Typische Fehler in C

```
1 int i;
2 if ( i != 0 ) {
3     printf( "i ist nicht 0.\n" );
4 }
```

Typische Fehler in C

```
1 int i;
2 if ( i != 0 ) {
3     printf( "i ist nicht 0.\n" );
4 }
```

- Fehler: nicht initialisierte Variable; korrekt wäre:

```
1 int i = 0;
2 if ( i != 0 ) {
3     printf( "i ist nicht 0.\n" );
4 }
```

- Im Gegensatz zu Java warnt der gcc unter Anderem nicht vor uninitialisierten Variablen. Abhilfe schafft die Compiler-Option `-Wall`.

Typische Fehler in C

```
1  void count( void ) {
2      int k, i = 0;
3
4      for( i = 0; i < 10; i++ )
5      {
6          k = k + 1;
7      }
8
9      printf("%d", k);
10 }
```

Typische Fehler in C

```
1  void count( void ) {
2      int k, i = 0;
3
4      for( i = 0; i < 10; i++ )
5      {
6          k = k + 1;
7      }
8
9      printf("%d", k);
10 }
```

- Fehler: nur i wird mit 0 intitialisiert

Typische Fehler in C

```
1 int a = 0;
2 int b = 0;
3 int sum = a + b;
4 a=getA();
5 b=getB();
6 printf( "sum: %d", sum );
```

Typische Fehler in C

```
1 int a = 0;
2 int b = 0;
3 int sum = a + b;
4 a=getA();
5 b=getB();
6 printf( "sum: %d", sum );
```

- Fehler: sum uninitialisiert
- