Unit test research

Project Team

2 Team

Date

2016-10-31

Team Information

201511260 문 성찬
201511284 이 종빈
201211356 송 원종

Define of Unit test

Unit test를 하는 이유는 소스 코드가 각 모듈별로 의도하는 바에 대하여 정확하게 작동하는지 검증하는 절차이다. 각 세션을 분리시켜 오류가 생겼을 때 이를 쉽게 감지 및 변경을 하게 해준다. 이를 위해서 프로그래밍 소스 코드를 작은 테스트 케이스 별로 분리시켜야 한다.

C Unit test tool list

Name	xUnit	Fixtures	Group fixtures	Generators	License
AceUnit	Yes	Yes			BSD License
API Sanity Checker	Yes	Yes (spectypes)	Yes (spectypes)	Yes	LGPL
Automated Testing Framework					BSD
Autounit (GNU)					LGPL
Catsrunner					GPL
Cfix	Yes				
Cgreen					LGPL
Check	Yes	Yes	Yes		LGPL
Cmocka	Yes	Yes	Yes		Apache License 2.0
Cmockery	Yes				Apache License 2.0
CppUTest	Yes	Yes	No	Yes	
Criterion	Yes	Yes	Yes	Yes	MIT
CTest	Yes	Yes	Yes		Apache License 2.0
CU					LGPL
CUnit	Yes				LGPL
CUnitWin32	Yes				LGPL
CUT	No				BSD
CuTest	Yes				zlib
Cutter	Yes				LGPL
EmbeddedUnit	Yes	Yes			MIT
Embunit	No				Proprietary

FCTX	Yes				BSD
GLib Testing	Yes	Yes			
GUnit					
lcut	Yes	Yes	Yes		Apache License 2.0
libcbdd	Yes	Yes	Yes		Apache License
LibU	Yes	No			BSD
MinUnit					as-is
Mut	No	No	No	No	MIT
NovaProva	Yes	Yes	No	Yes	Apache License 2.0
Opmock	Yes	Yes	Yes	Yes	GPLv3
Parasoft C/C++test	Yes	Yes	Yes	Yes	Proprietary
QA Systems Cantata	No	Yes	Yes	Yes	Proprietary
RCUNIT	Yes	Yes	Yes		MIT
RTRT					
SeaTest	Yes	Yes			MIT
Smarttester					
Sput					2-clause BSD
STRIDE	Yes	Yes	Yes	No	Proprietary
TBrun					
Tessy					
Test Dept.	Yes				GPL
TestApe					
TF unit test	Yes	Yes			GNU Lesser GPL
tinytest	Yes	Yes			Apache
TPT	Yes	Yes	Yes	Yes	Proprietary
Unity	Yes			Yes	MIT
VectorCAST/C	No	Yes	Yes	Yes	Proprietary
Visual Assert	Yes				
xTests					BSD

Cmake

복잡한 make 를 대용하여 생성될 타겟에 의존적인 파일들과 빌드옵션 등을 자동적으로 설정해주는 기능을 가지고 있다. 즉 기존의 make 를 자동으로 생성해주는 툴.

cmake 를 사용하기 위해서는 CMakeLists.txt 라는 파일을 만들어서 cmake 에게 명령을 내려야함. 우리가 빌드할 파일은 크게 라이브러리 파일과 유닛테스트 파일로 나눌 수 있다. cmake 는 기본적으로 변수를 설정하고 cmake 가 제공하는 함수에 설정된 변수를 전달하는 형태가 많이 사용됩니다. 라이브러리에 들어갈 소스 파일들의 이름을 변수에 저장하고 라이브러리를 빌드하는 함수에 이 변수를 전달하면, 자동으로 해당 파일들을 찾아서 라이브러리로 빌드한다. 실행 파일을 만들 때도 실행 파일에 들어갈 소스 파일 이름과 실행 파일의 이름을 빌드 함수에 전달하면 실행 파일을 빌드한다.

예시 ->

CMakeLists.txt

```
CMAKE_MINIMUM_REQUIRED (VERSION 2.6)
PROJECT (calib_project)
SET(CMAKE_VERBOSE_MAKEFILE ON)
if ("${build}" MATCHES "debug")
 SET (CMAKE_BUILD_TYPE "debug")
else ("${build}" MATCHES "debug")
 SET (CMAKE_BUILD_TYPE "release")
endif ("${build}" MATCHES "debug")
//cmake 를 실행할 때 -D 옵션으로 cmake 에게 변수 값을 전달
// cmake -D build=debug 형태로 cmake 를 실행, build 라는 이름의 변수를 만들어서 cmake 에게 전달 가능
ADD_DEFINITIONS(-DCALIB_CFG_BUILD_MODE="${CMAKE_BUILD_TYPE}")
//컴파일 옵션에 추가될 매크로 정의는 ADD_DEFINITIONS 함수로 지정
if ("${bit}" MATCHES "32")
 ADD_DEFINITIONS(-DCALIB_CFG_COMPILE_BIT=32 -m32)
 SET (CMAKE_EXE_LINKER_FLAGS -m32)
else ("${bit}" MATCHES "32")
 ADD_DEFINITIONS(-DCALIB_CFG_COMPILE_BIT=64 -m64)
 SET (CMAKE_EXE_LINKER_FLAGS -m64)
endif ("${bit}" MATCHES "32")
//빌드 비트를 확인하는 부분
ADD_DEFINITIONS(-Wall -DCALIB_CFG_OS="${CMAKE_SYSTEM_NAME}" -
DCALIB_CFG_CPU="${CMAKE_SYSTEM_PROCESSOR}")
```

ADD_SUBDIRECTORY(src)

src/CMakeLists.txt

ADD_SUBDIRECTORY(test).
SET (LIBRARY_OUTPUT_PATH ../lib)
//빌드된 라이브러리가 저장될 디렉토리
INCLUDE_DIRECTORIES (../include)
//소스 빌드에 필요한 헤더 파일이 저장된 디렉토리
SET (LIBSRCS sys_info.c build_info.c)
//소스 파일의 리스트
CMAKE_MINIMUM_REQUIRED (VERSION 2.6)
PROJECT (calib_project)
set dir to store library
SET (LIBRARY_OUTPUT_PATH ../lib)
set dir for header
INCLUDE_DIRECTORIES (../include)
sources
SET (LIBSRCS sys_info.c build_info.c)
build libcalib.a with LIBSRCS sources
ADD_LIBRARY (calib STATIC ${LIBSRCS})

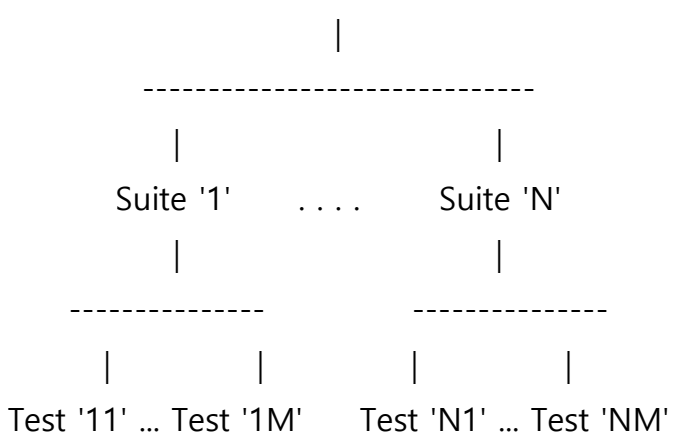
test/CMakeLists.txt

ENABLE_TESTING ()
//ctest 를 이용, ctest 를 사용하겠다는 의미로 ENABLE_TESTING 함수를 호출
INCLUDE_DIRECTORIES (../include)
//헤더 파일의 디렉토리 설정
ADD_EXECUTABLE (test_build_info test_build_info.c)
//ADD_EXECUTABLE 은 어떤 소스를 가지고 어떤 실행 파일을 만들지를 지정
// test_build_info.c 를 가지고 test_build_info 실행파일을 빌드합니다.
TARGET_LINK_LIBRARIES (test_build_info calib)
//test_build_info 를 빌드할 때 calib 라이브러리를 링크
ADD_EXECUTABLE (test_build_info test_build_info.c)
TARGET_LINK_LIBRARIES (test_build_info calib)
//test_build_info 의 빌드와 동일합니다. test_sys_info.c 소스와 calib 라이브러리를 가지고 test_sys_info 라는 실행 파일을 빌드
ADD_TEST (unittest1 test_build_info)
ADD_TEST (unittest2 test_sys_info)
//ADD_TEST 함수는 ctest 가 실행할 유닛테스트 파일을 지정
$ cmake -D build=debug -D bit=64 CMakeLists.txt
$ make
//유닛테스트를 실행하기 위해서는 cmake test 명령을 내리거나 ctest 를 실행
$ cmake test
$ ctest
//cmake test 를 실행하면 유닛테스트가 정상적으로 실행되었다는 메시지만 출력
// 유닛테스트에서 출력한 메시지를 확인하고 싶다면 ctest -V 나 ctest --debug 명령을 실행

CUnit

CUnit 은 C 에서 Unit test 를 작성, 관리 및 실행하기 위한 시스템으로, 사용자의 test code 가 링크되는 정적 라이브러리로서 구축된다. CUnit 의 test 구조를 구축하기 위한 간단한 구조를 사용하고, 일반적인 데이터 유형을 test 하기 위한 다양한 세트를 제공한다. 또한, 여러 가지 인터페이스 test 를 실행하고, 그 결과를 보고하기 위해 제공된다.

- 구조 –



Test registry 에는 여러 개의 suite 가 존재하며, 각각의 suite 에는 여러 개의 Test 가 존재한다.
즉, suite 가 N 개, test 가 각각 M 개씩 존재한다면, 총 test 개수는 N*M 개이다.
Test registry 에 suite 를 추가하는 것은 사용법의 3 번에 해당하며, 각 suite 에 test 를 추가하는 것은 사용법의 4 번에 해당한다.

