declaration and definition, with an example. We suppose to have a function called *foo*. The function declaration of *foo* is:

```
int foo();
```

whereas the function definition is:

```
int foo(){
  // function body
}
```

It means that a function declaration does not have a body. It simply tells to the compiler that it can use the function and expect that it will be defined somewhere with a function definition.

Hence, each time the parser reads a declaration, the event `PLUGIN_FINISH_DECL` will be raised and its callback performed. The callback will receive as parameter the AST node of class declaration. It means that the declaration node can be a node that represents a function or something else. On the contrary, the callback associated to the second event will receive as argument only function definitions. It is raised up each time the parser reads a function definition.

We will start to explain how we have built the plugin that intercept external functions, step by step. Listing 4.1 shows the *plugin-mock.c* file. The source code is a starting point in which we will add further functions during this chapter. As usual, the plugin must import the plugin-header file, export the *plugin_init* function and finally set the callbacks to manage these three events. Note also that we have developed the plugin in C++.

Listing 4.1. plugin-mock.cxx.

```
#include "gcc-plugin.h"

#include "tree.h"
#include "tree-iterator.h"

using namespace std;
int plugin_is_GPL_compatible;

void cb_finish_decl(void *gcc_data, void *plugin_data)
{
  //Event's callback: PLUGIN_FINISH_DECL
}

void cb_plugin_pre_generize (void *gcc_data, void *
   plugin_data)
{
   //Event's callback: PLUGIN_PRE_GENERICIZE
}

void cb_plugin_finish(void *gcc_data, void *plugin_data)
{
   //Event's callback: PLUGIN_FINISH
}

int plugin_init (plugin_name_args* info,
                 plugin_gcc_version*)
{
  int r (0);
  // Register callbacks.
  register_callback(info->base_name,
                    PLUGIN_FINISH_DECL,
                    &cb_finish_decl,
                    0);

  register_callback(info->base_name,
                    PLUGIN_PRE_GENERICIZE,
                    &cb_plugin_pre_generize,
                    0);

  register_callback(info->base_name,
                    PLUGIN_FINISH,
                    &cb_plugin_finish,
                    0);

  return r;
}
```

In the source code above, there are three different callbacks. One for each event used.

We can compile such plugin with the command, as seen in Chapter 3:

Listing 4.2. Command to compile the plugin: *plugin-mock.cxx*.

```
g++ -I‘g++ -print-file-name=plugin‘/include -fPIC
-shared plugin-mock.cxx -o plugin-mock.so
```

To explain how the plugin can intercept the external function, we provide this example of system under test (SUT) and we call its file: *system_under_test.c*. The SUT's source code is shown in 4.3.

Listing 4.3. system_under_test.c

```
#include <stdio.h>

double baz(int y);
double foo(int  y, int x);
int function_to_test();
int helper_func();

int helper_func(){
  int x = 1;
  //...
  return x;
}
int function_to_test()
{
  double result_baz =  baz(5);
  double result_foo =  foo(4, 7);
  printf("Total: %d", result_baz + result_foo +
     helper_func() );
  return 1;
}
```

To read the SUT file with a loaded plugin we need to execute the following command:

Listing 4.4. Command to compile the SUT with the plugin-mock loaded

```
gcc -c -fplugin=./plugin-mock.so system_under_test.c -o
   system_under_test.o
```

Note that this time we add the -c flag to avoid the GCC linking phase. By executing the command above, the event `PLUGIN_FINISH_DECL` will call several times. According at the callback registration, each time that the parser finds a declaration, it will be raised up the function:

Listing 4.5. Registered callback with the PLUGIN_FINISH_DECL event

```
void cb_finish_decl(void *gcc_data, void *){
  //Event's callback: PLUGIN_FINISH_DECL
}
```

In which, it will receive as `gcc_data` parameter, the AST tree node that represents the declaration just parsed.

Whereas, the `PLUGIN_PRE_GENERICIZE` event is raised up every time that the parser reads a function definition. For example, by continuing to refer at the SUT shown in Listing 4.3, the event will be hooked to the following callback 4.6 twice:

Listing 4.6. Callback for the PLUGIN_PRE_GENERICIZE event.

```
void cb_plugin_pre_generize (void *gcc_data, void*)
{
   //Event's callback: PLUGIN_PRE_GENERICIZE
}
```

The first time, when the GCC parser reads the function *helper_func()* and the second time when it reads *function_under_test()*.

Finally, the event `PLUGIN_FINISH` is raised up always only once, at the end of the GCC compilation, and it calls the callback function `cb_plugin_finish`.

4.1.1 Callback cb_finish_decl

As we stated previously, the `PLUGIN_FINISH_DECL` event allows us to access each node of class declaration present in the file in which the parser is reading. When the event `PLUGIN_FINISH_DECL` occurs, GCC calls the function in the plugin *cb_finish_decl*. In such function we filter the AST declaration node received, we save the node in a list if it is a function node. It means, we save only nodes with node's tree code equal to FUNCTION_DECL. The list is declared as follows:

```
list<tree_node*> list_functions_declarations;
```

There is another aspect to consider. In this way, we get also the function's declaration node from any headers inclusion of the standard library; as for example, the *printf* function. And we would like to avoid to mock also them.

We avoid functions declaration node from the standard library by checking the node's origin with the macro `DECL_SOURCE_FILE`. The macro returns in a string the source code's path in which the function is defined. Hence, we do not consider all the function nodes declared in the *"/usr/include"* path.

It follows the source code of the callback in Listing 4.7.

Listing 4.7. How to get all the functions node.

```
void cb_finish_decl(void *gcc_data, void *user_data){
  //cout << "*** EVENT PLUGIN_DECLARATION ***" << endl;
  tree_node *ast_node = reinterpret_cast<tree_node*>(
     gcc_data);
  int treecode = TREE_CODE(ast_node);
  print_node_declaration(ast_node);
  if(treecode == FUNCTION_DECL){// we are interested only
      to node with tree node == FUNCTION_DECL
    string source_file = DECL_SOURCE_FILE(ast_node);
    size_t found = source_file.find("/usr/include");
    if(found==string::npos){// we are NOT interested to
       mock library functions.
      print_node_declaration(ast_node);
      list_functions_declarations.push_back(ast_node);
    }
  }
}
```

The function *print_node_declaration* prints in output each function node and its source code is shown in Listing 4.8.

Listing 4.8. How to get all the functions node.

```
void print_node_declaration(tree_node* node_declaration){
 int treecode = TREE_CODE(node_declaration);
 tree id = DECL_NAME(node_declaration);
 string source_file = DECL_SOURCE_FILE(node_declaration);
 int number_line =  DECL_SOURCE_LINE(node_declaration);
 const char* name (id ? IDENTIFIER_POINTER (id): "<
    unnamed>");
 cout << "*** ~~~ " << tree_code_name[treecode] << " name
     " << name << "** in "<< source_file << " number line
    : " << number_line << endl;
}
```

If we try to compile the SUT with the command in Listing 4.4, it will be printed the output shown in Listing 4.9.

Listing 4.9. Output printed when the SUT is compiled with the plugin.

```
*** Tree node:function_decl Function name: baz in
   system_under_test.c  number line: 3 ***

*** Tree node:function_decl Function name: foo in
   system_under_test.c  number line: 4 ***

*** Tree node:function_decl Function name:
   function_to_test in system_under_test.c  number line:
   5 ***

*** Tree node:function_decl Function name: helper_func in
    system_under_test.c  number line: 6 ***
```

By analyzing the output above, we can see that we got also function's nodes that we are not interested to mock. In this particular case, the functions *function_to_test* and *helper_func*. Since, we aim to automatically mock only external functions, we need to detect which are the functions that should not be mocked, that are the SUT's function definitions.

4.1.2 Callback cb_plugin_pre_generize

The event PLUGIN_PRE_GENERICIZE is raised when the parser reads a function definition. Hence, we use each *cb_plugin_pre_generize* callback to process its function definition AST node, received as parameter. We save them in another list defined as follow:

```
list<tree_node*> list_functions_definitions;
```

The callback's source code is in Listing 4.10.

Listing 4.10. Saving all the function definition nodes.

```
void cb_plugin_pre_generize (void *maintree, void*)
{
  cout << "** PLUGIN_PRE_GENERIZE EVENT **"<< endl;
  tree_node *tn = reinterpret_cast<tree_node*>(maintree);
  cout << " --- Function name: " << IDENTIFIER_POINTER(
     DECL_NAME(tn)) << " ---" << endl;
  list_functions_definitions.push_back(tn);
  cout << "** Ends PLUGIN_PRE_GENERIZE EVENT **" << endl;
}
```

By compiling our SUT file with this new modified callback, we will save into the list the nodes that represent *function_to_test* and of *helper_func*.

4.1.3 Callback cb_plugin_finish

Finally, now that we have got both declaration and definition functions nodes, we can use them to build a new list that contains only the external functions to be mocked. By continuing to use the SUT in Listing 4.3, the external functions are *foo* and *baz*.

Hence, we declare a third list in *plugin-mock.cxx* with the following declaration:

```
list<tree_node*> list_functions_to_mock;
```

And then, we insert in that list only the node that are presented in *list_functions_declarations* but not in *list_functions_definitions*.

It follows in Listing 4.11, the source code of the third callback and an helper function called *find_functions_to_mock*.

Listing 4.11. Saving into a list only the external functions to mock.

```
void find_functions_to_mock()
{
   for (list<tree_node*>::iterator it =
      list_functions_declarations.begin(); it !=
      list_functions_declarations.end(); ++it)
   {
    tree id = DECL_NAME(*it);
    string name_function__declaration (IDENTIFIER_POINTER
        (id));
    bool isPresent = false;
    for (list<tree_node*>::iterator it2 =
       list_functions_definitions.begin(); (!isPresent)
       && it2 != list_functions_definitions.end(); ++it2)
       {
      tree id2 = DECL_NAME(*it2);
      string name_function_definition(IDENTIFIER_POINTER
         (id2));
      if(name_function_definition.compare(
name_function_declaration) == 0)
      {
      isPresent = true;
      }
    }
    if(!isPresent)
    {
      list_functions_to_mock.push_back(*it);
    }
  }
}
void cb_plugin_finish(void *gcc_data, void *user_data)
{
  cout << "****** EVENT PLUGIN_FINISH *******" << endl;
  find_functions_to_mock();

  //Dynamic generation of the run time system's first
```

```
       part.
   write_function_on_file();
   write_header();
   cout << " *** END PLUGIN **** " << endl;
}
```

The *write_function_on_file* and *write_header* functions are part of the run-time system; and for this reason, we will see them in the next chapter.

Chapter 5

The Runtime System

The runtime system is divided in two parts. One part is SUT-dependent and it is automatically generated when the plugin reads the SUT; it is contained in the *temporary_file.cpp* file. The other one is SUT independent and it is contained in *helper.cpp*.

The reader may have noted that the runtime system is actually developed in C++, but we are investigating the possibility to create a runtime system in pure C. We have decided to use C++ because it is faster during the plugin development; by keeping in mind that it will not require long time to rewrite all the project in pure C.

In the last chapter, we have seen how we can get the AST nodes of the SUT external functions; and in this chapter, we will see how we can use them to write the new mock functions and their respective expectation functions, into a temporary file.

During the current chapter, we will show the API implemented, we will explain how the runtime system is built, and finally we will redirect the SUT external functions to the new mock functions.

5.1 Framework API

In this section, we show the framework API. During the study of this project, we have realized that we could develop an API as similar as possible at the Google Mock API. In this way, the people that already know GMock may use easily also this new framework.

The API allows the user to set the expectations, or in other words, to set the behavior of the external functions (how will they be called? what will they do?). We can set the external function behavior in a C file (as we pointed out during the Introduction chapter with the file *test_sut.c*) by including two headers:

temporary_file.h and *myframework.h.* The first header, *temporary_file.h* is and generated via plugin. It contains all the respective declaration of the expectation functions. The second header contains the framework API.

The general expectation function prototype follows in 5.1.

Listing 5.1. Expectation function prototype.

```
EXPECT_functionname(Matchers,
                    Cardinalities,
                    Actions
                   );
```

5.1.1 Expectation Order

As happen in Google Mock, also in our project, by default the expectations can be matched in any order.

5.1.2 Matchers: What Arguments Do We Expect?

A *matcher* matches a single argument. An argument is the parameter received when the mock function is called. Matchers allow us to set the proper parameters that an external function should receive during a test. If, when running the test, the mock function will receive the wrong arguments, the system will output an error message.

Note: we must set a matcher or a wildcard, for each parameter of the mock function. The API *Matcher* are shown in Table 5.1.

Table 5.1. Matcher - Generic Comparison

VALUE(x)	argument == x
Gt(x)	argument > x
Lt(x)	argument < x
Ge(x)	argument >= x
Le(x)	argument <= x
Ne(x)	argument != x
STRING(x)	argument is a string (char*) equal to x
USER_FUNCTION(*func*)	*func* is an user defined function. It returns 0 if the test is positive. The argument is checked with the function *func*.

An interesting macro is `USER_FUNCTION`. It allows us to set an our function to check the arguments. This is really useful, especially when we need to check variables defined from the user, such as structures or enumeration.

For example, if we want to mock an external function with the following declaration:

Listing 5.2. Example - External function to mock.

```
int foo(int x);
```

We may use an user function to check if its argument is within an interval. In Listing 5.3 is shown an example of expectation setting with an user defined function.

Listing 5.3. Example - How to use the user-function macro.

```
int my_user_function(int x){
  if(x > 10 && x < 20){
    return 0;   // the argument is acceptable
  }
  else return 1;// the runtime system will print an error
      message.
}
void test_sut(){
  EXPECT_foo(USER_FUNCTION(&my_user_function),
        TIMES(1),
        WILL_ONCE(RETURN(7))
        );
  function_to_test();
}
```

A special kind of matcher is the wildcard. A wildcard is useful when we are not interested to check arguments; we want to define that each possible value of an argument is acceptable. The wildcard macro is shown in Table 5.2.

Table 5.2. Matcher - Wildcard

ANY_VALUE	*Wildcard*: argument can be any value of the correct type

5.1.3 Cardinalities: How Many Times Will It Be Called?

Cardinalities specify how many times a mock function will be called. Cardinalities macros are shown in Table 5.3.

Note: we must set the cardinalities only once.

Table 5.3. Times.

TIMES(x)	the mock function should be called exactly x times
TIMES_AT_LEAST(x)	the mock function should be called at least x times
TIMES_AT_MOST(x)	the mock function should be called at most x times

5.1.4 Actions: What Should It Do?

Actions tell what the mock function should return.

Table 5.4. Return Values.

WILL_ONCE(RETURN(x))	the return value is x
WILL_REPEATEDLY(RETURN(x))	it will repeatedly return x

The macro `RETURN` is a separated macro to keeps the *actions* more readable at the user.

5.1.5 Multiple Expectations

We provide also the features to set multiple expectations. It means we can set more than one expectation for the same external function. As in GMock[8], our system will search the expectation in the reverse order in which the expectations are defined, and stop when an active expectation that matches the arguments is found: newer rules override older ones.

Referring us at the external function example in Listing 5.2, we could suppose to have two different expectations for the same external function as shown in Listing 5.4.

Listing 5.4. Example - Multiple expectations.

```
EXPECT_foo( VALUE(4),
            TIMES(1),
            WILL_ONCE(RETURN(3))
          );  // #1
EXPECT_foo( VALUE(5),
            TIMES(5),
            WILL_ONCE(RETURN(5)),
            WILL_REPEATEDLY(RETURN(3))
          );  // #2
```

If `foo(5)` is called six times, the sixth time it will be an error, as the last matching expectation #2 has been saturated. If, however the sixth time `foo(5)` is replaced by `foo(4)`, then it would be OK, as now #1 will be the matching expectation. In the Google Mock site [4], we can see how it sets multiple expectations.

If the mock function called has a matching with the expectation #1, it will return the value three. In the other case, when the function matches the expectation #2, it will return, the first time the value five; and then always three for the other four times.

Note: Why do we search for a match in the reverse order of the expectations? As it happens in Google Mock[8], if we have two expectations on the same external function, we want to put the one with more specific matchers after the other, or the more specific rule would be shadowed by the more general one that comes after it.

5.2 Writing the temporary file

In Section 4.1.3, we use the event `PLUGIN_FINISH` to find the external functions; and we save them in the respective list called *list_function_to_mock*. In addition such event is used also to write into the temporary file.

The temporary file is the SUT-dependent part of the runtime system; it is contained into the file *temporary_file.cpp* and *temporary_file.h*. The runtime system SUT-independent is contained in the files *helper.cpp* and *helper.cpp*.

In Listing 4.11, the function *write_function_on_file* writes into the *temporary_file.cpp*; whereas the function *write_header* writes the *temporary_file.h* file header.

We write both files by using the C++ `fstream` library.

The plugin generates the *temporary_file* following these operations:

- The temporary file is connected with the SUT independent runtime system part: the header *helper.h* is included into the *temporary_file*.
- The plugin generated all the necessary structures to keep the user expectations in memory at runtime,
- it writes the new expectation functions,
- and the respective new mock functions.
- Finally, it generates a function for a global check at the and of the test.

A partial plugin source code to write into the *temporary_file* is shown in Listing 5.5.

Listing 5.5. *plugin-mocking.cpp* - The plugin writes into the temporary file.

```
void write_function_on_file(){
  ofstream myfile;
  myfile.open("temporary_file.cpp");
  myfile << "#include \"helper.h\" "<< endl;
  ...
  // Section: Saving the expectation's settings
  write_structs();
  //Section: Expectation functions
  write_expectation_functions();
  //Section: New mock functions
  write_external_functions();
  //Section:
  write_check_testCompleted_function();
  myfile.close();
}
```

Each function called in Listing 5.5 will be described in depth in the next sections.

5.3 Saving the expectations

In this section, we describe the data structures which keep the expectation settings of the user. These structures will be read when the mock functions are called, allowing to check at runtime, if the external functions follow the correct behavior.

The plugin generates an ad-hoc structure for each SUT external function node of the `list_functions_to_mock` list. When we iterate this list, we get the function nodes with tree code equal at *FUNCT_DECL* type. We use these nodes to access at information such as: the function name, the function arguments, the function return type and so on.

The generated structure names have a prefix equal to *Mock_* followed by the name of the external function. For example, if we refer to the external function in Listing 5.2, the structure will be *Mock_foo*. In these structures all the values given by the user in the expectation settings are saved.

Continuing with our example, we may have an expectation as the following:

Listing 5.6. Example of *foo* expectation.

```
EXPECT_foo(VALUE(5),
        TIMES(1),
        WILL_ONCE(RETURN(7))
        );
```

In this case, we need to save the value *5*. To keep such value in memory at runtime, we should need a simple structure similar as that shown in Listing 5.7.

Listing 5.7. *temporary_file.cpp* - A simple possible struct to maintain in memory the argument of the function *foo*.

```
struct Mock_foo{
  int param0;
};
```

During the test execution, the respective new mock function *foo(int x)* will read the structure *Mock_foo*, to compare the received arguments with the right parameter expected (in this particular example the argument *5*).

5.3.1 Wildcard Implementation

Since we provide the wildcard feature, we needed a way to distinguish when, by setting the expectation, we do not want to check the value of an argument during the the mock function execution. Unfortunately, the last struct, in Listing 5.7, is not enough to manage such feature.

In C, it has not been easy to implement the wildcard. After some research, we came up with the following solution: we create into the respective structure *Mock_*, two variables for each parameter of the external function. Then we use macros to hide these two variables.

Let us clarify this last concept with an example. Suppose we have an external function declaration as in Listing 5.2; consequently the plugin could generate in the *temporary_file.cpp* file the structure shown in Listing 5.8.

Listing 5.8. *temporary_file.cpp* - Wildcard struct concept.

```
struct Mock_foo{
  int switch0;  int param0;
};
```

In the structure above, the integer variable *switch0* will allow the respective mock function to check if its argument must be an exactly value or it can be everything, i.e. a wildcard.

Managing the arguments in such way involves, in the expectation function definitions, to have the double number of the parameter as shown in Listing 5.9.

Listing 5.9. Expectation with double parameters

```
extern "C"
void expect_foo(int switch0, int param0);
```

To keep the expectation settings user-friendly, we *hide* that two parameters by defining new macros into the file *myframework.h*, as shown in Listing 5.10.

Listing 5.10. *framework.h* - How we may implement the wildcard feature.

```
#define VALUE(x)          0, x
#define ANY_VALUE         1, 0
```

Each macro has an unique *identification number* that distinguishes one macro from another one. In the macro definitions above, the number zero indicates that the macro used is *VALUE* and one for the wildcard *ANY_VALUE*.

Note: the value *0* in *ANY_VALUE* is acceptable also when we are using the wildcard for string (char *) because it is converted to the *NULL* value.

Using this kind of trick, we have implemented a working wildcard. In addition, we do not have to create a wildcard for each different type of variable as an integer variable, a double or pointer and so on. We have one wildcard that can be used for all different types of variable.

5.3.2 User-function macro Implementation

As we mentioned earlier, the *USER_FUNCTION* macro is useful when we want to check an argument in a specific way. For example, we can check if the arguments are within a proper interval, or if the fields of an user-struct argument contain the right values. For this feature, we have added a third parameter at the *Mock_* struct: an user function pointer. Now, the expectation function declaration will have three parameters for each correspondent parameter of the respective SUT external function.

For example, if the plugin detects an external function as *foo(int x, double y)*, it may write a *Mock_foo* structure into the temporary file as shown in Listing 5.11.

Listing 5.11. *temporary_file.cpp* - How the *Mock_foo* structure may be implemented.

```
struct Mock_foo{
 int switch0;  int param0; int (*userfunc0)(int x);
 int switch1;  double param1; int (*userfunc1)(double x);
};
```

Note: the return type of the user function is always an integer, whereas the user function's parameter is dependent to the respective type of variable of the external function parameter.

The new macro definitions are shown in Listing 5.12. We have added the *NULL* value at the third parameter, when we do not need an user function pointer.

Listing 5.12. *framework.h* - Adding the user-function macro.

```
#define VALUE(x)          0, x, NULL
#define ANY_VALUE         1, 0, NULL
//USER_FUNCTION(x) x: (*function)(``argument type'' x)
   The function return 0 if the test is positive.
   Otherwise the test will fail.
#define USER_FUNCTION(x) 3, NULL, x
```

5.3.3 Matchers Implementation

For completeness, all the *Matcher* macros definition are shown in Listing 5.13.

Listing 5.13. *framework.h* - All the *Matcher* macros definitions.

```
#define VALUE(x)          0, x, NULL
#define ANY_VALUE         1, 0, NULL

// STRING(x) manage char* types
#define STRING(x)         2, x, NULL

//USER_FUNCTION(x) x: (*function)(``argument type'' x)
   The function return 0 if the test is positive.
   Otherwise the test will fail.
#define USER_FUNCTION(x) 3, NULL, x

//y is the parameter got from the mocked function call
#define Gt(x)             4, x, NULL    //  x > y
#define Lt(x)             5, x, NULL    //  x < y
#define Ge(x)             6, x, NULL    //  x >= y
#define Le(x)             7, x, NULL    //  x <= y
#define Ne(x)             8, x, NULL    //  x != y
```

5.3.4 Cardinalities and Actions

To keep the information about the cardinalities and the actions, we have built an ad-hoc structure. It is defined into the independent part of the runtime system, into the header *helper.h*. In this way, it does not need to be generated by the plugin.

Such structure is called *Expectation*, and it is shown in Listing 5.14.

Listing 5.14. *helper.h* - Expectation struct.

```
struct Expectation {
  int type_expectation; //It can be: TIMES,
     TIMES_AT_LEAST, TIMES_AT_MOST, WILLONCE or
     REPEATEDLY
  void* returnvalue;   //for TIMES/TIMES_AT_LEAST/
     TIMES_AT_MOST, it contains the number of times that
     the function should be called. For WILL_ONCE and
     WILL_REPEATEDLY contains the return value.
  int number_calls ; // TIMES/TIMES_AT_LEAST/
     TIMES_AT_MOST uses to keep track of the number of
     times that we call the correspondent function.
  Expectation(){
    number_calls = 0;
  }
};
```

Every time that the tester uses the macros to set the cardinalities in Table 5.3, or to set the return value in Table 5.4, the runtime system has to save in the *Expectation* structures all the values necessary. This operation is done during the expectation function's execution. In the next Section, we will clarify in detail, how the expectation functions use these structures. Moreover, since we do not know before how many macros will be used by the user, we need to maintain the *Expectation* structures in a list. Such list is defined in each *Mock_* structure.

Figure 5.1 shows a generic representation of the memory based on the previously structures.

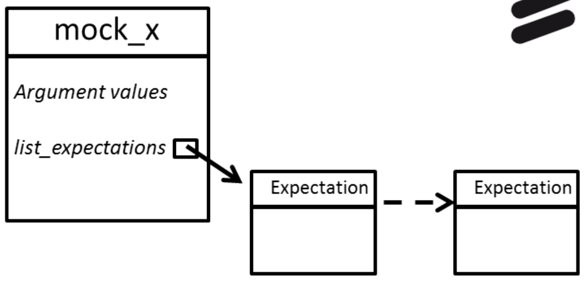

Figure 5.1. Representation of the data structure of a generic SUT external function *x*.

For example, with an external function *foo(int x, double y)*, finally the plugin will generate the following new respective *Mock_foo* structure:

Listing 5.15. *helper.h* - Expectation struct.

```
struct Mock_foo{
 int switch0;  int param0; int (*userfunc0)(int x);
 int switch1;  double param1; int (*userfunc1)(double x);
 list<Expectation*> list_expectations;
};
```

And by supposing to have a *foo* function behavior as follows:

```
EXPECT_foo(
 VALUE(5),
 Lt(8),
 TIMES(3),
 WILL_ONCE( RETURN(7) ),
 WILL_REPEATEDLY( RETURN(5) )
);
```

The runtime system will create three different *Expectation* structures to save respectively the values given by *TIMES*, *WILL_ONCE* and *WILL_REPEATEDLY* macros. Finally, these structures are linked into the *list_expectation* field of the respective *Mock_foo* structure.

Cardinalities and action macros have a specific unique integer as the matcher macros. In Listing 5.16 is reported the rest of the *framework.h* source code containing the *Actions* and *Cardinalities* definitions.

Listing 5.16. *framework.h* - Cardinalities and Actions definitions

```
#define TIMES(x)            200, x
#define WILL_ONCE(x)        201, x
#define WILL_REPEATEDLY(x)  202, x
#define RETURN(x)                x
#define TIMES_AT_LEAST(x)   203, x
#define TIMES_AT_MOST(x)    204, x
```

5.3.5 Multiple Expectations

Since we provide the multiple expectations feature, we need to keep a list of different expectations for the same external function, as it happens in the example shown in Listing 5.4.

To manage that situations, the plugin creates as well as *Mock_* structure, a list of pointers to such structures.

Referring us to the the external function *foo(int x, double y)*, the plugin will generate the source code shown in Listing 5.17.

Listing 5.17. *temporary_file.cpp* - Source code generated to save the *foo* expectation settings.

```
struct Mock_foo{
 int switch0;  int param0; int (*userfunc0)(int x);
 int switch1;  double param1; int (*userfunc1)(double x);
 list<Expectation*> list_expectations;
};
list<Mock_foo*> list_foo;
```

The above structure is finally the example of what the plugin generates for each SUT external function.

In Figure 5.2 follows a representation of the previous structures in memory, supposing to have a SUT external function *x*.

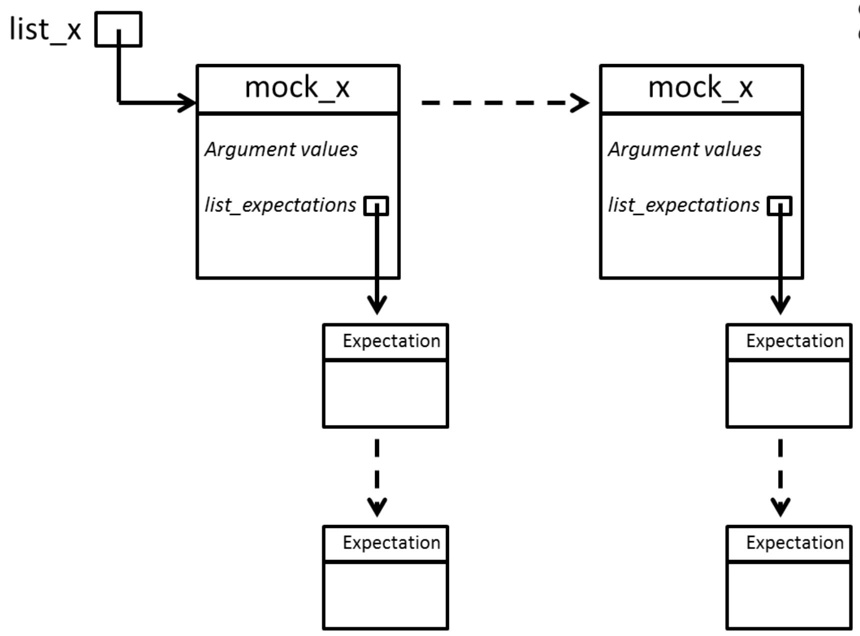

Figure 5.2. Representation of the data structure of a generic SUT external function *x*.

5.3.6 The write_struct function

Summarizing, for each external function *x* node contained into *list_functions_to_mock*, the plugin writes into the *temporary_file.cpp*, the respective *Mock_x* structure and the list *list_x* to save these latter. The *Mock_x* structure will contain three variables for each parameter of the respective external function and a list of pointers to the *Expectation* structures. All these operations are performed by the function *write_structs()* contained in the file *plugin.cpp*. For each function node, we need to get the respective function's name and the function's parameters. As we have pointed out during the Chapter 3, we can access at the function's name by using the AST macros as follows:

```
for (list<tree_node*>::iterator it =
    list_functions_to_mock.begin(); it !=
    list_functions_to_mock.end(); ++it){
 ...
 string function_name =  IDENTIFIER_POINTER(DECL_NAME(*it
     ));
```

And we can access and navigate the function's parameters with the following macros:

```
tree node = TYPE_ARG_TYPES(TREE_TYPE(*it));//get function
    's arguments
 if(node != NULL) //if node is null the function does NOT
      have arguments
 {
    do{
      list_node_args.push_back( TREE_VALUE(node) );
    }while(node = TREE_CHAIN(node) );
...
```

We save into the list *list_node_args*, the node parameters got via the macro `TREE_VALUE(node)`. Each node parameter must be elaborated with the proper macro depending on the type of the node. We iterate the list, to write the *Mock_* structure for each respective function node.

For a more depth details, it is suggested to look at the source code in the Appendix 2.

5.4 Expectation functions

Now that we have defined the structures to keep in memory the behavior of the mock functions, we need to gather the expectation settings and save them in these structures. This task is done trough the expectation functions.

Again, the plugin writes into the temporary file an expectation function for each SUT external function found. These expectation functions save in the respective proper structures, all the user information related at the mock functions' behavior. The plugin writes the expectation functions when the function *write_expectation_functions* is called, Listing 5.5.

In accordance with the API macros definition in Listing 5.13 and Listing 5.16, we must write the expectation functions with at least, the triple number of parameters respect at the number of its external functions

Let us clarify with an example. If we suppose to have a SUT with a simple external function as *int foo(int x)*; and its behavior defined by the user with following expectation:

Listing 5.18. *foo* mock function behavior.

```
EXPECT_foo(
  VALUE(5),
  TIMES(3),
  WILL_ONCE(RETURN(4)),
  WILL_REPEATEDLY(RETURN(0))
);
```

The respective *foo* expectation function must have at least 3 parameters for each parameter of its external function.

Note that, we cannot know early how many *Cardinalities* or *Actions* macros will be used from the user. Hence, we must have an expectation functions *expect_foo*, with an indefinite number of parameters. To manage an indefinite number of arguments in C, we have used the *stdarg.h* header file of the C standard library. We import this header at the top of the *temporary_file.cpp* as follows:

```
extern "C" {
  #include <stdarg.h>
}
```

In which, we have used the *extern* key word to indicate that we are using a C library into a C++ file.

We have developed the plugin to write the *foo* expectation function as shown in Listing 5.20.

Listing 5.19. *temporary_file.coo* - *foo* Expectation function.

```
void expect_foo(int switch0,  int param0, int (*userfunc0
   )(int x), int startvararg, ...){
  ..
}
```

When we use the *stdarg* library, we must define an our *final parameter* to be inserted at the last macro used by the user, during the expectation settings. However, we cannot know early how many macros the tester will use.

To solve such problem, the plugin create an header file called *temporary_file.h*. In this file, we have developed variadic macros to set automatically a *final parameter* at the end of each expectation function.

Continuing to refer us at the last example, the *temporary_file.h* generated will be as the following:

Listing 5.20. *temporary_file.coo* - *foo* Expectation function

```
#ifndef TEMPORARYFILE_H
#define TEMPORARYFILE_H

#define EXPECT_foo(...) expect_foo( __VA_ARGS__,
    NO_MORE_PARAMETERS)
void expect_foo(int switch0,  int param0, int (*userfunc0
    )(int x), int startvararg, ...);

#endif
```

In which, we have defined the macro *NO_MORE_PARAMETERS* is the *final parameter* set at the end of each expectation. The *NO_MORE_PARAMETERS* macro is defined in *framework.h* as follows:

Listing 5.21. *framework.h* - Definition of the *Final parameter.*

```
...
#define NO_MORE_PARAMETERS -256
```

The plugin writes the *temporary_file.h* when the function *write_header* is called. It iterates the function node of the *list_functions_to_mock* list to access at the information of an external functions information, by using, as we have already seen, the AST macros.

In general, when the user calls an expectation function of a respective external function x; the expectation function should be able to perform the following steps:

1. It dynamically allocates the respective *Mock_x* structure,

2. it saves the user expectation settings into *Mock_x*,

3. and finally, it inserts the *Mock_x* structures into the respective list, *list_x*.

During Step 2, the expectation function must save the *Actions* and *Cardinalities* user settings, by allocating the structure *Expectation*. The plugin inserts the *Expectation* structures allocated into *list_expectation*, with the following policy: *last Expectation created is inserted at the end of the list.*

Figure 5.3 shows the memory presentation of the runtime system memory, when the function *expect_foo* is called in Listing 5.18.

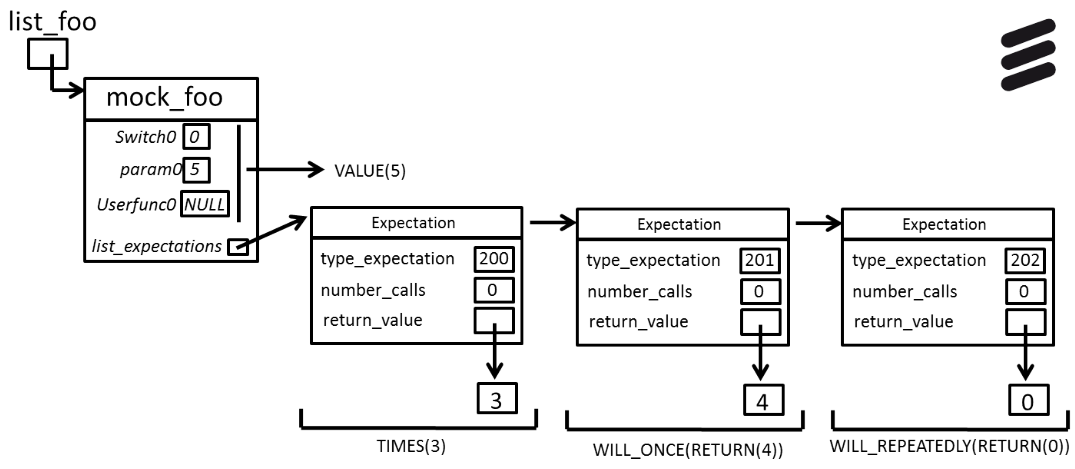

Figure 5.3. Memory representation at the end of the *expect_foo* execution.

We remember that the *return_value* field is a void pointer; and when a new value must be inserted in the *Expectation* structure, we allocate a new piece of memory to save the value received from the macro and then, we save the address of the new allocated memory in *return_value*.

Listing 5.22 shows the source code of the respective expectation function of the SUT external function *int foo(int x)*.

Listing 5.22. *temporary_file.cpp* - The *expect_foo* function generated by the plugin.

```
extern "C"
void expect_foo(int switch0, int param0, int (*userfunc0)
   (int x), int startvararg, ...){
  struct Mock_foo *obj = new Mock_foo();
  list_warehouse_has_inventory.push_front(obj);
  obj->switch0= switch0;
  obj->param0= param0;
  obj->userfunc0= userfunc0;
//#1: Add Times Expectations
  if(startvararg == NO_MORE_PARAMETERS ){ // use the
     default setting
   printf("no more parameters  \n ");
   return;
  }
  va_list ap;
  va_start(ap, startvararg);
  int time_value;
  if(startvararg == TIMES || startvararg ==
     TIMES_AT_LEAST || startvararg == TIMES_AT_MOST){
    time_value = va_arg(ap,int);
    struct Expectation* exp = new Expectation();
    exp->type_expectation = startvararg;
    int* p = (int*)malloc(sizeof(int));
    *p = time_value;
    exp->returnvalue = (void*)p;
    obj->list_expectations.push_back(exp);
  }else{
    time_value = va_arg(ap,int); // to jump the "
       time_value parameter" used from the TIMES macro
  }
//#2: Manage WILL_ONCE and WILL_REPEATEDLY
  int type_expectation; // the type of the variable is as
      the type return function
  int return_value;
  for(type_expectation = va_arg(ap, int);
     type_expectation != NO_MORE_PARAMETERS;
     type_expectation = va_arg(ap,int) ){
    struct Expectation* exp = new Expectation();
    return_value = va_arg(ap, int);// the return type is
       always as the return type of the original function
       . We are treating just WILLONCE and REPEATEDLY
    int* p = (int*)malloc(sizeof(int));
    *p = return_value;
    exp->type_expectation = type_expectation;
    exp->returnvalue = (void*)p;
    obj->list_expectations.push_back(exp);
  }
  va_end(ap);
}
```

Note, the source code to manage the macros *WILL_ONCE* and *WILL_REPEATEDLY* is not generated, if the respective external function returns has *void* as return type.

5.5 New mock functions

The plugin generates the new mock functions when the function *write_external_functions*, in Listing 5.5, is called. The SUT external functions will be replaced with the the new mock functions by linking the *temporary_file.cpp* and the SUT.

The new mock functions will be able to check at runtime, if they are running in according at the expectation settings. This is done by comparing the arguments received (when the mock function is called), with the values maintained in memory, though the structures that we have seen previously.

If we refer at the SUT external function declaration in Listing 5.2, and its respective behavior in Listing 5.4, then the plugin writes a new mock function into the temporary file; the source code follows in Listing 5.23.

Listing 5.23. *framework.h*

```
extern "C"
int foo(int param0){
  ..
}
```

This new mock function, first checks if *param0* is equal at the value 5 or everything else, as set in Listing 5.4. Then, it checks if it has been called the right number of times. Finally, it gives back at the SUT the proper return value. If the SUT external function was declared with *void* as return value, then this last step is not performed.

In general, when a mock function x is called, it iterates the respective list: *list_x*. It checks if the received arguments match the values contained into one of the possible *Mock_x* structures contained in *list_x*. We call such kind of control: the *matching parameter* control. If there is not even a matching, the system will print out an error. Otherwise if a matching occurs, then the mock function x must check, if it has been called within of a right interval of times. Finally, if the called at x respects such interval, the mock function will update the data *Mock_x* structure and it will give back to the SUT the right return value.

The *matching parameter* control is performed directly in the mock function. The source code to perform such control is generated from the plugin. The *matching parameter* control works as follows: when the mock function is called, the values of the arguments received are compared with the corresponding parameters contained in the respective *Mock_* structure. We remember that the *Mock_* structure contains an unique integer values to indicate which macros were used during the expectation settings. By accessing these values, the mock function knows which kind of controls to perform. In addition, the kind control will change in according at the type of macro used (Le, VALUE, STRING, etc..).

As previously stated, the unique integer values are defined into the file *framework.h*, described in Listing 5.1 and Listing 5.2. All these macros are suitably redefined into the header file *helper.h*, as shown in Listing 5.24.

Listing 5.24. *helper.h* - Macros redefinition.

```
//VALUE(x)
#define CHECK_PARAMS_VALUE  0
#define ANY_VALUE           1
//STRING(x)
#define CHECK_PARAMS_STRING 2
/*     USER_FUNCTION(x)    x is the address of the function
    to use. Return 0 if the test is positive. Otherwise
   the test will fail.
*/
//USER_FUNCTION(x)
#define USER_FUNCTION       3
#define Gt                  4
#define Lt                  5
#define Ge                  6
#define Le                  7
#define Ne                  8
```

When the *matching parameter* control is positive, the mock function checks the *time interval.* If also this latter is respected, then the mock function returns the appropriate value. These two operations are performed by calling a function built in the runtime system SUT independent. In fact, since the structure *Expectation* in Listing 5.14 is independent from the SUT external functions; we can develop a SUT independent source code.

Having a part of the runtime system, SUT independent, involves several advantages. First, the project development is faster: we write the source code directly into a C++ file instead to generate the code via plugin. Second, it will improve the project's performance in phase of compilation: the plugin has less source code to generate.

In general, each mock function calls *check_expectations* if the *matching parameter* control is positive. On the contrary, it will print in output an error, by calling *manage_errors*. Both the functions *check_expectations* and *manage_errors* are defined in *helper.cpp*.

In Listing 5.25 is shown a part of the source code generated if we are testing a SUT with the same external function *foo*.

Listing 5.25. *temporary_file.cpp* - A part of the mock function's source code.

```
if(check_parameter == true)
{
 return_value = check_expectations(obj->list_expectations
    , "foo");
}else{
 manage_errors(DIFFERENT_PARAMETER, "foo");
 return_value = NULL;
}
if(return_value == NULL)//Error in check_expectations
{
 return NULL;
}
double casting_return_value = *(double*) return_value;
return casting_return_value;
```

When the *matching parameter* control is positive the variable *check_parameter* is true and the *check_expectations* is called; otherwise an error is managed with the *manage_errors* function. Obviously, the plugin generates the source code to manage the return value, only if its respective external function has actually a return value.

For more detail, we suggest to read the source code in the Appendix 2.

5.6 Runtime system: SUT independent part

The runtime system SUT independent part performs all the tasks that we do not need to generate with the plugin, as the *time interval* checking through the function *check_expectations* and manage the errors with *manage_errors*.

When a mock function calls *check_expectations*, it will pass two parameters. A pointer at the *list_expectations* field of the *Mock_* structure; in which the *matching parameter* control was resulted positive. And the name of the mock function, as a string.

The first *Expectation* structure (Listing 5.14) of such list, it must always contain the values defined with a cardinality macro. Hence, we compare the number of times that the mock function has been called y, with the number of times set during the expectation settings z. If y is less or equal of z, then the interval is still acceptable and we can continue to iterate the list, to get the return value. The return value is got with the next *Expectation* structure of the list. On the contrary, if the interval is not respected, an error is printed in output.

In Listing 5.26 follows the *check_expectation* source code.

Listing 5.26. *helper.cpp - check_expectation* function

```
void* check_expectations(list<Expectation*>& list_exp,
   string function_name){
  if( list_exp.size() == 0 ){
    return NULL;
  }
  //The first expectation it has to be always TIMES,
     TIMES_AT_LEAST or TIMES_AT_MOST
  list<Expectation*>::iterator it_exp = list_exp.begin();
  Expectation* exp_times = *it_exp;
  int total_number_calls = *(int*)exp_times->returnvalue;
  ++it_exp;
  exp_times->number_calls ++;
  if(exp_times->number_calls > total_number_calls){
    manage_errors( TIMES_CALLS_EXCEEDED, function_name);
    return NULL;
  }
  //a mock function that correspond at a SUT external
     function that returns void will not continue with
     this loop.
  for (int x = 1; it_exp != list_exp.end(); ++it_exp, x
     ++){
    switch((*it_exp)->type_expectation){
      case WILLONCE:
    if(x == exp_times->number_calls){
      return (*it_exp)->returnvalue;
    }
    break;
      case WILL_REPEATEDLY:
        return (*it_exp)->returnvalue;
    break;
    }
  }
  return NULL;
}
```

In addition, the plugin generates, in the temporary file, the source code to call a function called *check_expectations_test_finished*. This function is called to verify that the number of calls at the respective mock function has been respected for each expectation set by the tester.

If we refer again at the SUT external function shown in Listing 5.2 and its expectation setting in Listing 5.4; the function *check_expectations_test_finished* will be called twice, once for each *Mock_foo* structures present in memory. In this particular example, the plugin will generate into the temporary file the source code shown in Listing 5.27.

Listing 5.27. *temporary_file.cpp* - the *testCompleted* function called at the end of the test.

```
void testCompleted(){
  expectation_number = 1;
  for ( list<Mock_foo*>::iterator it = list_foo.begin();
     it!= list_foo.end() ;expectation_number++, ++it){
    struct Mock_foo* obj_foo = *it;
    std::stringstream ss ;
    ss << "foo" << " from Expectation number: "<<
       expectation_number;
    check_expectations_test_finished(obj_foo->
       list_expectations , ss.str() );
  }
}
```

Note that for each SUT external function found, the plugin generates a loop equal to that shown in the source code above.

The *testCompleted* function must be called from in the file *test_sut.c*, at the end of the test by using the macro *TEST_FINISHED* defined in the header file *temporary_file.h*.

5.7 The Makefile

The Makefile is an important component of this project, it controls the exact order of the compilation steps, to test the SUT.

The files used during this project are summarized here:

- *plugin-mocking.cxx*: It is the GCC-plugin developed during this thesis.
- *test_sut.c*: It contains all the expectation of the SUT external functions. This file is written by the user.
- *SUT.c*: It contains the peaces of software that we want to test.
- *temporary_file.cpp*: It will contains all the new mock and expectation functions. It is generated when the SUT is compiled with the plugin loaded in GCC.

- *temporary_file.h*: It is generated through the plugin and it is included by the *test_sut.c* to define the expectation functions and the *TEST_COMPLETED* macro.
- *helper.cpp*: It contains the runtime system SUT independent part.
- *helper.h*: Header file of the runtime system SUT independent part. It is included in *temporary_file.cpp*.
- *framework.h*: It defines the API of the project.

The Makefile performs these compilation steps with the following order:

In the first step, we compile the plugin developed, with the command seen during the third Chapter.

Listing 5.28. Makefile - Step 1.

```
#step 1
plugin.so: plugin-mocking.cxx
        g++ -I‘g++ -print-file-name=plugin‘/include -fPIC
            -shared plugin-mocking.cxx -o plugin.so
```

Then, we compile the software under test with the plugin loaded. In this step the temporary file and the SUT object file are generated.

Listing 5.29. Makefile - Step 2.

```
#step 2
testc:  plugin.so SUT.c
        gcc -c -g -fplugin=./plugin.so SUT.c -o SUT.o
```

Third step: we compile the runtime system SUT independent part by avoiding the GCC linking phase.

Listing 5.30. Makefile - Step 3.

```
#step 3
helper.o: helper.cpp helper.h
        g++ -c -g helper.cpp
```

Step 4, also the temporary file is compiled avoiding the GCC linking phase.

Listing 5.31. Makefile - Step 4.

```
#step 4
temporary_file.o: temporary_file.cpp
    gcc -c -g temporary_file.cpp
```

Step 5, we compile the file written by the tester in which are setted the external function behaviors.

Listing 5.32. Makefile - Step 5.

```
#step 5
test_SUT.o: test_SUT.c
    gcc -g -c  test_SUT.c
```

Finally, it is performed the redirection of the SUT external functions to the new external functions. In addition, in this step we also link the expectation functions contained into the temporary file with the expectations called in the *test_SUT.c* file. The result is an executable file.

Listing 5.33. Makefile - Step 6.

```
result: temporary_file.o SUT.o  test_SUT.o helper.o
    g++ -g -o result SUT.o temporary_file.o  test_SUT.o
       helper.o
```

By running the executable file, we are able to test our software and see if everything is fine as we expected.

5.8 Manage the User Data Structures

We have seen in the third chapter that we can access via macro at the AST nodes to read the user data structures; but this is not enough to manage them. The temporary file must know where are declared the SUT structures used by the SUT external functions. We have two choices. We can recreate them into a new header; and then import this header in the temporary file. Or otherwise, we can leave a way at the tester to indicate where the temporary file can find the SUT structures definitions.

We decided to use the second choice and leave the first alternative as future work. The tester will include the files containing the SUT structure definitions into the header file: *user_headers.h*.

If for example, the SUT have an adhoc structure declared in its header file *SUT.h*, as shown IN Listing 5.34.

Listing 5.34. *SUT.h* - SUT adhoc structures management example.

```
typedef struct user_data{
  struct user_data* next_u;
  int key;
  int value;
};
```

The tester must import such header file into the file *user_headers.h* as follow:

Listing 5.35. *user_headers.h*

```
#include "SUT.h"
```

In this way, all the SUT headers can be included easily in one unique file. The *user_headers.h* files will be visible both for the SUT purposes and for the temporary file. We remember that during the expectation settings, the tester is able to check the user structure type dates with *matcher* macro *USER_FUNCTION*.

However, in our plugin, there is also a function that implement the first choice, it is called *write_user_structs*. It rebuilds all the user data definition, but it is actually not used.

For more detail, see the source code in the Appendix 2.

5.9 Debugging the Project

Compiling GCC directly from its the source code, we will be allowed to access at the GCC internal source code lines in case of debugging. An interesting case is when we want to debug the step 2 of the Makefile: when the plugin is reading the software to test. Such command is shown in Listing 5.36.

Listing 5.36. Debugging command

```
//wrapper with gdb:
gcc -wrapper gdb,--args -g -S -fplugin=./plugin.so SUT.c
```

However, it is possible to debug all the other files with the usual way: adding the flag *-g* in the compilation, and then use *gdb*.

Chapter 6

Evaluation

In this chapter, we will describe, with examples, how the system works and which are the results. Finally, we will compare the system developed with other existing frameworks.

6.1 SUT with an User data structures.

One open question was how to handle parameters with user-defined types such struct. We will show in this section, how our plugin manages these kind of types through an example. Actually the plugin can manage two user-defined types: the structures and the enumerations.

In this example, we mean to mock the external function *set_person* used in SUT, as shown in Listing 6.1. Such function returns a pointer to the user structure *person*.

The function that we want to test is, easily called, *function_to_test*. It prints the first name of the user structure *person* received when *set_person* is called. In Listing 6.2 is also shown the SUT header source code.

Listing 6.1. *system_under_test.c* - Example with the user structure *person*.

```
#include <stdio.h>
#include "system_under_test.h"
person * set_person(char* first_name, char* last_name,
   int age, SEX x);

int function_to_test()
{
  person* y;
  y = set_person("Luca", "Ciavaglia", 23, MALE);
  printf("name: %s\n", y->first_name);
  return 1;
}
```

Listing 6.2. *system_under_test.h* - Example with the user structure *person*.

```
typedef enum
{
  MALE,
  FEMALE
} SEX;

typedef struct{
  int   age;
  SEX   sex;
  char* first_name;
  char* last_name;
} person;
```

In this test case, we check if the *set_person* function is called with the right parameters and if it sends back our return value. This is easily settable in one expectation, as shown in Listing 6.3.

Listing 6.3. *test_sut.c* - Example with the user structure *person*.

```
#include <stdio.h>
#include "myframework.h"
#include "temporary_file.h"//import expectation functions
    MACROs
int check_sex_male(SEX y){
  if(y == MALE){
    return 0;//the test is POSITIVE.
  }
  return 1;
}
void test_sut(){
  person p ;
  p.first_name = "Luca";
  p.last_name  = "Ciavaglia";
  p.age = "23";
  EXPECT_set_person(STRING("Luca"),
            STRING("Ciavaglia"),
            Ge(18),
            USER_FUNCTION(&check_sex_male),
                TIMES(1),
                    WILL_ONCE(RETURN(&p))
                );//#1
  function_to_test(); //Execution: Call the function to
     test!
}
int main(int argc, char **argv)
{
  test_sut();
  TEST_FINISHED;
  return 1;
}
```

We remember that we must include the header file *system_under_test.h* in the header *user_headers.h*, as explained in section 5.8.

At this point, if we compile the above source codes with our Makefile and if we run the executable got with the compilation; we will see the test running without error messages.

On the contrary, if we modify the expectation of before, by replacing the macro *Ge(18)* with macro *VALUE(18)* as follow:

Listing 6.4. Expectation 2

```
EXPECT_set_person(
STRING("Luca"),
STRING("Ciavaglia"),
VALUE(18),
USER_FUNCTION(&check_sex_male),
TIMES(1),
WILL_ONCE(RETURN(&p))
);
```

The system will print in output the following error:

Listing 6.5. Error message raised by the Expectation 2.

```
=== TEST FAIL ===
From set_person
Parameters do not matching
=================
Segmentation fault (core dumped)
```

In fact, when the external function *set_person* is called, it expects to receive exactly the value *18*, but it will receive instead the value *23*.

The *segmentation fault* error above is caused from the *printf* in Listing 6.1. Since there was an error during the matching of the parameters, the system did not give back a return value and the *printf* function will try to print a value that is not in memory.

Next, we get a different kind of error, if we replace Expectation 2 with the new one shown in Listing 6.6:

Listing 6.6. Expectation 3.

```
EXPECT_set_person(STRING("Luca"),
  STRING("Ciavaglia"),
  Ge(18),
  USER_FUNCTION(&check_sex_male),
  TIMES_AT_LEAST(2),
  WILL_REPEATEDLY(RETURN(&p))
);
```

In which, we have stated that the function *set_person* should be called at least two times; and for every time, it should return the structure *person p*, as shown previously in Listing 6.3.

This time the error is caused because the function *set_person* is called only once during the *function_to_test* execution, in Listing 6.1. The error message printed in output to describe such situation is the following:

Listing 6.7. Error message raised by Expectation 3

```
=== TEST FAIL ===
The function "set_person from Expectation number: 1"
should be called AT LEAST 2 times
but it is called just 1 time
=================
```

As we may notice from the message above, the system recognizes that the function *set_person* is called only once, when it was expected to be called at least twice. This last error message derives from the function called at the end of the test, with the macro *TEST_FINISHED*; such macro was described in Section 5.6.

6.2 Comparison with JMock and GMock.

In the second chapter, we have exposed an example in Java with two different frameworks: JUnit and JMock. The source codes of such examples are shown respectively in Listing 2.1 and in Listing 2.2. We re propose in this section the same example, but this time in C, using the system realized in this project. The purpose of that example was to test the *fill* function of the object Order. The *fill* function works together with the Warehouse object; in which it holds inventories of different products. The *fill* function works as follow: if Warehouse has enough quantity of the same product defined in Order, the order becomes filled and the warehouse's amount of product is properly reduced. Otherwise, the Order is not filled and the state of Warehouse does not change. The Order object was the system under test and Warehouse object its collaborator.

We can rethink these two objects in two C files: *order.c* and *warehouse.c*. Since our plugin generates automatically mock external functions, we suppose to not have the file *warehouse.c*, but only its header file *warehouse.h*, included in *order.c*.

Keeping the idea of the Order Object as simple as possible, we propose the source codes in *order.c* and in *warehouse.h* files respectively in Listing 6.8 and Listing 6.9.

Listing 6.8. *warehouse.h* - Warehouse header.

```
int warehouse_has_inventory(char* name, int quantity);
void warehouse_remove(char* name, int quantity);
```

Listing 6.9. *order.c* - The Order object re proposed in C language.

```
#include <stdio.h>
#include "warehouse.h"

int is_filled;

void create_order(){
 is_filled = 0;
}

void fill(char* name, int quantity){ // function to test
 if( warehouse_has_inventory(name, quantity) == 0 ){
   warehouse_remove(name, quantity);
   is_filled = 1;
 }
}

int isFilled(){
  return is_filled;
}
```

From the source code above, the plugin should be able to automatically mock both the functions *warehouse_has_inventory* and *warehouse_remove*. We would be able to have the same expectations set as the Java examples. Hence, we propose the file *test_order.c* shown below in Listing 6.10.

Listing 6.10. *test_order.c* - The equivalent test file in C language of the respective Java Example shown in Listing 2.2.

```
#include <stdio.h>
#include "myframework.h"
#include "temporary_file.h" //import MACROs for the
   expectation functions
void test_order_fillingRemovesInventoryIfInStock(){
  EXPECT_warehouse_remove(STRING("Talisker"),
                    VALUE(50),
                    TIMES(1)
                   );
  EXPECT_warehouse_has_inventory(STRING("Talisker"),
                 VALUE(50),
                 TIMES(1),
                 WILL_ONCE(RETURN(0) )
                );
  fill("Talisker", 50); //Execution: Call the function to
      test!
}
int main(int argc, char **argv)
{
  create_order(); // Set up phase.

  test_order_fillingRemovesInventoryIfInStock(); //Start
     the test.
  TEST_FINISHED; // Verification - part 1 phase

  /*Source code corresponding to the Java source code
     line in
    Listing 2.2:
    assertTrue(order.isFilled());
   */
  //Verification - part 2 phase
  if(isFilled() != 1){
    printf("Test failed\n");
  }else{
    printf("Test PASSED\n");
  }
  return 1;
}
```

We can easily modify the makefile to compile the file above: the *SUT.c* is replaced with the file *order.c*, whereas *test_SUT.c* is replaced with the *test_order.c* file. When we run the *Makefile*, we get the executable file. By launching the executable, we can see in output the message: *Test PASSED*. The temporary file source code generated by the plugin is shown in the Appendix, in Listing 9.

When we started this thesis, we mainly wanted to ask at two questions. The first one was if we could use plugin compilers to automatically create mocks for external functions. The second question was if we could create a runtime system as powerful almost as Google mock in pure C. Comparing the example above with that one done with JMock, we can state that we have achieved a good result: we are able to mock automatically the SUT external functions; and the API developed allows us to set the expectations as happen in Listing 2.2. Obviously, our API is poor compared with the API of JMock. For example, we do not have a way to define that the function *warehouse_remove* should be executed *after* the function *warehouse_has_inventory*. The same aspect is valid if we consider to compare our project with Google Mock. Our API is still poor than the Google Mock API. However with this thesis, we have proved: first that it is possible to build a new framework in C ables to automatically mock external functions. Second, we have created a runtime system almost powerful as Google Mock. The API developed allow us to set a basic function behaviors, but of course it can be improved in future by adding new features. It would be easy to improve our API. In fact, we have developed all the project in manner to add easily new features. For example, we may add the macro *AFTER("x")* to indicate that an external function should be called after another external function *x*. We could do this, because first of all, all the expectation function declarations are built to accept a variable number of parameters, it means there are no problems to add new macros. Second, we have seen that by using macros, we can hide to the tester the number of parameters behind that macro, in this way we can distinguish different macros just with a simple integer value. For example we may add the definition of the macro *AFTER* in *framework.h* and *helper.h* as shown in Listing 6.2.

```
--- framework.h ---
//x may be a string that contain the name of the function
    as happens in JMock
```

```
#define AFTER(x) 250, x

--- helper.h   ---
#define AFTER 250
```

Third, we may always manage new features by adding new independent data structures (as the *Expectation* structure shown in Chapter 5) and new functions in the runtime system SUT independent part (in *helper.h* and *helper.cpp*); then we may call these new functions by generating with the plugin, new few lines of code in the temporary file. In general, it would not be difficult to add new features at our project.

Some example of features present on Google Mock, but no in our project are:

- further *generic comparison matchers* as *isNull* and *NotNull*, to check if the argument is a NULL pointer,
- *floating-Point Matchers* to check if an argument is a double or float value approximately equal to what it has been specified in the expectation (in our framework this kind of control can be easily done by defining an user function and set such function with the macro *USER_FUNCTION*, as we have seen in Chapter 5),
- additional *string matchers* for particular controls on string values;
- or also *composite matchers* to connect different kind of matchers.
- In Google Mock there is also the possibility to define the exact order in which the mocked function should be called (this feature could be implemented in our framework by keeping in the runtime system memory an array, with the exact order of the external function names to be called).

Another interesting, possible future improvement could be, to integrate our project with an existent xUnit framework. In the source code above, we have replaced the Java line code: `assertTrue(order.isFilled())`, with an *if* C construct, to check if the *fill* function has worked as expected: it controls the value of the integer variable *is_filled*. In the example in Java, that kind of control was done with the JUnit framework. In our project, such control could be done with one of the xUnit C frameworks available.

By working with this thesis, we have also noted that in Google Mock is not possible to mock a function with a variable number of parameters, the variadic functions. An example of this type of function can be the well known function *printf*. Note, we have developed the plugin to not mock functions from the GCC standard library.

The author thinks that it is possible to implement such feature in this project. In fact, the plugin generates an exactly copy of the SUT external function declaration in a new mock function declaration (into the temporary file). Actually the plugin does not recognize a function parameter composed of three dots (such dots are utilized to indicate that it is a variadic function): if we suppose to have in the SUT, an external function *foo(int x, ...)*, the plugin does not know how to reply to these dots in the temporary file. However, into the AST tree, there are all the necessary information that we should need to manage such case.

A critic at the project is that, we did not work at the appearance of the messages in output. It could be well-accepted at the user, to have a better graphic. Moreover, the plugin does not manage all the possible different types in C. A future task could be to manage the union type and boolean.

Chapter 7

Summary and Conclusion

In this thesis we have realized a framework to test part of a software in isolation. To isolate a software, we need to replace its dependencies and we have chosen to replace its dependencies by using the *mocking* technique. This technique is more often used in object oriented language than in procedural languages as such C (see Chapter 2); it allows to set the behaviors of the external functions of the software to test.

Actually, there are no available frameworks that use the plugin compiler mechanism to access the software information to automatically create mocks; hence we decided to investigate this area. The framework developed aims at making mocking easier in C. It reads the abstract syntax tree built when GCC parses the software to test with a GCC plugin; as we have explained in the Chapter 3 and 4; then when the framework knows which are the external functions to mock, it generates new mock functions in a temporary file. The new mock functions replace the respective external ones, as explained in Chapter 5.

At the end of this thesis, we can state that it is possible to use plugin compilers to automatically create mocks for external functions. Moreover, we have also shown that there is the possibility to create such system almost powerful as Google Mock in pure C. These were the main points at which we wanted to answer at the research questions exposed in Chapter 1. Obviously, it is out of the scope of this thesis to realize a system powerful as Google Mock. Google Mock has been development with a professional team for several years.

The runtime system is actually developed in C++ and this decision allowed us to be faster during the software development, we have considered that it would be easy write the runtime system completely in pure C.

With this framework written in pure C, it would make possible to run tests on C-only system, as in DSP platforms; where for example, it is not possible to use frameworks as Google Mock, developed in C++.

An interesting topic for future work could be to access at the GCC abstract syntax tree to specify special attributes when the compiler parses a DSP-C (a C dialect) code. It would be also possible to have a plugin that creates and inserts the new mock functions by modifying directly the AST of the system under test, instead to generate temporary files.

Summurizing, we have answered at the research questions exposed in Chapter 1; and we are satisfied from the achieved result of the system developed, as discussed in Chapter 6.

Bibliography

[1] Clang: a c language family frontend for llvm. http://clang.llvm.org/.

[2] Cmock. http://throwtheswitch.org/white-papers/cmock-intro.html.

[3] Easymock. http://easymock.org/.

[4] Expectations setting gmock. http://code.google.com/p/googlemock/wiki/ForDummies#Using_Multiple_Expectations.

[5] Gcc melt. http://gcc-melt.org/.

[6] Gcc plugin api. http://gcc.gnu.org/onlinedocs/gccint/Plugin-API.html#Plugin-API.

[7] Gcc plugin site. http://gcc.gnu.org/wiki/plugins.

[8] Gmock. http://code.google.com/p/googlemock/.

[9] Jmock. http://jmock.org/.

[10] Junit framework. http://junit.org/.

[11] Testdept. http://code.google.com/p/test-dept/.

[12] Extreme programming: A gentle introduction. http://www.extremeprogramming.org/ (1999).

[13] Aho, A. V. et al. *Compilers: principles, techniques, & tools.* Pearson Education India (2007).

[14] Bender, J. *Professional test-driven development with C#: developing real world applications with TDD.* Indianapolis, Ind. ISBN: 9781118102107.

[15] Fowler, M. Mock aren't stubs. http://martinfowler.com/articles/mocksArentStubs.html.

[16] Kent Beck, ., Arie van Bennekum. Manifesto for agile software development. `http://agilemanifesto.org/` (2001).

[17] Kolawa, D., Adam; Huizinga. Automated defect prevention: Best practices in software management. *Wiley-IEEE Computer Society Press. p. 74*, (2007).

[18] McConnell, S. *Code complete.* O'Reilly Media, Inc. (2004).

[19] Meszaros, G. *XUnit Test Patterns: Refactoring Test Code.* Addison-Wesley (2007).

[20] Raffeiner, S. Functionality and design of the cmock framework.

[21] Smith, S. W. *Digital signal processing: a practical guide for engineers and scientists.* Access Online via Elsevier (2003).

[22] Stober, T. and Hansmann, U. *Agile Software Development.* Springer-Verlag (2010). ISBN 978-3-540-70830-8. Available from: `http://dx.doi.org/10.1007/978-3-540-70832-2`, `doi:10.1007/978-3-540-70832-2`.

Appendix

.1 Project source code

.1.1 *Makefile*

Listing 1. *Makefile.*

```
GXX := g++
GCC := gcc
CXXFLAGS := -Wshadow -Wall
CFLAGS := -Wshadow -Wall

SYSTEM_UNDER_TEST := system_under_test
TEST_FILE := test_sut

.PHONY: testc testcc


result: temporary_file.o $(SYSTEM_UNDER_TEST).o  $(
   TEST_FILE).o helper.o
    $(GXX) $(CXXFLAGS) -g -o result $(SYSTEM_UNDER_TEST).
       o temporary_file.o  $(TEST_FILE).o helper.o

$(TEST_FILE).o: $(TEST_FILE).c
    $(GCC) $(CFLAGS) -g -c  $(TEST_FILE).c

temporary_file.o: temporary_file.cpp
    $(GXX) $(CXXFLAGS) -c -g temporary_file.cpp

temporary_file.cpp: testc

helper.o: helper.cpp helper.h
    $(GXX) $(CXXFLAGS) -g -c helper.cpp

$(SYSTEM_UNDER_TEST).o: $(SYSTEM_UNDER_TEST).c
    $(GCC) $(CFLAGS) -g -c $(SYSTEM_UNDER_TEST).c

testc:  plugin.so $(SYSTEM_UNDER_TEST).c
```

```
    $(GCC) $(CFLAGS) -g -S -fplugin=./plugin.so $(
       SYSTEM_UNDER_TEST).c

plugin.so: plugin-declation.cxx
    @echo Compiling plugin.so
    $(GXX) $(CXXFLAGS) -g -I‘$(GXX) -print-file-name=
       plugin‘/include -fPIC -shared plugin-declation.cxx
        -o plugin.so

clean:
    rm -f plugin.so temporary_file.o temporary_file.cpp
       result temporary_file.h.gch temporary_file.h
       helper.o temporary_file.cpp~ temporary_file.h~
       helper.h~ temporary_file.c~ myframework.h~ plugin
       -3.cxx~ helper.cpp~ $(SYSTEM_UNDER_TEST) $(
       TEST_FILE).o $(SYSTEM_UNDER_TEST).o
```

.1.2 *plugin-mock.cxx*

Listing 2. *plugin-mock.cxx.*

```
//#include <stdlib.h>
#include <gmp.h>
#include <cstdlib> // Include before GCC poison some
   declarations.


#include "gcc-plugin.h"
#include "config.h"
#include "system.h"
#include "coretypes.h"
#include "tree.h"
#include "tree-iterator.h"
#include "intl.h"

#include "tm.h"
#include "diagnostic.h"

#include <set>


//my inclusions
#include <list>
#include <string>
#include <sstream>
#include <iostream>
#include <fstream>
```

```
using namespace std;

int plugin_is_GPL_compatible;

list<tree_node*> list_functions_with_only_declaration;
list<tree_node*> list_functions_with_body;
list<tree_node*> list_functions_to_mock;

ofstream myfile;
ofstream headerfile;
ofstream userStructDefined_file;


string manage_types(tree , bool , bool);

void print_node_declaration(tree_node* node_declaration){
  int treecode = TREE_CODE(node_declaration);
  tree id = DECL_NAME(node_declaration);
  string source_file = DECL_SOURCE_FILE(node_declaration)
     ;
  int number_line =  DECL_SOURCE_LINE(node_declaration);
  const char* name (id ? IDENTIFIER_POINTER (id): "<
     unnamed>");
  cout << "*** Tree node:"  << tree_code_name[treecode]
     << " Function name: " << name << " in "
       <<  source_file <<"  number line: " << number_line
           << " ***" << endl;
}
void cb_finish_decl(void *gcc_data, void *user_data){
  //cout << "****** EVENT PLUGIN_DECLARATION *******" <<
     endl;
  tree_node *ast_node = reinterpret_cast<tree_node*>(
     gcc_data);
  int treecode = TREE_CODE(ast_node);

  //  print_node_declaration(ast_node);

  if(treecode == FUNCTION_DECL){// we are interested only
      to the function that are rapresented in the AST as
     a FUNCTION_DECL node
    string source_file = DECL_SOURCE_FILE(ast_node);
    size_t found = source_file.find("/usr/include");
    if(found==string::npos ){// we are NOT interested to
       mock library functions.
      print_node_declaration(ast_node);
      //      list_functions_to_mock.push_back(ast_node);
      list_functions_with_only_declaration.push_back(
        ast_node);
    }
  }
}

//
   ==================================================================

/*
     We save all the function definitions
 */
void cb_plugin_pre_generize (void *maintree, void*)
{

  cout <<  "************** PLUGIN_PRE_GENERIZE EVENT
     *******************"<< endl;
  tree_node *tn = reinterpret_cast<tree_node*>(maintree);
  cout << " --- Function name: " << IDENTIFIER_POINTER(
     DECL_NAME(tn)) << " ---" << endl;
  // print_node_declaration(tn);
  int treecode = TREE_CODE(tn);
  //if(treecode == FUNCTION_DECL){
      list_functions_with_body.push_back(tn);
    //}
  cout << "*************** Ends PLUGIN_PRE_GENERIZE
     EVENT *************" << endl;
}

//
   ==================================================================



void print_list(const list<tree_node*>& list_){

  for ( list<tree_node*>::const_iterator it = list_.begin
     (); it != list_.end(); ++it){
    print_node_declaration(*it);
  }

}

string manage_enumeral_type(tree node, bool isPartial){
  cout << "== manage enumeral type ===" << endl;
  string name_enumeration;
  std::stringstream user_enumeration;
  if (TYPE_NAME (node)){

    if (TREE_CODE (TYPE_NAME (node)) == IDENTIFIER_NODE){
      name_enumeration = IDENTIFIER_POINTER (TYPE_NAME (
         node));
    }

    else if (TREE_CODE (TYPE_NAME (node)) == TYPE_DECL
         && DECL_NAME (TYPE_NAME (node))){
      name_enumeration = IDENTIFIER_POINTER (DECL_NAME (
         TYPE_NAME (node)));
    }
  }
  cout << " the name of the enumeration is: " <<
     name_enumeration << endl;
//  debug_tree(node);

  if(isPartial){
    return name_enumeration;
  }

  user_enumeration << "typedef enum \n{" << endl;
  tree node_list = TYPE_VALUES(node);

  while(node_list != NULL){
    user_enumeration <<  IDENTIFIER_POINTER(TREE_PURPOSE(
       node_list)) ;
    // cout  << IDENTIFIER_POINTER(TREE_PURPOSE(node_list
       )) << ";" << endl;
    node_list = TREE_CHAIN(node_list);
    if(node_list!=NULL){
      user_enumeration << "," << endl;
    }else{
      user_enumeration << endl;
    }
  }

  user_enumeration << "}" << name_enumeration <<";" <<
     endl;
  //cout << "complete enumeration: \n" <<
     user_enumeration <<  endl;
  //cout << "print the enumeration: "<< user_enumeration.
     str() <<  endl;
  return user_enumeration.str();
}

string manage_record_type(tree node, bool isPartial) //
```

```
   if ispartial == true then return only the "struc
   nameofthestruct"
{
  //string user_struct = "struct ";
  string name_struct;

  std::stringstream user_struct;
  debug_tree(node);
  if (TYPE_NAME (node)){
    if (TREE_CODE (TYPE_NAME (node)) == IDENTIFIER_NODE){
      name_struct = IDENTIFIER_POINTER (TYPE_NAME (node))
         ;
    }
    else if (TREE_CODE (TYPE_NAME (node)) == TYPE_DECL
         && DECL_NAME (TYPE_NAME (node))){
            name_struct = IDENTIFIER_POINTER (DECL_NAME (
               TYPE_NAME (node)));
    }
  }
  cout << " the name of the struct is: " << name_struct
     << endl;
  //user_struct = user_struct + name_struct + "{\n" ;
  if(isPartial){
    user_struct << "struct "<< name_struct ;
    return user_struct.str();
  }

  user_struct << "struct " << name_struct << "{" << endl;
  //Navigate struct's type fields.
  tree node_field_chain = TYPE_FIELDS(node);
  tree field;
  int number_param = 0;
   while(node_field_chain !=NULL ){
     field = TREE_TYPE(node_field_chain);
     //debug_tree(field);

     user_struct <<"  "<< manage_types(field,false,false)
         << " param" << number_param << ";" << endl;

     node_field_chain = TREE_CHAIN(node_field_chain);
     number_param ++;
   }
   user_struct << "};" <<  endl;
   //cout << "complete struct: \n" << user_struct <<
      endl;
   cout <<"print the struct: "<< user_struct.str() <<
      endl;
   return user_struct.str();

}


 string
 manage_types(tree nodetype, bool isPuntator, bool
    isPartial)
{
  string result;
  cout << " *** Into manage_types() ***" << endl;
  if(isPuntator)
    cout << "POINTER  to an: "<< endl;
  //debug_tree(nodetype);
   //  cout << "Node Type: " <<  tree_code_name[TREE_CODE
      (nodetype)] << endl;
  switch(TREE_CODE(nodetype)){
   case INTEGER_TYPE:
     if( TYPE_STRING_FLAG(nodetype) == 1 ){
       result = "char";
       //   cout <<" Type: char " <<  endl;
     }else{
       //cout <<" TYPE_PRECISION:  " << TYPE_PRECISION(
          nodetype)  << endl;
       switch( TYPE_PRECISION(nodetype)){
        case 16 :
      result = "short";
      // cout << "type: short"<< endl;
          break;
        case 32 :
       result = "int";
      //cout << "type: int"<< endl;
          break;
        case 64 :
       result = "long";
      //cout << "type: long"<< endl;
          break;
       }
     }
     break;

  case REAL_TYPE:
    // cout <<" TYPE_PRECISION:  " << TYPE_PRECISION(
      nodetype)  << endl;
    switch( TYPE_PRECISION(nodetype)){
      case 32 :
     result = "float";
    //  cout << "type: float"<< endl;
    break;
      case 64 :
     result = "double";
    //  cout << "type: double"<< endl;
    break;
      case 80:
      result = "long double";
     // cout << "type: long double" << endl;
    }
    break;

  case VOID_TYPE:
     result = "void";
    break;
  case COMPLEX_TYPE:
    break;
  case ENUMERAL_TYPE:
    if(isPartial){
      result = manage_enumeral_type(nodetype, true);
    }else{
      result = manage_enumeral_type(nodetype, false);
    }
    break;
  case BOOLEAN_TYPE:
    break;
  case POINTER_TYPE:
     cout << "Into POINTER_TYPE" << endl;
    // debug_tree(nodetype);
    //cout << "==================" << endl;
    //debug_tree(TREE_TYPE(nodetype));
    // It is really important give true to the 3rd
       parameter to manage the struct list. Otherwise it
       will be an infinite loop
    result = manage_types(TREE_TYPE(nodetype), true, true
       )+"*";
    break;
  case RECORD_TYPE:
    //cout << "Into RECORD_TYPE" << endl;
    // debug_tree(nodetype);
    cout << "==================" << endl;

    if(isPartial){
      result = manage_record_type(nodetype, true);
    }else{
      result = manage_record_type(nodetype, false);
    }
    break;
  default:
     break;
  }
   cout << endl;
   cout << "Into Manage types. result= "<< result << endl
      ;
   cout << endl;
  return result;
}
void write_external_functions(){
 for (list<tree_node*>::iterator it =
    list_functions_to_mock.begin(); it !=
    list_functions_to_mock.end(); ++it){
    list<string> list_type_args;
    list<tree_node*> list_node_args;
    cout << endl;
    print_node_declaration(*it);
    cout << endl;
    // debug_tree(*it);
    cout << endl;
    string function_name =  IDENTIFIER_POINTER(DECL_NAME
       (*it));
    cout << "Writing mock function: " << function_name <<
        endl;
    // cout << "Function name:" << function_name<< endl;
    //cout << "---GET THE RETURN TYPE---"<< endl;
    tree_node *tn =  reinterpret_cast<tree_node*>(*it);
    tree get_decl_result = DECL_RESULT(tn);
    string return_type =  manage_types(TREE_TYPE(
       TREE_TYPE(*it) ),false, true);
    myfile << "extern \"C\" "<<endl;
    myfile << return_type << " " << function_name << "(";
    cout << endl;
    //cout << "take arguments " << endl;
    tree node = TYPE_ARG_TYPES( TREE_TYPE(*it) );
    int count = 0;
    if(node!=NULL){
      do{
    list_type_args.push_back( manage_types(TREE_VALUE(
       node),false,true));//for each arguments: get the
       type argument and put it into the list_type_args
    list_node_args.push_back(node);
      }
      while(node = TREE_CHAIN(node) );
      cout << endl;
      //its necessary jump the last element that its
```

```
         always (a void type) not necessary
      for (list<string>::iterator x = list_type_args.
         begin(); count < list_type_args.size()-1 ; ++x,
         count++){

    myfile << *x << " param" << count;
    if(count != list_type_args.size()-2){
      myfile << ", ";
    }
      }
    }
    myfile << "){" << endl;
    myfile << " // check expectations " << endl;
    // myfile << "  struct Mock_" << function_name << "*
       obj  = list_"<<function_name << ".front();" <<
       endl;
    myfile << "  int ris;" << endl;
    myfile << "  void* return_value;" << endl;
    myfile << "  bool check_parameter = false;" << endl;
    count = 0;
    myfile << "  Mock_" << function_name  << " *obj ;"<<
       endl;
    myfile << "  for (list<Mock_"<< function_name  <<"
       *>::iterator it = list_"<<function_name <<".begin
       (); it!= list_"<< function_name  <<".end() &&  !
       check_parameter ; ++it){" << endl;
    myfile << "  check_parameter = true;" << endl;
    myfile << "  obj = *it;" << endl;

    for (list<tree_node*>::iterator x = list_node_args.
       begin();!(list_node_args.empty()) &&  count <
       list_node_args.size()-1 ; ++x, count++){



      myfile << "  switch( obj->switch"<< count <<" ){"<<
          endl;
      myfile << "    case ANY_VALUE: "<< endl;
      myfile << "      break; "<< endl;

      if(!(TREE_CODE(TREE_VALUE(*x)) == RECORD_TYPE)){
    myfile << "    case CHECK_PARAMS_VALUE: "<< endl;
    myfile << "      if( obj->param" << count << " !=
       param" << count <<" ){ "<< endl;
    //myfile << "         manage_errors(
       DIFFERENT_PARAMETER, \""<<function_name<<"\"); "<<
        endl;
```

```
myfile << "          check_parameter = false;" << endl
   ;
myfile << "       }"<< endl;
myfile << "       break; "<< endl;

myfile << "     case Gt: "<< endl;
myfile << "       if( obj->param" << count << " >=
   param" << count <<" ){ "<< endl;
//myfile << "           manage_errors(Gt_ERROR, \""<<
   function_name<<"\"); "<< endl;
myfile << "          check_parameter = false;" << endl
   ;
myfile << "       }"<< endl;
myfile << "       break; "<< endl;

myfile << "     case Lt: "<< endl;
myfile << "       if( obj->param" << count << " <=
   param" << count <<" ){ "<< endl;
//myfile << "           manage_errors(Lt_ERROR, \""<<
   function_name<<"\"); "<< endl;
myfile << "          check_parameter = false;" << endl
   ;
myfile << "       }"<< endl;
myfile << "       break; "<< endl;

myfile << "     case Ge: "<< endl;
myfile << "       if( obj->param" << count << " >
   param" << count <<" ){ "<< endl;
//myfile << "           manage_errors(Ge_ERROR, \""<<
   function_name<<"\"); "<< endl;
myfile << "          check_parameter = false;" << endl
   ;
myfile << "       }"<< endl;
myfile << "       break; "<< endl;

myfile << "     case Le: "<< endl;
myfile << "       if( obj->param" << count << " <
   param" << count <<" ){ "<< endl;
//myfile << "           manage_errors(Le_ERROR, \""<<
   function_name<<"\"); "<< endl;
myfile << "          check_parameter = false;" << endl
   ;
myfile << "       }"<< endl;
myfile << "       break; "<< endl;

myfile << "     case Ne: "<< endl;
myfile << "       if( obj->param" << count << " ==
   param" << count <<" ){ "<< endl;
//myfile << "          manage_errors(Ne_ERROR, \""<<
   function_name<<"\"); "<< endl;
myfile << "          check_parameter = false;" << endl
   ;
myfile << "       }"<< endl;
myfile << "       break; "<< endl;
  }

  myfile << "     case USER_FUNCTION: "<< endl;
  myfile << "       ris = obj->userfunc"<< count << "(
     param" << count << ");" << endl;
  //myfile << "       printf(\"******* ris = %d \\n
     \",ris);" << endl;
  myfile << "       if( ris !=0 ){ "<< endl;
  //myfile << "          manage_errors(
     DIFFERENT_PARAMETER, \""<<function_name<<"\");
     "<< endl;
  myfile << "          check_parameter = false;" <<
     endl;
  myfile << "       }"<< endl;
  myfile << "       break; "<< endl;

  string type(manage_types(TREE_VALUE(*x),false,true)
     );
  if(type.compare("char*") == 0){
myfile << "     case CHECK_PARAMS_STRING: "<< endl;
myfile << "       if( strcmp(obj->param" << count << "
   , param" << count <<")!=0 ){ "<< endl;
//myfile << "          manage_errors(
   DIFFERENT_PARAMETER, \""<<function_name<<"\"); "<<
    endl;
myfile << "          check_parameter = false;" << endl
   ;
myfile << "       }"<< endl;
myfile << "       break; "<< endl;
  }

  myfile << "     default: "<< endl;
  myfile << "       break; "<< endl;
  myfile << "   }//close switch "<< endl;
}
myfile << "}//close for" << endl;
 myfile << "  if(check_parameter == true){" << endl;
 myfile << "    return_value = check_expectations(obj
    ->list_expectations, \"" << function_name << "\")
    ;" << endl;
```

```
        myfile << "  }else{"<< endl;
        myfile <<"     manage_errors(DIFFERENT_PARAMETER,
           \""<<function_name<<"\");" << endl;
        myfile << "    return_value = NULL;" << endl;
        myfile << "}" << endl;
      if(return_type.compare("void")!=0 ){
         if(return_type.compare("void")!=0 ){
        myfile << "  if(return_value == NULL){return NULL
           ;} //Error in check_expectations" << endl;
      }

        myfile <<"  "<< return_type << "
           casting_return_value = *("<<return_type<<"*)
           return_value;" << endl;
     myfile <<"  return casting_return_value;" << endl;
      }
     myfile << endl;

     myfile << "}" << endl;
     myfile << endl;
   }
}

void  write_expectation_functions(){
 for (list<tree_node*>::iterator it =
     list_functions_to_mock.begin(); it !=
     list_functions_to_mock.end(); ++it){
     list<string> list_type_args;
     list<tree_node*> list_node_args;
     cout << endl;
     string function_name =  IDENTIFIER_POINTER(DECL_NAME
        (*it));
     cout << "Writing Expectation of the function: "<<
        function_name << endl;;

     tree_node *tn =  reinterpret_cast<tree_node*>(*it);
     tree get_decl_result = DECL_RESULT(tn);
     string return_type =  manage_types(TREE_TYPE(
        TREE_TYPE(*it) ), false, true);
     //myfile << return_type << " " << function_name <<
        "(";
     myfile << "extern \"C\"" << endl;
     myfile << "void expect_" << function_name << "(";

     tree node = TYPE_ARG_TYPES( TREE_TYPE(*it) );//get
        function's arguments
     int count = 0;
```

```
if(node != NULL){
  do{
list_type_args.push_back( manage_types(TREE_VALUE(
   node), false, true ) );//for each arguments: get
   the type argument and put it into the
   list_type_args
list_node_args.push_back( TREE_VALUE(node));
  }
  while(node = TREE_CHAIN(node) );

  //write on temporary_file.c the type arguments
  count = 0;
  //it is necessary jump the last element that its
     always (a void type) not necessary
  for (list<tree_node*>::iterator x = list_node_args.
     begin();count < list_node_args.size()-1 ; ++x,
     count++){

if( TREE_CODE(*x) == RECORD_TYPE ){
  myfile << "int switch"<< count << ", " <<
     manage_types(*x,false, true) << "* param" <<
     count << ", int (*userfunc"<<count<<")( "<<
     manage_types(*x, false, true) << " x)," << endl;

}else{
  myfile << "int switch"<< count << ", " <<
     manage_types(*x, false, true) << " param" <<
     count << ", int (*userfunc"<<count<<")("<<
     manage_types(*x,false, true)  <<" x)," << endl;
}
  }
}
myfile << "int startvararg, ...){" << endl;


myfile << "  printf(\"INTO: "<<function_name<<" \\n\"
    );" << endl;

string namestruct = "Mock_"+ function_name;
myfile << "  struct "<< namestruct<< " *obj = new "<<
    namestruct <<"();" << endl;
myfile << "  list_"<< function_name << ".push_front(
   obj);"<< endl;

for (count = 0; ( !list_type_args.empty() ) && count
   < list_type_args.size()-1 ;  count++){ //size()-1
   because we need always to skip the last argument.
```

```
       size()-2 because here I dont consider the "int
       startvararg" parameter.

 //myfile << " printf(\"param"<<count<<": %g, switch"<<
    count<<": %g  \\n \", param"<<count<< " , switch"<<
    count<<" );"<< endl;

      myfile << "  obj->switch" << count << "= switch"<<
         count<<";"<< endl;
      myfile << "  obj->param" << count << "= param"<<
         count<<";"<< endl;
      myfile << "  obj->userfunc" << count << "= userfunc
         "<< count<<";"<< endl;
      myfile << endl;
    }

    myfile << endl;
    myfile << "//Add Times Expectations" << endl;
    myfile << "  if(startvararg == NO_MORE_PARAMETERS ){
       // use the default setting" << endl;
    myfile << "   printf(\"no more parameters  \\n \");"
       << endl;
    myfile << "   return;" << endl;
    myfile << "  }" << endl;
    myfile << "  va_list ap; " << endl;
    myfile << "  va_start(ap, startvararg);" << endl;
    myfile << "  int time_value;" << endl;
    //  myfile << "  printf(\"startvararg: %d\\n \",
       startvararg);" << endl;
    myfile << "  if(startvararg == TIMES || startvararg
       == TIMES_AT_LEAST || startvararg == TIMES_AT_MOST)
       {" << endl;
    myfile << "    time_value = va_arg(ap,int);" << endl;

    myfile << "    struct Expectation* exp = new
       Expectation(); "<< endl;
    myfile << "    exp->type_expectation = startvararg; "
        << endl;
    myfile << "    int* p = (int*)malloc(sizeof(int)); "
       << endl;
    myfile << "    *p = time_value;" << endl;
    myfile << "    exp->returnvalue = (void*)p;" << endl;
    myfile << "    obj->list_expectations.push_back(exp);
       " << endl;
    myfile << "  }else{ " << endl;
    myfile << "    time_value = va_arg(ap,int); // to
       jump the \"time_value parameter\" used from the
```

```
   TIMES macro " << endl;
myfile << "  }" << endl;
myfile << "  //Manage WILLONCE and REPEADETLY" <<
   endl;

if(return_type.compare("void")!=0 ){  // you do not
   need to manage return type for function that
   return void.


  myfile << "  int type_expectation; // the type of
     the variable is as the type return function" <<
     endl;
  myfile << "  " << return_type << " return_value;"
     << endl;
  myfile << "  for(type_expectation = va_arg(ap, int)
     ; type_expectation != NO_MORE_PARAMETERS;
     type_expectation = va_arg(ap,int) ){" << endl;
  myfile << "    struct Expectation* exp = new
     Expectation(); "<< endl;
  myfile << "    return_value = va_arg(ap, " <<
     return_type << ");// the return type is always
     as the return type of the orginal function. We
     are treatting just WILLONCE and REPEADETLY " <<
     endl;
  myfile << "    "<< return_type << "* p = ("<<
     return_type<<"*)malloc(sizeof("<<return_type<<")
     );" << endl;
  myfile << "    *p = return_value;" << endl;
  myfile << "    exp->type_expectation =
     type_expectation; " << endl;
  myfile << "    exp->returnvalue = (void*)p; " <<
     endl;
  //  myfile << "    printf(\"type_expectation: %d
     \\n\", type_expectation);  " << endl;
  //myfile << "    printf(\"return value: %d \\n\",
     return_value);//occhio che non e sempre di tipo
     intero  " << endl;
  myfile << "    obj->list_expectations.push_back(exp
     );" << endl;
  myfile << "  }" << endl;
}
myfile << "  va_end(ap);" << endl;


myfile << "}" << endl;
myfile << endl;
```

```
  }


}




void write_structs(){

 for (list<tree_node*>::iterator it =
     list_functions_to_mock.begin(); it !=
     list_functions_to_mock.end(); ++it){
    list<string> list_type_args;
    list<tree_node*> list_node_args;

    cout << endl;
    string function_name =  IDENTIFIER_POINTER(DECL_NAME(*
       it));
    tree_node *tn =  reinterpret_cast<tree_node*>(*it);
    tree get_decl_result = DECL_RESULT(tn);
    // string return_type =  manage_types(TREE_TYPE(
       TREE_TYPE(*it) ), false);
    //myfile << return_type << " " << function_name <<
       "(";

    cout << "Writing the struct for " << function_name <<
       endl;
    myfile << "struct Mock_" << function_name << "{"<<
       endl;

    //  debug_tree(TREE_TYPE(*it));
    tree node = TYPE_ARG_TYPES( TREE_TYPE(*it) );//get
       function's arguments
    if(node != NULL){ //if node is null the function does
       NOT have arguments
      do{
        list_type_args.push_back( manage_types(TREE_VALUE(
           node), false, true ) );//for each arguments:
           get the type argument and put it into the
           list_type_args
        list_node_args.push_back( TREE_VALUE(node) );
      }while(node = TREE_CHAIN(node) );

      //write on temporary_file.c the type arguments
      int count =0;
      //it is necessary jump the last element that its
```

```
        always (a void type) not necessary

     for (list<tree_node*>::iterator x = list_node_args.
        begin();  count < list_node_args.size()-1 ; ++x,
        count++){
       if(TREE_CODE(*x) == RECORD_TYPE){
     myfile << "  int switch"<< count << "; " <<
        manage_types(*x,false, true) << "* param" <<
        count << ";" << " int (*userfunc" << count << ")(
        "<< manage_types(*x, false, true) << " x);"  <<
        endl;
       }else{
     myfile << "  int switch"<< count << ";" <<"   " <<
        manage_types(*x,false, true) << " param" << count
         << ";" << " int (*userfunc" << count << ")("<<
        manage_types(*x,false, true) << " x);"  << endl;
       }
     }
   }
   myfile << "  list<Expectation*> list_expectations;" <<
       endl;
   myfile << "};" << endl;
   myfile << "list<Mock_"<< function_name << "*> list_"
      << function_name << ";"<< endl;
   // string up_function_name = function_name;
   //transform(up_function_name.begin(), up_function_name
      .end(), up_function_name.begin(), toupper);
   //myfile << "#define EXPECT_" << function_name <<
      "(...) expect_" << function_name << "( __VA_ARGS__,
       NO_MORE_PARAMETERS)" << endl;
   myfile << endl;
 }
}


void write_header(){

  headerfile.open("temporary_file.h");
  headerfile << "#ifndef TEMPORARYFILE_H"<< endl;
  headerfile << "#define TEMPORARYFILE_H" << endl;
  headerfile << "#include \"user_headers.h\"" << endl;
  //headerfile << "#include \"temporary_user_structs.h
     \""<< endl;
  for (list<tree_node*>::iterator it =
     list_functions_to_mock.begin(); it !=
     list_functions_to_mock.end(); ++it){
    list<string> list_type_args;
    list<tree_node*> list_node_args;

    cout << endl;
    string function_name =  IDENTIFIER_POINTER(DECL_NAME
       (*it));
    cout << "Writing header for the function: " <<
       function_name << endl;
    tree_node *tn =  reinterpret_cast<tree_node*>(*it);
    tree get_decl_result = DECL_RESULT(tn);

    headerfile << "#define EXPECT_" << function_name << "
       (...) expect_" << function_name << "( __VA_ARGS__,
        NO_MORE_PARAMETERS)" << endl;
    headerfile << "void expect_" << function_name << "(";
    tree node = TYPE_ARG_TYPES( TREE_TYPE(*it) );//get
       function's arguments
    if(node!=NULL){
      do{
    list_type_args.push_back( manage_types(TREE_VALUE(
       node),false, true ));//for each arguments: get the
        type argument and put it into the list_type_args
    list_node_args.push_back(TREE_VALUE(node));
      }
      while(node = TREE_CHAIN(node) );

      //write on temporary_file.c the type arguments
      int count =0;
      //it is necessary jump the last element that its
         always (a void type) not necessary
      for (list<tree_node*>::iterator x = list_node_args.
         begin(); count < list_node_args.size()-1 ; ++x,
         count++){

    if( (TREE_CODE(*x) == RECORD_TYPE) ||(TREE_CODE(*x)
       == ENUMERAL_TYPE)){
      headerfile << "int switch" << count << ", " <<
         manage_types(*x, false, true) << "* param" <<
         count << ", int (*userfunc"<<count<<")(" <<
         manage_types(*x, false, true) <<" x)";
    }else{
      headerfile << "int switch" << count << ",  " <<
         manage_types(*x, false, true) << " param" <<
         count << ", int (*userfunc"<<count<<")("<<
         manage_types(*x,false, true)  <<" x)";
    }

    if(count != list_node_args.size()-2){
      headerfile << ", ";
    }
      }
      headerfile << ", int startvararg, ...); " << endl;
    }else{
      headerfile << "int startvararg, ...); " << endl;
    }
  }
    headerfile << endl;
    headerfile << "#define TEST_FINISHED testCompleted()"
        << endl;
    headerfile << "void testCompleted();" << endl;
    headerfile << "#endif" << endl;
    headerfile.close();
}



void write_check_testCompleted_function(){
  myfile << endl;

  myfile << "//When the test is finished we have to see
     if all the expectations have been respected or not."
      << endl;
  myfile << "extern \"C\" " << endl;
  myfile << "void testCompleted(){"<<endl;
  myfile << "  int expectation_number = 1 ;"<< endl;
  myfile << endl;
  for (list<tree_node*>::iterator it =
     list_functions_to_mock.begin(); it !=
     list_functions_to_mock.end(); ++it){
    string function_name =  IDENTIFIER_POINTER(DECL_NAME
       (*it));
    myfile << "  for ( list<Mock_" << function_name << "
       *>::iterator it = list_" << function_name << ".
       begin(); it!= list_" << function_name <<".end() ;
       expectation_number++, ++it){" << endl;


    myfile << "    struct Mock_"<< function_name <<"*
       obj_"<<function_name <<" = *it;"<< endl;
    myfile << "    std::stringstream ss ;" << endl;
    myfile << "    ss << " << "\"" << function_name << "
       \"" <<" << \" from Expectation number: \"<<
       expectation_number; " << endl;
    myfile << "    check_expectations_test_finished(obj_"
       << function_name <<"->list_expectations , ss.str()
```

```
        );" << endl;
    myfile << "  }" << endl;
    myfile << endl;
    myfile <<"   expectation_number = 1;" << endl;
  }
  myfile << "}"<< endl;
}

void write_function_on_file(){

  cout << "-- Write function calls to mock into the
     temporary file: temporary_file.c "<< endl;
  myfile.open("temporary_file.cpp");
  myfile << "#include \"helper.h\" "<< endl;
  myfile << "extern \"C\" {"<< endl;
  myfile << "  #include <stdarg.h>" << endl;
  myfile << "#include <string.h>//strcmp()" << endl;
  // myfile << "#include \"temporary_file.h\""<< endl;
  myfile << "#include <stdio.h>"<< endl;
  myfile << "#include <stdlib.h>"<< endl;
  //myfile << "#include \"helper.h\""<< endl;
  myfile << "}"<< endl;
  myfile << "#include <sstream>" << endl;
  myfile << "#include <list>"<< endl;
  myfile << "#include <iostream>" << endl;
  //  myfile << "#include \"temporary_user_structs.h\""
     << endl;
  myfile << "#include \"user_headers.h\"" << endl;
  myfile << endl;

  myfile << "using namespace std;" << endl;
  //============

  cout << "debug 1 " << endl;
  write_structs();
  myfile << endl;
  cout << "debug 2 " << endl;
  write_expectation_functions();
  cout << "debug 3 " << endl;

  write_external_functions();
  cout << "debug 4" << endl;
  write_check_testCompleted_function();
  cout << "debug 5" << endl;
  myfile.close();
}
```

```
//Function to re-built all the user data definition. Not
   used
void write_user_structs(){
  cout << "+++ Into write_user_structs +++" << endl;
  userStructDefined_file.open("temporary_user_structs.h")
     ;
  list<string> list_struct;

  for (list<tree_node*>::iterator it =
     list_functions_to_mock.begin(); it !=
     list_functions_to_mock.end(); ++it){

    string function_name =  IDENTIFIER_POINTER(DECL_NAME
       (*it));
    cout << "Checking if there are user-structs as
       parameters into the function: " << function_name<<
        endl;
    tree_node *tn =  reinterpret_cast<tree_node*>(*it);
    tree get_decl_result = DECL_RESULT(tn);

    tree node = TYPE_ARG_TYPES( TREE_TYPE(*it) );//get
       function's arguments
    if(node!=NULL){
      do{
    if(( TREE_CODE(  TREE_VALUE(node)) == ENUMERAL_TYPE))
       {
       bool isPresent = false;
      for(list<string>::iterator user_enum_it =
         list_struct.begin(); user_enum_it != list_struct
         .end() ; user_enum_it++){

        if( (*user_enum_it).compare( manage_enumeral_type
           (TREE_VALUE(node),true )   ) == 0 ){
          isPresent = true;
        }

      }
      if(!isPresent){//write the struct in the user
         structs into the file "temporary_user_structs.h"
        list_struct.push_back(manage_enumeral_type(
           TREE_VALUE(node), true) );
        userStructDefined_file  <<  manage_enumeral_type(
           TREE_VALUE(node), false);
      }
    }

    if ( (TREE_CODE(  TREE_VALUE(node)) == RECORD_TYPE) )
```

```
        { //check if the parameter is a user struct
       bool isPresent = false;
       for(list<string>::iterator userstruct_it =
          list_struct.begin(); userstruct_it !=
          list_struct.end() ; userstruct_it++){ //check if
           I have already written the same struct

         if( (*userstruct_it).compare( manage_record_type(
            TREE_VALUE(node),true )  ) == 0 ){
           isPresent = true;
         }

       }
       if(!isPresent){//write the struct in the user
          structs into the file "temporary_user_structs.h"
         list_struct.push_back(manage_record_type(
            TREE_VALUE(node), true) );
         userStructDefined_file  << "typedef " <<
            manage_record_type(TREE_VALUE(node), false);
       }
     }
       }
       while(node = TREE_CHAIN(node) );
     }
}
  userStructDefined_file.close();

}

void find_functions_to_mock(){
  cout << "##### find functions to mock ###### " << endl;
  for (list<tree_node*>::iterator it =
     list_functions_with_only_declaration.begin(); it !=
     list_functions_with_only_declaration.end(); ++it){
    tree id = DECL_NAME(*it);
    string name_function_with_onyl_declaration (
       IDENTIFIER_POINTER (id));
    bool isPresent = false;
    cout << "function name with: " <<
       name_function_with_onyl_declaration << endl;
    for (list<tree_node*>::iterator it2 =
       list_functions_with_body.begin(); (!isPresent) &&
       it2 != list_functions_with_body.end(); ++it2){
      tree id2 = DECL_NAME(*it2);
      string name_function_with_body (IDENTIFIER_POINTER
         (id2));
      cout << "function name: " <<
```

```
         name_function_with_onyl_declaration << endl;
      if(name_function_with_body.compare(
         name_function_with_onyl_declaration) == 0){
    isPresent = true;
      }
    }
    if(!isPresent){
      list_functions_to_mock.push_back(*it);
    }
  }
}

void cb_plugin_finish(void *gcc_data, void *user_data){
  cout << "****** EVENT PLUGIN_FINISH *******" << endl;
  find_functions_to_mock();
  cout<<"Print Functions to mock" <<endl;
  print_list(list_functions_to_mock);
  cout <<endl;
  //write_user_structs();
  write_function_on_file();
  write_header();
  cout << " *** END PLUGIN **** " << endl;
}


int plugin_init (plugin_name_args* info,
             plugin_gcc_version*)
{
  int r (0);

  cerr << "starting " << info->base_name << endl;
  //
  // Parse options if any.
  //

  // Disable assembly output.
  //
  // asm_file_name = HOST_BIT_BUCKET;

  // Register callbacks.
  register_callback(info->base_name,
            PLUGIN_FINISH_DECL,
            &cb_finish_decl,
            0);

  register_callback(info->base_name,
            PLUGIN_PRE_GENERICIZE,
            &cb_plugin_pre_generize,
            0);

  register_callback(info->base_name,
            PLUGIN_FINISH,
            &cb_plugin_finish,
            0);

  return r;
}
```

.1.3 *helper.cpp*

Listing 3. *helper.cpp.*

```
#include "helper.h"

//using namespace std;
void manage_errors(int  error_type, string function_name)
     {
    cout << endl;
    cout << endl;
    cout << "=== TEST FAIL ===" << endl;
    switch(error_type){
    case DIFFERENT_PARAMETER :
          cout << "From "<< function_name << endl;
          cout << "Parameters do not matching "<< endl;
          break;

    case  TIMES_CALLS_EXCEEDED:
          cout << "From "<< function_name << endl;
          cout << "Time calls exceeded! "<< endl;
          break;

    case Gt_ERROR:
          cout << "From "<< function_name << endl;
          cout << "The value received from the function called
                 is not GREATER of one set in the expectation "<<
                 endl;
          break;

    case  Lt_ERROR:
          cout << "From "<< function_name << endl;
          cout << "The value received from the function called
                 is not LESS of one set in the expectation "<<
               endl;
          break;
```

```
  case  Le_ERROR:
     cout << "From "<< function_name << endl;
     cout << "The value received from the function called
         is not LESS or EQUAL of one set in the
        expectation "<< endl;
     break;

  case  Ge_ERROR:
     cout << "From "<< function_name << endl;
     cout << "The value received from the function called
         is not GREATER or EQUAL of one set in the
        expectation "<< endl;
     break;

  case  Ne_ERROR:
     cout << "From "<< function_name << endl;
     cout << "The value received from the function called
         is not EQUAL of one set in the expectation "<<
        endl;
     break;

  default:
    break;
  }
  cout << "=================" << endl;
  cout << endl;
}

void manage_times_errors(int  error_type, string
   function_name, int total_calls, int made_calls){
  cout << "=== TEST FAIL ===" << endl;
  switch(error_type){

    case TIMES_CALLS_LESSER:
      cout << "The function \""<< function_name << "\"
         should be called AT LEAST " << total_calls;
      cout << (total_calls > 1? " times" : " time") <<
         endl;
      cout << "but it is called just " << made_calls << (
         made_calls > 1? " times":" time") << endl;
      break;

    case TIMES_CALLS_NEQUAL:
      cout << "The function \""<< function_name << "\"
         should be called EXACTLY " << total_calls;
      cout << (total_calls > 1? " times" : " time") <<
         endl;
      cout << "but it is called just " << made_calls << (
         made_calls > 1? " times":" time") << endl;
      break;

    case TIMES_CALLS_MORE:
      cout << "The function \""<< function_name << "\"
         should be called AT MOST " << total_calls;
      cout << (total_calls > 1? " times" : " time") <<
         endl;
      cout << "but it is called just " << made_calls << (
         made_calls > 1? " times":" time") << endl;
      break;

  default:
    break;
  }
  cout << "=================" << endl;
}
void check_expectations_test_finished(list<Expectation*>&
    list_exp, string function_name){

  //The first expectation it has to be always TIMES or
     TIME_AT_MOST or TIME_AT_LEAST
  list<Expectation*>::iterator it_exp = list_exp.begin();
  Expectation* exp_times = *it_exp;
  int total_number_calls = *(int*)exp_times->returnvalue;

//we need to check just if the number calls made are less
    then the total_number_calls setted in the expectation
    because the check for the "greater" case is done
   directly in the "check_expectations" function.

  switch(exp_times->type_expectation){
  case TIMES:
    if( exp_times->number_calls != total_number_calls ){
      manage_times_errors(TIMES_CALLS_NEQUAL ,
         function_name, total_number_calls, exp_times->
         number_calls );
    }
    break;

  case TIMES_AT_LEAST:
    if( exp_times->number_calls < total_number_calls ){
      manage_times_errors(TIMES_CALLS_LESSER ,
         function_name, total_number_calls, exp_times->
         number_calls );
    }
  break;

  case TIMES_AT_MOST:
    if( exp_times->number_calls > total_number_calls ){
      manage_times_errors(TIMES_CALLS_MORE ,
         function_name, total_number_calls, exp_times->
         number_calls );
    }
  default:
    break;
  }
  return;
}

void* check_expectations(list<Expectation*>& list_exp,
   string function_name){
  if( list_exp.size() == 0 ){
    return NULL;
  }
  //The first expectation it has to be always TIMES,
     TIMES_AT_LEAST or TIMES_AT_MOST
  list<Expectation*>::iterator it_exp = list_exp.begin();
  Expectation* exp_times = *it_exp;
  int total_number_calls = *(int*)exp_times->returnvalue;
  ++it_exp;
  exp_times->number_calls ++;
  if(exp_times->number_calls > total_number_calls){
    manage_errors( TIMES_CALLS_EXCEEDED, function_name);
    return NULL;
  }
  // a mock function corrisponding at a SUT eternal
     function that return void will not flow this loop.
  for (int x = 1; it_exp != list_exp.end(); ++it_exp, x
     ++){
    switch((*it_exp)->type_expectation){
      case WILLONCE:
    if(x == exp_times->number_calls){
      return (*it_exp)->returnvalue;
    }
    break;

      case WILL_REPEATEDLY:
        return (*it_exp)->returnvalue;
    break;
    }
  }
  return NULL;
```

```
}
```

.1.4 *helper.h*

Listing 4. *helper.h.*

```
#ifndef HELPER_H
#define HELPER_H

#include <iostream>
#include <list>
#include <string>

using namespace std;

struct Expectation {
  int type_expectation; //TIMES, WILLONCE, REPEADETLY
  void* returnvalue;   //for TIMES it contains the number
      of times that the function should be called. For
     WILLONCE and REAPEADETLY contain the return value.
  int number_calls ; // TIMES use to keep trace of the
     number of times that we call the corrispondent
     function.
  Expectation(){
    number_calls = 0;
  }
};

//VALUE(x)
#define CHECK_PARAMS_VALUE  0
#define ANY_VALUE           1
//STRING(x)
#define CHECK_PARAMS_STRING 2
/*    USER_FUNCTION(x)   x is the address of the function
    to use. Return 0 if the test is positive. Otherwise
   the test will fail.
*/
//USER_FUNCTION(x)
#define USER_FUNCTION       3

#define Gt                  4
#define Lt                  5
#define Ge                  6
#define Le                  7
#define Ne                  8
```

```
// ===== End Generic Comparison =======

// ===== Errors ===============

#define DIFFERENT_PARAMETER       100
#define TIMES_CALLS_EXCEEDED      101
#define Gt_ERROR                  102
#define Lt_ERROR                  103
#define Ge_ERROR                  104
#define Le_ERROR                  105
#define Ne_ERROR                  106
#define TIMES_CALLS_LESSER        107
#define TIMES_CALLS_NEQUAL        108
#define TIMES_CALLS_MORE          109

// ===== End Errors ============

/*========== Expectations =====================
    .With(multi_argument_matcher)                 No
    .Times(cardinality)                           Yes
    .InSequence(sequences)                        No
    .After(expectations)                          No
    .WillOnce(action)                             Yes
    .WillRepeatedly(action)                       Yes
    .RetiresOnSaturation();                       No
 */
#define TIMES             200
#define WILLONCE          201
#define WILL_REPEATEDLY   202
#define TIMES_AT_LEAST     203
#define TIMES_AT_MOST      204

#define NO_MORE_PARAMETERS -256


// ====== END Actions definitions ===================


void* check_expectations(list<Expectation*>& list_exp,
   string function_name);
void check_expectations_test_finished(list<Expectation*>&
    list_exp, string function_name);



using namespace std;
void manage_errors(int  error_type, string function_name)
```

```
   ;

#endif
```

.1.5 *framework.h*

Listing 5. *framework.h.*

```
#ifndef MYFRAMEWORK_H
#define MYFRAMEWORK_H



#define VALUE(x)         0, x, NULL
#define ANY_VALUE        1, 0, NULL

// STRING(x) manage char* types
#define STRING(x)        2, x, NULL

//USER_FUNCTION(x) x: (*function)() The function return 0
    if the test is postive. Otherwise the test will fail.
#define USER_FUNCTION(x) 3, NULL, x

//y is the parameter got from the mocked function call
#define Gt(x)            4, x, NULL    //  x > y
#define Lt(x)            5, x, NULL    //  x < y
#define Ge(x)            6, x, NULL    //  x >= y
#define Le(x)            7, x, NULL    //  x <= y
#define Ne(x)            8, x, NULL    //  x != y

#define TIMES(x)           200, x
#define WILL_ONCE(x)        201, x
#define WILL_REPEATEDLY(x) 202, x
#define RETURN(x)          x

#define TIMES_AT_LEAST(x)      203, x
#define TIMES_AT_MOST(x)       204, x


#define NO_MORE_PARAMETERS -256


#endif
```

.1.6 *order.c*

Listing 6. *order.c.*

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

```c
#include "warehouse.h"

int is_filled;
void create_order(){
  is_filled = 0;
}
void fill(char* name, int quantity){//function to test
  if( warehouse_has_inventory(name, quantity) == 0 ){
    warehouse_remove(name, quantity);
    is_filled = 1;
  }
}
int isFilled(){
  return is_filled;
}
```

.1.7 *warehouse.h*

Listing 7. *warehouse.h.*

```c
int warehouse_has_inventory(char* name, int quantity);
void warehouse_remove(char* name, int quantity);
```

.1.8 *test_order.c*

Listing 8. *test_order.c.*

```c
#include <stdio.h>

#include "myframework.h"

//#include "temporary_user_structs.h"
//#include "system_under_test.h"
#include "temporary_file.h" //import MACROs for the
   expectation functions


void test_order_fillingRemovesInventoryIfInStock(){

  EXPECT_warehouse_remove(STRING("Talisker"),
                        VALUE(50),
                        TIMES(1)
                       );

  EXPECT_warehouse_has_inventory(STRING("Talisker"),
                       VALUE(50),
                       TIMES(1),
                       WILL_ONCE(RETURN(0) )
                      );
  fill("Talisker", 50); //Execution: Call the function to
      test!

}
int main(int argc, char **argv)
{

  test_order_fillingRemovesInventoryIfInStock();
  TEST_FINISHED;

  if(isFilled() != 1){
    printf("Test falited\n");
  }else{
    printf("Test PASSED\n");
  }
  return 1;
}
```

.1.9 *temporary_file.cpp*

Listing 9. *temporary_file.cpp.*

```
#include "helper.h"
extern "C" {
  #include <stdarg.h>
#include <string.h>//strcmp()
#include <stdio.h>
#include <stdlib.h>
}
#include <sstream>
#include <list>
#include <iostream>
#include "user_headers.h"

using namespace std;
struct Mock_warehouse_has_inventory{
  int switch0;  char* param0; int (*userfunc0)(char* x);
  int switch1;  int param1; int (*userfunc1)(int x);
  list<Expectation*> list_expectations;
};
list<Mock_warehouse_has_inventory*>
   list_warehouse_has_inventory;

struct Mock_warehouse_remove{
  int switch0;  char* param0; int (*userfunc0)(char* x);
  int switch1;  int param1; int (*userfunc1)(int x);
  list<Expectation*> list_expectations;
```

```
};
list<Mock_warehouse_remove*> list_warehouse_remove;


extern "C"
void expect_warehouse_has_inventory(int switch0, char*
   param0, int (*userfunc0)(char* x),
int switch1, int param1, int (*userfunc1)(int x),
int startvararg, ...){
  printf("INTO: warehouse_has_inventory \n" );
  struct Mock_warehouse_has_inventory *obj = new
     Mock_warehouse_has_inventory();
  list_warehouse_has_inventory.push_front(obj);
  obj->switch0= switch0;
  obj->param0= param0;
  obj->userfunc0= userfunc0;

  obj->switch1= switch1;
  obj->param1= param1;
  obj->userfunc1= userfunc1;


//Add Times Expectations
  if(startvararg == NO_MORE_PARAMETERS ){ // use the
     default setting
   printf("no more parameters  \n ");
   return;
  }
  va_list ap;
  va_start(ap, startvararg);
  int time_value;
  if(startvararg == TIMES || startvararg ==
     TIMES_AT_LEAST || startvararg == TIMES_AT_MOST){
    time_value = va_arg(ap,int);
    struct Expectation* exp = new Expectation();
    exp->type_expectation = startvararg;
    int* p = (int*)malloc(sizeof(int));
    *p = time_value;
    exp->returnvalue = (void*)p;
    obj->list_expectations.push_back(exp);
  }else{
    time_value = va_arg(ap,int); // to jump the "
       time_value parameter" used from the TIMES macro
  }
  //Manage WILLONCE and REPEADETLY
  int type_expectation; // the type of the variable is as
     the type return function
  int return_value;
  for(type_expectation = va_arg(ap, int);
     type_expectation != NO_MORE_PARAMETERS;
     type_expectation = va_arg(ap,int) ){
    struct Expectation* exp = new Expectation();
    return_value = va_arg(ap, int);// the return type is
       always as the return type of the orginal function.
        We are treatting just WILLONCE and REPEADETLY
    int* p = (int*)malloc(sizeof(int));
    *p = return_value;
    exp->type_expectation = type_expectation;
    exp->returnvalue = (void*)p;
    obj->list_expectations.push_back(exp);
  }
  va_end(ap);
}

extern "C"
void expect_warehouse_remove(int switch0, char* param0,
   int (*userfunc0)(char* x),
int switch1, int param1, int (*userfunc1)(int x),
int startvararg, ...){
  printf("INTO: warehouse_remove \n" );
  struct Mock_warehouse_remove *obj = new
     Mock_warehouse_remove();
  list_warehouse_remove.push_front(obj);
  obj->switch0= switch0;
  obj->param0= param0;
  obj->userfunc0= userfunc0;

  obj->switch1= switch1;
  obj->param1= param1;
  obj->userfunc1= userfunc1;


//Add Times Expectations
  if(startvararg == NO_MORE_PARAMETERS ){ // use the
     default setting
   printf("no more parameters  \n ");
   return;
  }
  va_list ap;
  va_start(ap, startvararg);
  int time_value;
  if(startvararg == TIMES || startvararg ==
     TIMES_AT_LEAST || startvararg == TIMES_AT_MOST){
    time_value = va_arg(ap,int);
    struct Expectation* exp = new Expectation();
    exp->type_expectation = startvararg;
    int* p = (int*)malloc(sizeof(int));
    *p = time_value;
    exp->returnvalue = (void*)p;
    obj->list_expectations.push_back(exp);
  }else{
    time_value = va_arg(ap,int); // to jump the "
       time_value parameter" used from the TIMES macro
  }
  //Manage WILLONCE and REPEADETLY
  va_end(ap);
}

extern "C"
int warehouse_has_inventory(char* param0, int param1){
 // check expectations
  int ris;
  void* return_value;
  bool check_parameter = false;
  Mock_warehouse_has_inventory *obj ;
  for (list<Mock_warehouse_has_inventory*>::iterator it =
      list_warehouse_has_inventory.begin(); it!=
     list_warehouse_has_inventory.end() &&  !
     check_parameter ; ++it){
  check_parameter = true;
  obj = *it;
  switch( obj->switch0 ){
    case ANY_VALUE:
      break;
    case CHECK_PARAMS_VALUE:
      if( obj->param0 != param0 ){
         check_parameter = false;
      }
      break;
    case Gt:
      if( obj->param0 >= param0 ){
         check_parameter = false;
      }
      break;
    case Lt:
      if( obj->param0 <= param0 ){
         check_parameter = false;
      }
      break;
    case Ge:
      if( obj->param0 > param0 ){
        check_parameter = false;
      }
      break;
    case Le:
      if( obj->param0 < param0 ){
        check_parameter = false;
      }
      break;
    case Ne:
      if( obj->param0 == param0 ){
        check_parameter = false;
      }
      break;
    case USER_FUNCTION:
      ris = obj->userfunc0(param0);
      if( ris !=0 ){
        check_parameter = false;
      }
      break;
    case CHECK_PARAMS_STRING:
      if( strcmp(obj->param0, param0)!=0 ){
        check_parameter = false;
      }
      break;
    default:
      break;
  }//close switch
  switch( obj->switch1 ){
    case ANY_VALUE:
      break;
    case CHECK_PARAMS_VALUE:
      if( obj->param1 != param1 ){
        check_parameter = false;
      }
      break;
    case Gt:
      if( obj->param1 >= param1 ){
        check_parameter = false;
      }
      break;
    case Lt:
      if( obj->param1 <= param1 ){
        check_parameter = false;
      }
      break;
    case Ge:
      if( obj->param1 > param1 ){
         check_parameter = false;
      }
      break;
    case Le:
      if( obj->param1 < param1 ){
         check_parameter = false;
      }
      break;
    case Ne:
      if( obj->param1 == param1 ){
         check_parameter = false;
      }
      break;
    case USER_FUNCTION:
      ris = obj->userfunc1(param1);
      if( ris !=0 ){
         check_parameter = false;
      }
      break;
    default:
      break;
  }//close switch
}//close for
  if(check_parameter == true){
    return_value = check_expectations(obj->
       list_expectations, "warehouse_has_inventory");
  }else{
     manage_errors(DIFFERENT_PARAMETER, "
        warehouse_has_inventory");
    return_value = NULL;
}
  if(return_value == NULL){return NULL;} //Error in
     check_expectations
  int casting_return_value = *(int*) return_value;
  return casting_return_value;

}

extern "C"
void warehouse_remove(char* param0, int param1){
 // check expectations
  int ris;
  void* return_value;
  bool check_parameter = false;
  Mock_warehouse_remove *obj ;
  for (list<Mock_warehouse_remove*>::iterator it =
     list_warehouse_remove.begin(); it!=
     list_warehouse_remove.end() &&  !check_parameter ;
     ++it){
  check_parameter = true;
  obj = *it;
  switch( obj->switch0 ){
    case ANY_VALUE:
      break;
    case CHECK_PARAMS_VALUE:
      if( obj->param0 != param0 ){
         check_parameter = false;
      }
      break;
    case Gt:
      if( obj->param0 >= param0 ){
         check_parameter = false;
      }
      break;
    case Lt:
      if( obj->param0 <= param0 ){
         check_parameter = false;
      }
      break;
    case Ge:
      if( obj->param0 > param0 ){
         check_parameter = false;
      }
      break;
    case Le:
      if( obj->param0 < param0 ){
         check_parameter = false;
      }
      break;
    case Ne:
      if( obj->param0 == param0 ){
         check_parameter = false;
      }
      break;
    case USER_FUNCTION:
      ris = obj->userfunc0(param0);
      if( ris !=0 ){
         check_parameter = false;
      }
      break;
    case CHECK_PARAMS_STRING:
      if( strcmp(obj->param0, param0)!=0 ){
         check_parameter = false;
      }
```

```
        break;
      default:
        break;
    }//close switch
    switch( obj->switch1 ){
      case ANY_VALUE:
        break;
      case CHECK_PARAMS_VALUE:
        if( obj->param1 != param1 ){
           check_parameter = false;
        }
        break;
      case Gt:
        if( obj->param1 >= param1 ){
           check_parameter = false;
        }
        break;
      case Lt:
        if( obj->param1 <= param1 ){
           check_parameter = false;
        }
        break;
      case Ge:
        if( obj->param1 > param1 ){
           check_parameter = false;
        }
        break;
      case Le:
        if( obj->param1 < param1 ){
           check_parameter = false;
        }
        break;
      case Ne:
        if( obj->param1 == param1 ){
           check_parameter = false;
        }
        break;
      case USER_FUNCTION:
        ris = obj->userfunc1(param1);
        if( ris !=0 ){
           check_parameter = false;
        }
        break;
      default:
        break;
    }//close switch
  }//close for
    if(check_parameter == true){
      return_value = check_expectations(obj->
         list_expectations, "warehouse_remove");
    }else{
       manage_errors(DIFFERENT_PARAMETER, "warehouse_remove
          ");
      return_value = NULL;
  }

  }


  //When the test is finished we have to see if all the
     expectations have been respected or not.
  extern "C"
  void testCompleted(){
    int expectation_number = 1 ;

    for ( list<Mock_warehouse_has_inventory*>::iterator it
       = list_warehouse_has_inventory.begin(); it!=
       list_warehouse_has_inventory.end() ;
       expectation_number++, ++it){
      struct Mock_warehouse_has_inventory*
         obj_warehouse_has_inventory = *it;
      std::stringstream ss ;
      ss << "warehouse_has_inventory" << " from Expectation
          number: "<< expectation_number;
      check_expectations_test_finished(
         obj_warehouse_has_inventory->list_expectations ,
         ss.str() );
    }

     expectation_number = 1;
    for ( list<Mock_warehouse_remove*>::iterator it =
       list_warehouse_remove.begin(); it!=
       list_warehouse_remove.end() ;expectation_number++,
       ++it){
      struct Mock_warehouse_remove* obj_warehouse_remove =
         *it;
      std::stringstream ss ;
      ss << "warehouse_remove" << " from Expectation number
         : "<< expectation_number;
      check_expectations_test_finished(obj_warehouse_remove
         ->list_expectations , ss.str() );
    }

     expectation_number = 1;
```

```
}
```

.1.10 *temporary_file.h*

Listing 10. *temporary_file.h.*

```
#ifndef TEMPORARYFILE_H
#define TEMPORARYFILE_H
#include "user_headers.h"
#define EXPECT_warehouse_has_inventory(...)
   expect_warehouse_has_inventory( __VA_ARGS__,
   NO_MORE_PARAMETERS)
void expect_warehouse_has_inventory(int switch0,  char*
   param0, int (*userfunc0)(char* x), int switch1,  int
   param1, int (*userfunc1)(int x), int startvararg, ...)
   ;
#define EXPECT_warehouse_remove(...)
   expect_warehouse_remove( __VA_ARGS__,
   NO_MORE_PARAMETERS)
void expect_warehouse_remove(int switch0,  char* param0,
   int (*userfunc0)(char* x), int switch1,  int param1,
   int (*userfunc1)(int x), int startvararg, ...);

#define TEST_FINISHED testCompleted()
void testCompleted();
#endif
```