Praktikum zu Informatik 2

Freudenreich, Paulus, Schröder

Einführung ins Thema Testing in C

1. Was ist Testing?

Softwareentwicklung ist fehleranfällig. Selbst kleine Programme können unerwartetes Verhalten zeigen, wenn Eingaben nicht geprüft werden, Randfälle nicht bedacht werden oder Speicher falsch verwaltet wird. Testing bezeichnet systematische Verfahren, um Programme auf Korrektheit, Stabilität und Wartbarkeit zu überprüfen.

Wichtige Aspekte:

- **Unit-Tests** prüfen einzelne Funktionen oder Module isoliert, um deren korrektes Verhalten sicherzustellen.
- **Integrationstests** überprüfen das Zusammenspiel mehrerer Komponenten.
- **Systemtests** betrachten das komplette Programm aus Sicht der Benutzer.

Unit-Tests sind in der Praxis besonders relevant, da sie Fehler frühzeitig aufdecken und eine kontinuierliche Weiterentwicklung erleichtern. In der Programmiersprache C, die von sich aus keine eingebaute Testunterstützung bietet, werden externe Testframeworks eingesetzt, z.B. Unity oder CMocka.

2. Was sind Unit-Tests?

Ein Unit-Test überprüft eine einzelne Funktion oder ein kleines Modul isoliert. Es geht nicht darum, das gesamte Programm zu testen, sondern gezielt zu prüfen:

- Gibt die Funktion die erwarteten Werte zurück?
- Verhält sie sich korrekt bei Randfällen oder ungültigen Eingaben?
- Wird Speicher korrekt verwendet und entstehen keine Fehler?

Beispiel 1: einfache Funktion

```c
int multiply(int a, int b) {
    return a * b;
}
```

Ein Unit-Test könnte die Funktion wie folgt prüfen:

```c
// Test 1
result = multiply(2, 3);
if(result == 6)
    printf("Test 1 bestanden\n");
else
```

```
    printf("Test 1 fehlgeschlagen: erwartet 6, erhalten %d\n", result);

// Test 2
result = multiply(0, 5);
if(result == 0)
    printf("Test 2 bestanden\n");
else
    printf("Test 2 fehlgeschlagen: erwartet 0, erhalten %d\n", result);

// Test 3
result = multiply(-3, 4);
if(result == -12)
    printf("Test 3 bestanden\n");
else
    printf("Test 3 fehlgeschlagen: erwartet -12, erhalten %d\n", result);
```

Jeder Test überprüft die Funktion isoliert. Schlägt ein Test fehl, liegt das Problem direkt in `multiply`, nicht irgendwo anders im Programm.

Beispiel 2: komplexere Parameter

```
int* readMyData(double value, char *filename);
```

Hier können Unit-Tests die Funktion mit unterschiedlichen Werten und Dateien aufrufen, um sicherzustellen, dass die zurückgegebenen Daten in allen Fällen korrekt sind. So werden Fehler früh erkannt, bevor sie andere Programmteile beeinflussen.

3. Unit-Tests in C

C bietet keine eingebaute Testunterstützung. Deshalb werden externe Testframeworks wie Unity eingesetzt. Ein typischer Ablauf beim Ausführen von Unit-Tests sieht folgendermaßen aus:

Testinitialisierung:

Das Framework startet und legt interne Strukturen für die Testergebnisse an. Optional werden `setUp()`-Funktionen ausgeführt, um benötigte Ressourcen vorzubereiten.

Testausführung:

Jede Testfunktion wird isoliert ausgeführt. Die zu prüfenden Funktionen werden aufgerufen, und ihr Verhalten wird mit Assertions geprüft (z. B. `TEST_ASSERT_EQUAL_INT`).

Assertion:

Assertions vergleichen den erwarteten Wert mit dem tatsächlichen Ergebnis. Schlägt eine Assertion fehl, markiert das Framework den Test als fehlgeschlagen und zeigt die fehlerhafte Zeile an.

Aufräumen:

Optional wird `tearDown()` ausgeführt, um Ressourcen freizugeben oder Zustände zurückzusetzen.

Testergebnis:

Nach Ausführung aller Tests liefert das Framework eine Übersicht über bestandene und fehlgeschlagene Tests. Bei Unity liefert `UNITY_END()` zusätzlich die Anzahl fehlgeschlagener Tests als Rückgabewert.

4. Ziel von Unit-Tests

- Fehler früh erkennen und isoliert lokalisieren
- Reproduzierbare Tests für jede Funktion bereitstellen
- Grundlage für kontinuierliche Integration und automatisierte Tests schaffen

5. Einführung in Unity

Unity ist ein leichtgewichtiges C-Unit-Testframework, das häufig in eingebetteten Umgebungen verwendet wird. Es bietet einfache Assertion-Makros, eine klare Teststruktur und lässt sich problemlos in Continuous-Integration-Umgebungen integrieren.

Unity selbst enthält keine Mock-Engine. Mocking bezeichnet das Ersetzen von echten Abhängigkeiten eines Moduls durch simulierte Versionen (sogenannte Mocks), um das Verhalten einer Funktion oder eines Moduls isoliert testen zu können. Dadurch lassen sich z. B. externe Bibliotheken, Hardwarezugriffe oder komplexe Systemkomponenten kontrolliert simulieren, ohne dass das gesamte System laufen muss.

Wichtige Eigenschaften

- Sehr klein und portabel – läuft auch auf ressourcenbeschränkten Systemen.
- Einfache Assertion-Makros: `TEST_ASSERT_EQUAL_*`, `TEST_ASSERT_TRUE`, `TEST_ASSERT_NULL`, u.a.
- `setUp()`/`tearDown()`-Hooks, die vor bzw. nach jedem Test ausgeführt werden.
- Tests werden als einzelne Funktionen geschrieben und mit `RUN_TEST()` registriert.
- Keine eingebauten Mocks (CMock ergänzt diese Funktionalität).

Grundstruktur eines Unity-Tests

Minimalbeispiel (`test_add.c`):

```c
#include "unity.h"

/* Funktion unter Test */
int add(int a, int b) { return a + b; }

/* optional: wird vor jedem Test aufgerufen */
void setUp(void) {}

/* optional: wird nach jedem Test aufgerufen */
void tearDown(void) {}
```

```c
/* einzelne Testfunktionen */
static void test_add_positive(void)
{
    TEST_ASSERT_EQUAL_INT(5, add(2, 3));   /* expected, actual */
}

static void test_add_negative(void)
{
    TEST_ASSERT_EQUAL_INT(0, add(-1, 1));
}

/* Test-Runner */
int main(void)
{
    UNITY_BEGIN();
    RUN_TEST(test_add_positive);
    RUN_TEST(test_add_negative);
    return UNITY_END(); /* liefert Anzahl der fehlgeschlagenen Tests */
}
```

Kompilieren / Ausführen

Unity wird häufig als `unity.c`/`unity.h` in das Testprojekt aufgenommen. Kompilieren z. B.:

```
gcc -o test_add test_add.c unity.c -I/path/to/unity
./test_add
```

- `UNITY_END()` liefert die Zahl der fehlgeschlagenen Tests als Rückgabewert (non-zero → Fehler).

Assertions — Übersicht

- `TEST_ASSERT_EQUAL_INT(expected, actual)`– prüft Integer-Gleichheit
- `TEST_ASSERT_TRUE(expr) / TEST_ASSERT_FALSE(expr)`
- `TEST_ASSERT_EQUAL_MEMORY(expected, actual, len)`– byte-weise Vergleich
- `TEST_ASSERT_NULL(ptr) / TEST_ASSERT_NOT_NULL(ptr)` (Es gibt viele Varianten, komplette Liste in `unity.h`)

Mocking

Unity selbst enthält keine Mock-Engine. Für automatisches Mocking wird CMock verwendet. Es erzeugt Mock-Header aus bestehenden Interfaces und ermöglicht so die Simulation von Abhängigkeiten in Unit-Tests. CMock wird oft zusammen mit Ceedling eingesetzt, einem Build- und Test-Management-Tool für C.

Zusammenfassung

- Testing ist ein zentrales Element der Softwareentwicklung, um Fehler frühzeitig zu erkennen und Wartbarkeit sicherzustellen.

- Unity ist ein leichtgewichtiges Framework für Unit-Tests in C, das Assertions, Testorganisation und einfache Teststrukturen bietet.
- Für automatisiertes Mocking wird CMock ergänzend eingesetzt.

Die konsequente Nutzung von Unit-Tests mit Unity und ergänzenden Tools wie CMock trägt dazu bei, C-Programme robuster, zuverlässiger und leichter wartbar zu entwickeln.