- 사용법 –

테스트에 대한 함수를 작성한다. (필요한 경우 suite 를 초기화 및 정리) ①

테스트 registry 를 초기화한다. – CU_initialize_registry() ②

테스트 registry 에 스위트 룸을 추가 – CU_add_suite() ③

스위트 룸에 테스트를 추가 – CU_add_test() ④

적절한 인터페이스를 사용하여 테스트를 실행 ⑤

테스트 registry 를 정리 – CU_cleanup_registry() ⑥

예시

```
/*
 *  Simple example of a CUnit unit test.
 *
 *  This program (crudely) demonstrates a very simple "black box"
 *  test of the standard library functions fprintf() and fread().
 *  It uses suite initialization and cleanup functions to open
 *  and close a common temporary file used by the test functions.
 *  The test functions then write to and read from the temporary
 *  file in the course of testing the library functions.
 *
 *  The 2 test functions are added to a single CUnit suite, and
 *  then run using the CUnit Basic interface.  The output of the
 *  program (on CUnit version 2.0-2) is:
 *
 *           CUnit : A Unit testing framework for C.
 *           http://cunit.sourceforge.net/
 *
 *       Suite: Suite_1
 *         Test: test of fprintf() ... passed
 *         Test: test of fread() ... passed
```

```
 *
 *       --Run Summary: Type      Total     Ran   Passed   Failed
 *                      suites        1       1     n/a        0
 *                      tests         2       2       2        0
 *                      asserts       5       5       5        0
 */

#include <stdio.h>
#include <string.h>
#include "CUnit/Basic.h"

/* Pointer to the file used by the tests. */
static FILE* temp_file = NULL;

/* The suite initialization function.
 * Opens the temporary file used by the tests.
 * Returns zero on success, non-zero otherwise.
 * 여기서부터가 ① 시작부분.
 * 테스트에 대한 함수를 작성하고, initialize 및 clean 작성함.
 */
int init_suite1(void)
{
   if (NULL == (temp_file = fopen("temp.txt", "w+"))) {
      return -1;
   }
   else {
      return 0;
   }
}

/* The suite cleanup function.
 * Closes the temporary file used by the tests.
 * Returns zero on success, non-zero otherwise.
```

```
 */
int clean_suite1(void)
{
   if (0 != fclose(temp_file)) {
      return -1;
   }
   else {
      temp_file = NULL;
      return 0;
   }
}

/* Simple test of fprintf().
 * Writes test data to the temporary file and checks
 * whether the expected number of bytes were written.
 */
void testFPRINTF(void)
{
   int i1 = 10;

   if (NULL != temp_file) {
      CU_ASSERT(0 == fprintf(temp_file, ""));
      CU_ASSERT(2 == fprintf(temp_file, "Q₩n"));
      CU_ASSERT(7 == fprintf(temp_file, "i1 = %d", i1));
   }
}

/* Simple test of fread().
 * Reads the data previously written by testFPRINTF()
 * and checks whether the expected characters are present.
 * Must be run after testFPRINTF().
 */
void testFREAD(void)
```

```
{
   unsigned char buffer[20];

   if (NULL != temp_file) {
      rewind(temp_file);
      CU_ASSERT(9 == fread(buffer, sizeof(unsigned char), 20, temp_file));
      CU_ASSERT(0 == strncmp(buffer, "Q₩ni1 = 10", 9));
   }
}
/* 여기까지가 ①
 * ① 끝부분
 */

/* The main() function for setting up and running the tests.
 * Returns a CUE_SUCCESS on successful running, another
 * CUnit error code on failure.
 */
int main()
{
   CU_pSuite pSuite = NULL;

   /* initialize the CUnit test registry */
   if (CUE_SUCCESS != CU_initialize_registry())
      // if 문 안의 조건이 ②에 해당하는 부분 test registry 를 초기화함.
      return CU_get_error();

   /* add a suite to the registry */
   pSuite = CU_add_suite("Suite_1", init_suite1, clean_suite1);
     // 윗줄이 ③에 해당하는 부분. suite 를 추가함.
   if (NULL == pSuite) {
      CU_cleanup_registry();
      return CU_get_error();
   }
```

```
   /* add the tests to the suite */
   /* NOTE - ORDER IS IMPORTANT - MUST TEST fread() AFTER fprintf() */
   if ((NULL == CU_add_test(pSuite, "test of fprintf()", testFPRINTF)) ||
       (NULL == CU_add_test(pSuite, "test of fread()", testFREAD)))

      // if 문 안의 add_test 부분이 ④에 해당하는 부분. Test 를 추가함.
   {
      CU_cleanup_registry();
      return CU_get_error();
   }

   /* Run all tests using the CUnit Basic interface */
   CU_basic_set_mode(CU_BRM_VERBOSE);
   CU_basic_run_tests();
      // ⑤에 해당하는 부분. 적당한 interface 로 test 진행
   CU_cleanup_registry();
     // ⑥에 해당하는 부분. Registry 를 정리함.
   return CU_get_error();
}
```

Check Unit test

Check 의 특징(autoconf, autotool 등 여러가지 요구사항이 있다)

1. Check 는 적은 양의 코딩을 테스트하는데 특화 되어있다.
2. 테스트는 별도의 어드레스 공간에서 실행된다.(세그먼테이션 오류 혹은 시그널 캐치와 같은 코드에러와 장애를 확인할 수 있다)

Check Unit test 방법

Check 유닛 테스트를 하는 것은 간단하다.

요약)아래의 트리와 같은 구성을 만들고 원하는 유닛 테스트에 대한 정보를 test/Check_money.c 에 넣고 원래 소스 코드는 src/main.c 에 넣는다.

그후 명령어

autoreconf—install

$ ./configure

$ make

와 같은 형식으로 unit test 를 실행 시킬 수 있다.

make 가 unit test 의 결과를 알려준다.

