Enhetstestning med CUnit

Tobias Wrigstad

Inledning

CUnit är ett ramverk för att skriva och utföra enhetstester på C-kod. CUnit är utvecklat som ett biliotek av funktioner som länkas samman med användarens testkod.

Denna enkla lathund är menad att komplettera ”CUnit Programmers Guide” som finns på `http://cunit.sourceforge.net/doc/introduction.html`. Den visar hur man snabbt kan komma igång med CUnit, samt den kompilatorflagga som måste anges vid kompilering med CUnit på institutionens datorsystem. Titta på några av enhetstesterna som distributerats med koden för inlämningsuppgifterna (här tar vi exempel från en fil `unittests.c` från en inlämningsuppgift från en forntida IOOPM) och försök följa dem för att se hur enkelt enhetstest kan sättas upp.

Ett enhetstest är ett test av en enhet, t.ex. en modul eller sammanhängande samling av funktioner. (Täcks av föreläsning på kursen.) Ett enhetstest av en modul för att logga programmeddelanden på disk kunde t.ex. bestå av följande komponenter:

1. Sätta upp testet: skapa kataloger och filer för loggmeddelanden
2. Test att logga på
   (a) en tom fil,
   (b) en fil med en rad text i,
   (c) en fil med många rader text i,
   (d) en fil som inte finns

   med,
   (e) ett tomt loggmeddelande,
   (f) ett meddelande med ett tecken,
   (g) ett långt meddelande

   Utför testen $\{a, b, c, d\} \times \{e, f, g\}$[1] och jämför det förväntade utdatat med det faktiska utfallet och signalera fel
3. Riva ned testet: ta bort skapade kataloger och filer

Grundläggande

Ett enhetstest i CUnit består av ett antal testsviter som var och en innehåller ett antal olika test. Filen `unittests.c` i som vi använder som löpande exempel har tre sviter, en för test av ett binärt sökträd, ett för en enkellänkade lista och ett för att läsa ord från en inström.

Följande kod skapar den sistnämnda sviten i variabeln `pSuiteNW`. Först deklareras variabeln (rad 1), sedan skapas sviten (rad 6) med ett namn `"nextWord Suite"`, samt funktionerna för att sätta upp samt riva ned testen (`init_suite_nw` och `clean_suite_nw`), se nästa avsnitt. Om sviten inte skapades korrekt är värdet i `pSuiteNW` `NULL`; vi städar bland de registrerade testerna (rad 9) och avslutar genom att returnera en felkod (rad 10).

[1] Där t.ex. (a, g) avser ett test med en tom fil till vilken ett långt meddelande skrivs.

Rad 13–16 lägger till ett test i sviten. Funktionen `CU_add_test` lägger till testfunktionen `testNEXTWORD`, en funktion som vi (utvecklaren) skrivit själva, i sviten med ett beskrivande namn. Fel fångas upp och rapporteras på samma sätt som tidigare.

På rad 18 anges att vi vill att testen skall utföras ”verbose”, alltså att alla detaljer skall skrivas ut när testen körs. Rad 19 kör alla test (i detta fall bara ett). Rad 20 städar upp bland de registrerade testen (avallokerar minne, etc.). Slutligen, på rad 21, returnerar vi *de eventuella fel* som uppstått under körning.

```
1  CU_pSuite pSuiteNW = NULL;
2
3  if (CUE_SUCCESS != CU_initialize_registry())
4    return CU_get_error();
5
6  pSuiteNW = CU_add_suite("nextWord Suite", init_suite_nw, clean_suite_nw);
7
8  if (NULL == pSuiteNW) {
9    CU_cleanup_registry();
10   return CU_get_error();
11 }
12
13 if (NULL == CU_add_test(pSuiteNW, "test of nextWord()", testNEXTWORD)) {
14   CU_cleanup_registry();
15   return CU_get_error();
16 }
17
18 CU_basic_set_mode(CU_BRM_VERBOSE);
19 CU_basic_run_tests();
20 CU_cleanup_registry();
21 return CU_get_error();
```

Att sätta upp och riva ned tester

För vissa samlingar av test kan det vara smidigt att först utföra ett initialt arbete. Det kan röra sig om att skapa filer och kataloger i filsystemet där testerna kommer att skriva och läsa, eller t.ex. skapa ett antal binära sökträd som testerna sedan opererar på.

I `unittests.c` i inlämningsuppgift 1 finns följande två funktioner.

```
1  int init_suite_bst(void) {
2    return 0;
3  }
4
5  int init_suite_nw(void) {
6    temp_file = fopen("temp.txt", "w+"); // global variabel
7    if (temp_file == NULL {
8      return -1;
9    } else {
10     return 0;
11   }
12 }
```

Den första funktionen initierar alla test av modulen `bst` – det binära sökträdet, och gör som synes ingenting. Alla tester av sökträdet skapar ett nytt träd och utför testerna på det.

Den andra funktionen initierar alla test av modulen för `nextWord` och öppnar filen `temp.txt` för skrivning i den aktuella katalogen. Om detta inte är möjligt kommer ett fel signaleras och testen inte utföras vidare – vilket är rimligt då förutsättningarna för att läsa in ord uppenbarligen inte finns.

En motsvarande funktion river också ned `nextWord`-sviten av tester:

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

Här stängs filen i fråga, varvid vi kan rapportera att nedrivning av testen gick enligt planen (**return** 0).

Utföra tester

Varje funktion vars namn börjar på `test` avser ett test av en funktion i enheten. Följande test som återfinns bland enhetstesterna till inlämningsuppgift 1 testar att insertering i ett binärt sökträd skapar ett träd med förväntat djup.

Funktionen **int** `depth(TreeLink)` används för att ta reda på trädets djup. Denna funktion är inte strikt nödvändig för insertering och sökning, men är en hjälpfunktion som utvecklaren av trädet tillhandahåller bl.a. just för att underlätta test av trädet. Det är relativt vanligt att tillhandahålla ”extra kod” på detta sätt. Det underlättar testandet och håller dessutom testen på en rimlig abstraktionsnivå! I föreliggande exempel, om trädets interna representation ändras behöver vi inte skriva om testet. Mycket smidigt. Men ack! Den sista raden i testet nedan följer inte denna princip.

I koden nedan skapas ett nytt träd. För varje insertering kontrolleras att det resulterande trädets djup är den förväntade. Detta görs med `CU_ASSERT(<exp>)` där `<exp>` är ett booleskt uttryck som förväntas evaluera till *sant*. Om något uttryck i någon assert-sats evaluerar till *falskt* (vi säger att asserten fejlar, alt. misslyckas, på Svenska) under testet räknas testet som att det inte har passerat.

```
void testBST_DEPTH(void) {
  TreeLink t = insert(NULL, "ni", 1);
  CU_ASSERT(depth(t) == 1);
  t = insert(t, "spam", 2);
  CU_ASSERT(depth(t) == 2);
  t = insert(t, "eki", 3);
  CU_ASSERT(depth(t) == 2);
  t = insert(t, "eki", 4);
  CU_ASSERT(depth(t) == 2);
  CU_ASSERT(strcmp(t->left->key, "eki") == 0); // Bryter mot abstraktionsprincipen!
}
```

Observera att testet inte returnerar något. Om funktionen körs utan att någon assert-sats misslyckas anses testet ha passerat.

De olika typerna av assertions som finns är dokumenterade på `http://cunit.sourceforge.net/doc/writing_tests.html`.

Om man ville undvika att bryta mot abstraktionsprincipen skulle man kunna utveckla en funktion som tillät åtkomst till en specifik nod i trädet som en del av trädmodulen. Man skulle t.ex. kunna skriva följande:

```
char *keyForPath(TreeLink t, char *path) {
  if (!t) return NULL;
  switch (*path) {
    case 'L': return keyForPath(t->left, ++path);
    case 'R': return keyForPath(t->right, ++path);
    case '\0': return t->key;
    default:
      printf(stderr, "Bogus path value '%c' expected L or R\n", *path);
```

```
 9     }
10     return NULL;
11   }
```

Denna funktion vandrar i trädet i enlighet med en sökträng. T.ex. returnerar `"LRLLR"` den nyckel som fås efter att först gå vänster, sedan höger, sedan två gånger vänster, och sist höger i trädet.

Nu skulle man kunna skriva om sista raden i testet så här:

```
1    char *temp = keyForPath(t, "L");
2    CU_ASSERT(strcmp(temp, "eki") == 0);
```

Ofta kan det vara en bra idé att inte lägga denna typ av funktion i sin moduls headerfil, utan istället skapa en speciell headerfil som är specifik för testning och som definierar de ytterligare funktionerna.

Titta på de olika `test`-funktionerna i `unittests.c` i inlämningsuppgift 1 för att se olika exempel på tester av både ett binärt sökträd och en enkellänkad lista.

Arbeta på egen maskin

Om du vill arbeta på din egen maskin måste du installera CUnit. Detta handleds endast i mån av tid och förmåga, givet att din dator är åtkomlig från någon av institutionens datorsalar.

CUnit finns på `http://cunit.sourceforge.net/`.

Kompilera med CUnit

Vid kompilering med CUnit måste man explicit ange att CUnit skall länkas in. Detta görs med flaggan `-l`, med parametern ’`cunit`’, t.ex.:

```
gcc -ggdb -Wall -std=c11 unittests.c list.c bst.c -o unittests -lcunit
```

Om `-lcunit` inte anges kommer länknings-steget efter kompileringen att misslyckas, eftersom funktionerna för enhetstest, t.ex. `CUAssert`, då fortfarande saknas.