

Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Tobias Klaus, Florian Franzmann, Martin Hoffmann

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
<http://www4.cs.fau.de>

4. Juni 2014



- 1 C-Quiz Teil V
- 2 Testen
- 3 Übungsaufgabe



- C99
- x86 bzw. x86-64, d. h.
 - vorzeichenbehaftete Integer als Zweierkomplement implementiert
 - char hat 8 Bit
 - short hat 16 Bit
 - int hat 32 Bit
 - long hat 32 Bit auf x86 und 64 Bit auf x86-64



Frage 16

Angenommen x hat Typ `int`. Ist $x + 1 \dots$

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von x ?

Erklärung

- nicht definiert genau dann, wenn `INT_MAX`



Frage 17

Angenommen x hat Typ `int`. Ist $x - 1 + 1 \dots$

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von x ?

Erklärung

- additive Operatoren sind linksassoziativ
- ⇒ nicht definiert für `INT_MIN`



Frage 18

Angenommen x hat Typ `int`. Ist `(short)x + 1 ...`

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von x ?

Erklärung

- wenn x nicht in `short` passt
 ↪ implementierungsabhängig



Frage 19

Angenommen x hat Typ `int`. Ist `(short)(x + 1) ...`

1. definiert für alle Werte
2. definiert für manche Werte
3. definiert für keinen Wert

von x ?

Erklärung

- wenn $x + 1$ nicht in `short` passt
 - ↪ implementierungsabhängig
- die meisten Compiler schneiden beim Cast ab



Frage 20

Hat die Auswertung von `INT_MIN % -1` definiertes Verhalten?

1. ja
2. nein
3. das weiß niemand ...

Erklärung

- C99 macht dazu keine Aussage
- in C11 gilt folgendes:
 - wenn $(a/b)*b + a\%b$ darstellbar ist, haben a/b und $a\%b$ definiertes Verhalten
 - sonst nicht
 - $\text{INT_MIN} / -1$ entspricht $\text{INT_MAX} + 1$
 - was auf x86/x86-64 nicht darstellbar ist



<http://blog.regehr.org/archives/721>



1 C-Quiz Teil V

2 Testen

3 Übungsaufgabe



- Erste Grundregeln:
 - Testbarkeit von vornherein einplanen
 - ↪ Feingranulare Testfälle
 - ↪ *Einzelne Testfälle für einzelne Funktionen!*
 - Teste Datentypen an ihren Wertebereichsgrenzen
 - INT16_MAX, INT16_MIN, ...
 - *zumindest* den gesamten erreichbaren Code abdecken.
- Hilfsmittel:
 - Automatisierte Testinfrastruktur
 - Code-Coverage-Analysewerkzeug

Vorsicht!

- Testfälle können nur die Anwesenheit von Fehlern zeigen
- Nicht deren Abwesenheit! (→ vgl. Verifikation)
- ↪ Alle *Randfälle* erkennen und abdecken



- unterstützt die Integration von Tests im Softwareprojekt
- Automatisierte Ausführung und Auswertung von Testläufen
- Konfigurationsdatei: `tests/CMakeLists.txt`
 - Ausführbares Target:
`add_executable(plus_test plus_test.c)`
 - Hinzubinden der zu testenden Bibliothek:
`target_link_libraries(plus_test mathe)`
 - Bekanntmachen als Testfall:
`add_test(MatheTest_PLUS plus_test)`
- `make test` führt Tests aus
- Automatische Testauswertung:
 - Anhand Rückgabewert (0 → OK, -1 → Fehler)
 - Notfalls auch Parsen von Ausgaben



- Werkzeug aus der gcc-Toolchain
 - Instrumentierung des Binärcodes
 - Protokollieren der Programmausführung
 - Wie oft wird jede Codezeile ausgeführt?
 - Welche Zeilen werden überhaupt ausgeführt?
 - Welche Verzweigungen wurden genommen?
 - HTML Ausgabe: lcov
- Tests solange verfeinern, bis alles überdeckt ist!



Aufdecken von Laufzeitfehlern

- In dieser Übung: clang Sanitizer
 - Wird von unseren cmake-Skripten automatisch verwendet, wenn
 - Debugging aktiviert ist
 - und clang als Compiler verwendet wird
 - siehe cmake/sanitizer.cmake
 - *im CIP-Pool erst addpackage clang ausführen*
- ↪ `CC=clang CXX=clang++ cmake -DCMAKE_BUILD_TYPE=Debug ..`
- entdeckt z. B.
 - falsche Verwendung von Zeigern
 - nicht-definierte Integer-Operationen
 - lesen nichtinitialisierten Speichers
 - Integer-Überlauf

Entdeckt Fehler ...

... nur, wenn die verwendeten Testfälle diese auslösen.



Beispiel

■ Quellverzeichnis

```
% tree ~/source
~/source
|-- CMakeLists.txt
|-- include
|   |-- mathe.h
|-- src
|   |-- CMakeLists.txt
|   |-- abs.c
|   |-- plusminus.c
'-- tests
    |-- CMakeLists.txt
    |-- abs_test.c
    '-- plus_test.c
```

■ Binärverzeichnis

```
% cd ~/binary
% cmake ../
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Checking whether C compiler has -isysroot
...
-- Configuring done
-- Generating done
-- Build files have been written to: ~/build
% make
[ 20%] Building C object src/CMakeFiles/mathe.dir/plusminus.c.o
[ 40%] Building C object src/CMakeFiles/mathe.dir/abs.c.o
Linking C static library libmathe.a
[ 60%] Built target mathe
Scanning dependencies of target abs_test
[ 80%] Building C object tests/CMakeFiles/abs_test.dir/abs_test.c.o
Linking C executable abs_test
[ 80%] Built target abs_test
Scanning dependencies of target plus_test
[100%] Building C object tests/CMakeFiles/plus_test.dir/plus_test.c.o
Linking C executable plus_test
[100%] Built target plus_test
% make test
Running tests...
Test project ~/build
  Start 1: MatheTest_PLUS
1/2 Test #1: MatheTest_PLUS ..... Passed    0.00 sec
  Start 2: MatheTest_ABS
2/2 Test #2: MatheTest_ABS .....***Failed    0.00 sec
50% tests passed, 1 tests failed out of 2

Total Test time (real) =  0.02 sec

The following tests FAILED:
   2 - MatheTest_ABS (Failed)
Errors while running CTest
```



- 1 C-Quiz Teil V
- 2 Testen
- 3 Übungsaufgabe**



1. Objektbasierter Softwareentwurf
 2. Testfallentwurf
 - Vollständige Pfadeüberdeckung
 - Abdecken aller Randfälle
 3. Implementierung von Software und Testfällen
 - Getrennte Implementierung von Software und Testfällen
 - Möglichst durch verschiedene Übungsteilnehmer
- Implementiert werden soll eine *Prioritätswarteschlange*
 - einfügen, entfernen, *iterieren*
- ↪ `for (... x = ...; x != ...; ++x){...}`
- Implementierung?



- Datenstruktur als Array im Header vereinbaren
- Zugriff durch Zeigerarithmetik

```
1 typedef struct Element { ... } Element;  
2 Element elements[ELEMENTS_SIZE];  
3 ...  
4     for (size_t i = 0; i < ELEMENTS_SIZE; ++i)  
5         { use(elements[i]); }
```

- *Vorteile:*

- Einfache Implementierung
- Für den Compiler leicht zu optimieren

- *Nachteil:* Implementierung offengelegt

↪ Verpflichtung ggü. Benutzer



■ Iterator als Teil des Objekts

■ Header:

```
1 typedef struct Elements Elements;  
2 void El_reset_iterator(Elements *self);  
3 void El_next(Elements *self);  
4 bool El_isAtEnd(Elements *self);  
5 int64_t El_iterator_value(Elements *self);
```

■ Verwendung:

```
1 ...  
2 El_reset_iterator(dings);  
3 while(!El_isAtEnd(dings)) {  
4     use(El_iterator_value(dings));  
5     El_next(dings);  
6 }
```



Iterator – 2. Versuch

■ Implementierung:

```
1 typedef struct Element { int64_t value; } Element;
2 struct Elements {
3     Element elements[ELEMENTS_SIZE];
4     Element *it;
5 };
6 void El_reset_iterator(Elements *self)
7     { self->it = &self->elements }
8 void El_next(Elements *self)
9     { self->it = self->it + 1; }
10 bool El_isAtEnd(Elements *self)
11     { return self->it
12         == &(self->elements[ELEMENTS_SIZE]); }
13 int64_t El_iterator_value(Elements *self)
14     { return self->it->value; }
```

■ *Vorteil:* Kapselung sehr gut

■ *Nachteile:*

- Für den Compiler evt. nicht mehr optimierbar
- So nur ein Iterator gleichzeitig möglich



■ Iterator als eigenes Objekt

■ Header:

```
1 typedef struct Elements Elements;
2 typedef struct El_Iterator El_Iterator;
3
4 El_Iterator *El_begin(Elements *self);
5 void El_Iterator_destroy(El_Iterator *self);
6 void El_Iterator_next(El_Iterator *self);
7 bool El_Iterator_isAtEnd(El_Iterator *self);
8 int64_t El_Iterator_value(El_Iterator *self);
```

■ Verwendung:

```
1 ...
2 El_Iterator *it;
3 for (it = El_begin(dings);
4      not El_Iterator_isAtEnd(it);
5      El_Iterator_next(it)) {
6     use(El_Iterator_value(it))
7 }
8 El_Iterator_destroy(it);
9
```



■ Implementierung:

```
1 typedef struct Element { int64_t value; } Element;
2 struct Elements { Element elements[ELEMENTS_SIZE]; };
3 struct El_Iterator {
4     Element *position;
5     Element *end;
6 };
7 El_Iterator *El_begin(Elements *self) {
8     El_Iterator *ret = malloc(sizeof(El_Iterator));
9     if (ret == NULL) { return NULL; }
10    ret->position = self->elements;
11    ret->end = &self->elements[ELEMENTS_SIZE];
12    return ret;
13 }
14 void El_Iterator_next(El_Iterator *self)
15     { self->position += 1; }
16 bool El_Iterator_isAtEnd(El_Iterator *self) { ... }
17 int64_t El_Iterator_value(El_Iterator *self) { ... }
18 void El_Iterator_destroy(El_Iterator *self) { ... }
```



- *Vorteile:*

- Vollständige Kapselung
- Beliebig viele Iteratoren möglich

- *Nachteil:*

- Iterator muss nach Gebrauch beseitigt werden
- Compiler hat evt. Probleme zu optimieren