Unit test 생성시 주의 사항과 방법

우선 Check.h 라는 헤더 파일을 include 시킨다

>>#include <check.h>

유닛 테스트는 아래와 같은 형식으로 만들어진다.

```
START_TEST(test_name)
{
/*유닛 테스트할 내용*/
}
```

예를 들어 돈을 관리하는 것에 대한 유닛테스트를 실행한다고 하였을 때

Check 유닛 테스트는 아래와 같은 트리 형식으로 구성되어있다.

|-- **Makefile.am (1)**

|-- README

|-- **configure.ac (2)**

|-- src

| |-- **Makefile.am (3)**

| |-- **main.c (4)**

| |-- money.c

| `-- money.h

`-- tests

  |-- **Makefile.am(5)**

  `-- check_money.c (6)

(1) **Makefile.am** :서브 디렉토리의 실행될 순서를 알려준다. (src다음에 test가 진행됨.)

(2) **Configure.ac**: Autoconf를 실행시키면 configure.ac를 기준으로 자동으로 스크립트를 완성시켜준다.

(3) **src/Makefile.am**: libmoney를 만들고 main과 연결시켜준다.

(4) **main.c**: main.c안에는 코드의 기능(function)만 들어가야하며 unit test를 어떻게 할 것인가에 대한 내용은 check_money.c(6)에 들어간다.

*여기서 **(1)makefile.am** 에서 src 다음에 test 을 진행시키는데 이러한 이유는 src 에 컴파일된 내용을 바탕으로 tests 에서 unit test 를 실행시키기 때문이다.

(5) **tests/Makefile.am** 의 빌드 내용은 아래와 같다.

```
## Process this file with automake to produce Makefile.in

TESTS = check_money
check_PROGRAMS = check_money
check_money_SOURCES = check_money.c $(top_builddir)/src/money.h
check_money_CFLAGS = @CHECK_CFLAGS@
check_money_LDADD = $(top_builddir)/src/libmoney.la @CHECK_LIBS@
```

TESTS 는 어떤 파일을 체크 할 것인지를 지시해준다,
Check_PROGRAMS 는 TESTS 에서 지정해준 파일을 기준 자동으로
check_money_xxxx 의

파일들을 생성해준다. 이때 Check_PROGRAMS 는 TESTS 가 유효할 때 만 진행된다.
@CHECK_CFLAGS@ 와 **@CHECK_LIBS@**은 올바른 컴파일러와 linker flag 가 사용됐는지 판단해준다.(형식에 맞는지 판단)
$(top_builddir)/:는 src 의 c 파일, h 파일과 연결시켜준다.
Money.h 는 아래와 같이 구성되있다.

```
#ifndef MONEY_H
#define MONEY_H
#endif
```

***주의점**

Src/money.c 는 아무것도 안에 있으면 안된다.
Tests/check_money.c 는 빈 main() function 만을 가지고 있어야한다.

Unit test 의 결과에 따라 에러 메시지 또는 unit test 할 내용을 넣어야 하는데 이는 위의 money 에 대한 예제로 들 시 unit test 내용을 check_money.c 에 코딩한다.
기존에 check_money.c 에는 아래와 같은 main 만 있었는데

```
int main(void)
 {
     return 0;
 }
```

아래와 같이 내용이 추가된다.

```
#include <check.h>
#include "../src/money.h"

START_TEST(test_money_create)
{
    Money *m;
    m = money_create(5, "USD");
    ck_assert_int_eq(money_amount(m), 5);//1
    ck_assert_str_eq(money_currency(m), "USD");//2
    money_free(m);
}
END_TEST

 int main(void)
 {
     return 0;
 }
```

위에서 설명한 unit test 를 시작하는 START_TEST 와 END_TEST 를 추가한 내용이다.

1. **Ck_assert_int_eq**은 두 integer가 동일한지 판단.
2. **Ck_assert_str_eq**은 두 string이 동일한지 판단.

   Ck_assert_msg(strcmp(money_currency(m),"USD")==0,"USD 의 환율을 예상하고있었으나 %s 가 발견됨",money_currency(m)); 와 같은 방식으로 에러 시 메시지를 송출 할 수도 있다.

이와 같이 check_money.c 의 START 와 END 사이에 원하는 조건을 넣어서 unit test 를 진행할수있다.