UNIT TESTING IN C : A FIRST CONTACT

Prologue
Testing and evaluating an application is well known to be a very difficult and complex task, often even more complex than designing and implementing the given application! The failures and bugs in our daily software confirm and illustrate this fact very well.

The manual and automatic testings in interactive and batch modes we present here are often referred in the literature as UNIT testing, in complement to other type of testings like integrative testing.

A Bit Theory about UNIT Testing
Unit testing is a method which allows us to test individual units of source code. Unit tests are implemented in order to determine if the source code is fit for use. *The ultimate goal of unit testing is to isolate each part of the program and show that the individual parts are correct*.

Unit testing also plays an important role in *test-driven development*, a programming technique which relies on the repetition of very short development cycles. It's called *test-driven* because the developers are required to write the tests first, and then implement the corresponding code. The idea is to first define the code's requirements in the tests before writing the actual code.

Unit tests are especially useful when you are refactoring code and you want to make sure the refactoring doesn't affect the code's interface. The tests then make sure that other modules which depend on the code don't have to be changed in order to use the refactored code.

Main test program schemes

The main idea is to compare the obtained result of a new implemented function or set of functions with a handcrafted solution, and in case of refactoring with the original primitive(s).

For our labs, we have identified four main test program schemes:

1. Read a couple of simple data (e.g. an integer and a word)
   Do some action
   Print the result -- *and compare it with a handcrafted solution*
2. Repeat (1) until some conditions have been realized (e.g. an end of file has been reached).
3. Read a set of simple data (e.g. a list of integers and words)
   Do some action
   Print the result -- *and compare it with a handcrafted solution*
4. Test related actions, as they commonly appear in abstract data types or objects (like push/pop, add/remove, malloc/free, ...).

At first glance, these tasks appear to be very simple. In fact, they are not trivial, since we have to produce a program where not all possible cases have been enumerated during the specification and implementation time: during test time, the program should be flexible enough to be tested with all kinds of reasonable inputs, interactively as well as in batch mode.

Depending on the programming language and environment, this can be really difficult. Fortunately, in most modern languages and environments, especially for C/Unix, these tasks are feasible.

In C/Unix there are several methods to pass data to a process: reading a file, parameter passing to the main program, messages and events. Here we will only study the first one.

In the following we sketch C program excerpts for the four above mentionned schemes, and then we will introduce the notion of automatic testing. Furtunately it turns out that in the C/Unix environment the test programs are rather simple, but a clever error treatment is a bit more tricky.

Case 1: Test an action with a couple of simple inputs
For example, read a key (integer) and a word, and test some action:

```
int k; char w[20];

scanf("%i %s", &i, w);
// Do an action
// Print the result
```

and add, if useful, some error treatment of the input reading:

```
int k; char w[20];

while (scanf("%i %s", &k, w)) < 2)  //--- error treatment
   while (getchar() != '\n') ;      // skip the rest of the input line

// Do an action
// Print the result
```

Case 2: Repeat case 1 until some conditions are realized
For example, repeat case 1 until the end of the input file is encountered :

```
int k; char w[20];

while (scanf("%i %s", &k, w) != EOF) {
  // Do an action
  // Print the result
}
```

and add, if useful, some error treatment of the input reading:

```
int k; char w[20];
int d; char c;  // d for diagnostic

while ((d = scanf("%i %s", &k, w)) != EOF) {
  if (d >= 0 && d < 2) goto error;  //--- error treatment

  // Do an action
  // Print the result
  continue;

error:  //--- error treatment
  while (getchar() != '\n') ;  // skip the rest of the input line
  printf("return number of scanf: %d. Try again.\n", d);
}
```

Note that in this case, the error treatment appears at the end of the main loop, and is so properly separated from the main algorithm.

Case 3: Test an action with a set of simple inputs
For example, read a list of keys (integers), and test some action:

```
int i=0, key[1000];

while (scanf("%d", &key[i++]) != EOF) ;  // Read all data

// Do an action
// Print the result
```

and add, if useful, some error treatment of the input reading:

```
int i=0, key[1000];
int d;  // d for diagnostic

while ((d=scanf("%d", &key[i++])) != EOF) {  // Read all data

  if (d==0) { //--- error treatment
    while (getchar() != '\n') ;  // skip the rest of the input line
    --i;  // i has been incremented once to much !
  }

}  // End of readings

// Do an action
// Print the result
```

Note that in this case, the error treatment is strongly coupled with the input reading.

Case 4: Test related actions

This case is more tricky and will be illustrated in the case of a stack: push, pop, consult_top. A more elaborated version will later be studied in the Abstract Data Type chapter.

The following framework illustrates the case without error treatment:

```
while (c=getchar()) {
  switch(c) {
    case 'p': scanf("%i,&i); push(i); break;
    case 'o': pop(); break;
    case 'c': consult_top(&i); printf("%i\n",i); break;
    case 'Q': return 0; // Quit the test program
  }
}
```

The next framework is similar to the previous one, but this time with an exhaustive error treatment: reading a non integer from the input file, overflow (push an item to a full stack), underflow (pop an item from an empty stack), and consult the top of an empty stack :

```
while (c=getchar()) {
  switch(c) {
    case 'p': if (!scanf("%i,&i)) goto error; //--- possible input failure
              if (!push(i)) goto error; //--- possible overflow
              break;
    case 'o': if (!pop()) goto error; //--- possible underflow
              break;
    case 'c': if (!consult_top(&i)) goto error; //--- possible empty stack
              printf("%i\n",i);
              break;
    case 'Q': return 0; // Quit the test program
  }
  continue; // all is ok, go to the beginning of the main loop

error:
  //--- error treatment
}
```

To finish this tutorial we introduce the important notion of automatic testing, i.e. testing codes without the need for a human to check if the program output is correct.

Case 5: Automatic Testing
The idea is to compare the obtained result with a well known value and to complain only if the values differ. Fig. 1 (a high level pseudo code) and Fig. 2 (a C code excerpt) illustrate a classical situation where we have to refactore the implementation of a function f(x).

```
loop
  Get next item x.
  Compute my_f(x).
  Compare its value with f(x).
  Complain only if the values differ.
end loop
```

Figure 1. The idea of automatic testing

```
while ( (c=getchar()) != EOF ) {
  if ( my_f(c) != f(c) ) printf("ERROR -- %c, %c, %c\n", c, my_f(c), f(c));
}
```

Figure 2. The idea of automatic testing in C.

Suppose that the code of Fig. 2 has been compiled into a program named `‘test’`. Then we can run it in different modes, illustrated by Fig. 3, where `‘input.txt’` is a file containing a given test script.

```
% test                          // interactive
% test  < input.txt             // batch
% test  < input.txt > res.txt   // batch
```

Figure 3. Interactive and batch modes

If in Fig 2 we replace the second line just by :

`‘printf(%c, %c, %c\n", c, my_f(c), f(c);’`,

i.e. without the test 'if (my_f(c) != f(c))', we will have to check the results by hand. In the literature, this is called *manual testing* in opposite to *automatic testing.*

Summary

1. Testing a simple program follows the schema :

   - read input –> action –> print result, and analyse it critically or test it automatically

   and the procedure is possibly repeated. Note that it is very important that the inputs are not hard coded, but can be chosen as flexible as possible during test and evaluation time !

2. In C code, the error treatment of input reading is essentially captured by the two schemes:

   - Locally, i.e. just when the error occurs.
   - At the end of a loop or a procedure (with the goto mechanism).
     Remark: this very low level *goto* mechanism is replaced in higher level languages by the *event* or *exception* mechanism.

Note: The complete C codes are available online at http://www.unifr.ch/diuf/pai/ip > Tutorials.

Further readings: a good start is:

[1] Wikipedia: http://en.wikipedia.org/wiki/Unit_testing

[2] Wikipedia: http://en.wikipedia.org/wiki/Test_script

Prerequisite
A good knowledge of C’s input/output (i.e. scanf and printf), and of the Unix shell line command, including the two indirection operators < (input redirection) and > (output redirection).

Acknowledgements
The merits of introducing in this tutorial the important notion of automatic testing and the remarks about UNIT testing go to Muriel Bowie.