

Verlässliche Echtzeitsysteme

Übungen zur Vorlesung

Testen

Tobias Klaus, Florian Schmaus, Peter Wägemann

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

6. Juni 2016



[https://www4.cs.fau.de/Lehre/SS16/V_VEZS/Uebung/
dashboard/fail.html](https://www4.cs.fau.de/Lehre/SS16/V_VEZS/Uebung/dashboard/fail.html)

 **Deadline: 20.06.2016**



- 1 C-Quiz Teil V
- 2 Abfangen von Integer-Fehlern
- 3 Testen
- 4 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 13

Angenommen x hat Typ **short**. Ist $x < 29 \dots$

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

von x ?



Frage 13

Angenommen x hat Typ **short**. Ist $x \ll 29 \dots$

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

von x ?

Erklärung

- Vor der Verschiebeoperation wird nach **int** umgewandelt
- Verschiebung um mehr als die Bitbreite ist also kein Problem



Frage 14

Angenommen x hat Typ **unsigned**. Ist $x \ll 31 \dots$

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

von x ?



Frage 14

Angenommen `x` hat Typ **unsigned**. Ist `x << 31 ...`

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

von `x`?

Erklärung

- jeder Wert, dessen *promoted type* **unsigned** ist kann um nichtnegativen Wert verschoben werden
- solange die Bitbreite nicht erreicht wird



Frage 15

Angenommen x hat Typ **unsigned short**. Ist $x \ll 31 \dots$

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

von x ?



Frage 15

Angenommen `x` hat Typ **unsigned short**. Ist `x << 31 ...`

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

von `x`?

Erklärung

- **unsigned short** wird nach **int** umgewandelt
- eine 1 darf nicht in das Vorzeichenbit hineinverschoben werden
- Verschiebung um bis zu 15 wäre immer in Ordnung



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 ?



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



1 C-Quiz Teil V

2 Abfangen von Integer-Fehlern

3 Testen

4 Übungsaufgabe



Table of Contents

1 C-Quiz Teil V

2 Abfangen von Integer-Fehlern

3 Testen

4 Übungsaufgabe



- C bietet viele subtile Fehlermöglichkeiten
 - Im C-Quiz haben wir einige kennengelernt
 - Was uns noch fehlt:
 - *Wie verhält man sich als Programmierer richtig?*
- ~> Heute ein paar Beispiele



Was soll da schon schiefgehen...

```
1 unsigned int func(unsigned int a, unsigned int b) {  
2     return a + b;  
3 }
```



Was soll da schon schiefgehen...

```
1 unsigned int func(unsigned int a, unsigned int b) {  
2     return a + b;  
3 }
```

Vorbedingungstest

```
1 #include <limits.h>  
2 unsigned int func(unsigned int a, unsigned int b) {  
3     if (UINT_MAX - a < b) { raise("wraparound"); }  
4     return a + b;  
5 }
```

Nachbedingungstest

```
1 unsigned int func(unsigned int a, unsigned int b) {  
2     unsigned int ret = a + b;  
3     if (ret < a) { raise("wraparound"); }  
4     return ret;  
5 }
```



Was soll da schon schiefgehen...

```
1 unsigned int func(unsigned int a, unsigned int b) {  
2     return a - b;  
3 }
```



Was soll da schon schiefgehen...

```
1 unsigned int func(unsigned int a, unsigned int b) {  
2     return a - b;  
3 }
```

Vorbedingungstest

```
1 unsigned int func(unsigned int a, unsigned int b) {  
2     if (a < b) { raise("wraparound"); }  
3     return a - b;  
4 }
```

Nachbedingungstest

```
1 unsigned int func(unsigned int a, unsigned int b) {  
2     unsigned int ret = a - b;  
3     if (ret > a) { raise("wraparound"); }  
4     return ret;  
5 }
```



Was soll da schon schiefgehen...

```
1 unsigned int func(unsigned int a, unsigned int b) {  
2     return a * b;  
3 }
```



Was soll da schon schiefgehen...

```
1 unsigned int func(unsigned int a, unsigned int b) {  
2     return a * b;  
3 }
```

Vorbedingungstest

```
1 #include <limits.h>  
2 unsigned int func(unsigned int a, unsigned int b) {  
3     if (a == 0 or b == 0) { return 0; }  
4     if (UINT_MAX / a < b) { raise("wraparound"); }  
5     return a * b;  
6 }
```



Was soll da schon schiefgehen...

```
1 unsigned int func(signed int a) {  
2     return (unsigned int) a; /* keine Compilerwarnung wg. Cast */  
3 }
```



Was soll da schon schiefgehen...

```
1 unsigned int func(signed int a) {  
2     return (unsigned int) a; /* keine Compilerwarnung wg. Cast */  
3 }
```

Vorbedingungstest

```
1 unsigned int func(signed int a) {  
2     if (a < 0) { raise("wraparound"); }  
3     return (unsigned int) a;  
4 }
```



Was soll da schon schiefgehen...

```
1 unsigned char func(unsigned long int a) {  
2     return (unsigned char) a; /* keine Compilerwarnung wg. Cast */  
3 }
```



Was soll da schon schiefgehen...

```
1 unsigned char func(unsigned long int a) {  
2     return (unsigned char) a; /* keine Compilerwarnung wg. Cast */  
3 }
```

Vorbedingungstest

```
1 unsigned char func(unsigned long int a) {  
2     if (a > UCHAR_MAX) { raise("overflow"); }  
3     return (unsigned char) a; /* keine Compilerwarnung wg. Cast */  
4 }
```



Was soll da schon schiefgehen...

```
1 signed char func(unsigned long int a) {  
2     return (signed char) a; /* keine Compilerwarnung wg. Cast */  
3 }
```



Was soll da schon schiefgehen...

```
1 signed char func(unsigned long int a) {  
2     return (signed char) a; /* keine Compilerwarnung wg. Cast */  
3 }
```

Vorbedingungstest

```
1 #include <limits.h>  
2 signed char func(unsigned long int a) {  
3     if (a > SCHAR_MAX) { raise("overflow"); }  
4     return (signed char) a;  
5 }
```



Was soll da schon schiefgehen...

```
1 signed char func(signed long int a) {  
2     return (signed char) a; /* keine Compilerwarnung wg. Cast */  
3 }
```



Was soll da schon schiefgehen...

```
1 signed char func(signed long int a) {  
2     return (signed char) a; /* keine Compilerwarnung wg. Cast */  
3 }
```

Vorbedingungstest

```
1 #include <iso646.h>  
2 #include <limits.h>  
3 signed char func(signed long int a) {  
4     if (a < SCHAR_MIN or SCHAR_MAX < a) { raise("overflow"); }  
5     return (signed char) a; /* keine Compilerwarnung wg. Cast */  
6 }
```



Was soll da schon schiefgehen...

```
1 signed int func(signed int a, signed int b) {  
2     return a + b;  
3 }
```



Was soll da schon schiefgehen...

```
1 signed int func(signed int a, signed int b) {  
2     return a + b;  
3 }
```

Vorbedingungstest

```
1 #include <iso646.h>  
2 #include <limits.h>  
3 signed int func(signed int a, signed int b) {  
4     if ((b > 0 and a > INT_MAX - b)  
5         or (b < 0 and a < (INT_MIN - b))) { raise("overflow"); }  
6     return a + b;  
7 }
```



Was soll da schon schiefgehen...

```
1 signed int func(signed int a, signed int b) {  
2     return a - b;  
3 }
```



Was soll da schon schiefgehen...

```
1 signed int func(signed int a, signed int b) {  
2     return a - b;  
3 }
```

Vorbedingungstest

```
1 #include <iso646.h>  
2 #include <limits.h>  
3 signed int func(signed int a, signed int b) {  
4     if ((b > 0 and a < INT_MIN + b)  
5         or (b < 0 and a > INT_MAX + b)) { raise("overflow"); }  
6     return a - b;  
7 }
```



Was soll da schon schiefgehen...

```
1 signed long func(signed long a, signed long b) {  
2     return a / b;  
3 }
```



Was soll da schon schiefgehen...

```
1 signed long func(signed long a, signed long b) {  
2     return a / b;  
3 }
```

Vorbedingungstest

```
1 #include <iso646.h>  
2 #include <limits.h>  
3 signed long func(signed long a, signed long b) {  
4     if (b == 0) { raise("division by 0"); }  
5     return a / b;  
6 }
```



- Reicht das schon?

Was soll da schon schiefgehen...

```
1 signed long func(signed long a, signed long b) {  
2     if (b == 0) { raise("division by 0"); }  
3     return a / b;  
4 }
```



■ Reicht das schon?

Was soll da schon schiefgehen...

```
1 signed long func(signed long a, signed long b) {  
2     if (b == 0) { raise("division by 0"); }  
3     return a / b;  
4 }
```

Vorbedingungstest

```
1 #include <iso646.h>  
2 #include <limits.h>  
3 signed long func(signed long a, signed long b) {  
4     if (b == 0) { raise("division by zero"); }  
5     if (a == LONG_MIN and b == -1) { raise("overflow"); }  
6     return a / b;  
7 }
```



Was soll da schon schiefgehen...

```
1 signed long func(signed long a, signed long b) {  
2     return a % b;  
3 }
```



Was soll da schon schiefgehen...

```
1 signed long func(signed long a, signed long b) {  
2     return a % b;  
3 }
```

Vorbedingungstest

```
1 #include <iso646.h>  
2 #include <limits.h>  
3 signed long func(signed long a, signed long b) {  
4     if (b == 0) { raise("division by zero"); }  
5     if (a == LONG_MIN and b == -1) { raise("overflow"); }  
6     return a % b;  
7 }
```



Was soll da schon schiefgehen...

```
1 signed long func(signed long a) {  
2     return -a;  
3 }
```



Was soll da schon schiefgehen...

```
1 signed long func(signed long a) {  
2     return -a;  
3 }
```

Vorbedingungstest

```
1 #include <limits.h>  
2 signed long func(signed long a) {  
3     if (a == LONG_MIN) { raise("overflow"); }  
4     return -a;  
5 }
```



Was soll da schon schiefgehen...

```
1 signed int func(signed int a, signed int b) {  
2     return a * b;  
3 }
```



Was soll da schon schiefgehen...

```
1 signed int func(signed int a, signed int b) {  
2     return a * b;  
3 }
```

Vorbedingungstest

```
1 #include <iso646.h>  
2 #include <limits.h>  
3 signed int func(signed int a, signed int b) {  
4     if (a == 0 or b == 0) { return 0; }  
5     if (a > 0 and b > 0 and a > INT_MAX / b) { raise("overflow"); }  
6     if (a > 0 and b < 0 and b < INT_MIN / a) { raise("overflow"); }  
7     if (a < 0 and b > 0 and a < INT_MIN / b) { raise("overflow"); }  
8     if (a < 0 and b < 0 and b < INT_MAX / a) { raise("overflow"); }  
9     return a * b;  
10 }
```



1 C-Quiz Teil V

2 Abfangen von Integer-Fehlern

3 Testen

4 Übungsaufgabe



- Erste Grundregeln:
 - Testbarkeit von vornherein einplanen
 - ↪ Feingranulare Testfälle
 - ↪ *Ein Testfall für jede einzelne Funktion!*
 - Teste Datentypen an ihren Wertebereichsgrenzen
 - INT16_MAX, INT16_MIN, ...
 - *Minimale Testabdeckung*: erreichbarer Code/Zeilenüberdeckung
- 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





- 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)`
- Ausführung der Tests: `make test`
- Automatische Testauswertung:
 - Anhand Rückgabewert (0 → OK, -1 → Fehler)
 - Notfalls auch Parsen von Ausgaben



Codeüberdeckung: gcov/lcov

Current view: [top level](#) - src

Test: coverage.lcov

Date: 2016-06-05 19:42:10

Legend: Rating: low: < 75 % medium: >= 75 % high: >= 90 %

	Hit	Total	Coverage
Lines:	6	91	6.6 %
Functions:	1	13	7.7 %

Filename	Line Coverage (show details) ⬆	Functions ⬆
priority_queue.c	6.6 % 6 / 91	7.7 % 1 / 13

- Werkzeug aus der gcc-Toolchain
- Instrumentierung des Binärcodes $\leadsto$ *Laufzeitkosten*
- Protokollieren der Programmausführung
 - Wie oft wird jede Codezeile ausgeführt?
 - Welche Zeilen werden überhaupt ausgeführt?
 - Welche Verzweigungen wurden genommen?
- HTML Ausgabe: lcov

→ Tests solange erweitern, bis *vollständige Zeilenüberdeckung* erreicht!



- „Im besten Fall kracht es bei Speicherzugriffsfehlern!“
- In Übungen: Verwendung von Clang AddressSanitizer [1]¹
- Checks zur Laufzeit
 - falsche Verwendung von Zeigern
 - nicht-definierte Integer-Operationen
 - Lesen uninitialisierten Speichers
 - Integer-Überlauf
 - ...

Entdeckt Fehler ...

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

☞ **zur Laufzeit**

- Laufzeitkosten: $\approx 2 \times$

¹<http://clang.llvm.org/docs/AddressSanitizer.html>

Clang AddressSanitizer – Verwendung

```
1 // program.cpp
2 int main(int argc, char **argv) {
3     int *array = new int[100];
4     delete [] array;
5     return array[argc]; // BOOM
6 }
```

```
$ clang -O1 -g -fsanitize=address program.cpp
```

```
$ ./a.out
```

ERROR: AddressSanitizer: heap-use-after-free on address 0x602e0001fc64 at pc ...

- Wird von cmake-Skripten automatisch verwendet, wenn
 - Debugging aktiviert ist
 - und clang als Compiler verwendet wird
 - siehe cmake/sanitizer.cmake
- Aktivierung im CIP-Pool (im Terminal ausführen):
 - *Im CIP-Pool erst addpackage clang im Terminal ausführen*
 - Aufruf von cmake

→ CC=clang CXX=clang++ cmake -DCMAKE_BUILD_TYPE=Debug ..



- Analyse des Quellcodes (C, C++, Objective-C)
- Keine Ausführung des Codes auf Hardware $\leadsto$ „statische Analyse“
- Eingabewerte als *symbolisch* angenommen $\rightarrow$ *Symbolische Ausführung*
- Verfügbare Checks²
 - Wertebereichsanalysen: Division mit Null
 - Verwendung uninitialisierter Variablen
 - ...
- Analyse ist nicht fehlerfrei (engl. sound)
 - Nicht möglich alle Fehler zu finden (engl. false negatives)
 - Falsch positive Befunde mögliche (engl. false positives)

²http://clang-analyzer.llvm.org/available_checks.html

Clang Static Analyzer – Verwendung

```
1 void test() {  
2   int i, a[10];  
3   int x = a[i]; // warn: array subscript is undefined  
4 }
```

1 'i' declared without an initial value →

2 ← Array subscript is undefined

- Einzelne Datei überprüfen: `scan-build clang -c program.c`
- Übung: Aufruf von `scan-build` mit `cmake` als Argument
→ `CC=clang CXX=clang++ scan-build cmake -DCMAKE_BUILD_TYPE=Debug ..`
- Fehler/Warnungen gefunden → Ausgabe von HTML Dateien
- Aufruf von `scan-view` wie in Ausgabe beschrieben



1 C-Quiz Teil V

2 Abfangen von Integer-Fehlern

3 Testen

4 Übungsaufgabe



- Verwendung von GNU/Linux (kein eCos mehr)
- Ziele
 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*



- Verwendung von GNU/Linux (kein eCos mehr)
- Ziele
 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]); }
```



- 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 offen gelegt
~> 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 El_reset_iterator(dings);  
2 while(!El_isAtEnd(dings)) {  
3     use(El_iterator_value(dings));  
4     El_next(dings);  
5 }
```



■ 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; }
```



■ 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 evtl. nicht mehr optimierbar (Schleife ausrollen)
- 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 El_Iterator *it;
2 for (it = El_begin(dings);
3      not El_Iterator_isAtEnd(it);
4      El_Iterator_next(it)) {
5     use(El_Iterator_value(it))
6 }
7 El_Iterator_destroy(it);
```



■ 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
8 El_Iterator *El_begin(Elements *self) {
9     El_Iterator *ret = malloc(sizeof(El_Iterator));
10    if (ret == NULL) { return NULL; }
11    ret->position = self->elements;
12    ret->end = &self->elements[ELEMENTS_SIZE];
13    return ret;
14 }
15
16 void El_Iterator_next(El_Iterator *self)
17     { self->position += 1; }
18 bool El_Iterator_isAtEnd(El_Iterator *self) { ... }
19 int64_t El_Iterator_value(El_Iterator *self) { ... }
20 void El_Iterator_destroy(El_Iterator *self) { ... }
```



■ *Vorteile:*

- Vollständige Kapselung
- Beliebig viele Iteratoren möglich

■ *Nachteil:*

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



42





Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov.
AddressSanitizer: A fast address sanity checker.
In Proceedings of the USENIX Annual Technical Conference, pages 309–318, 2012.

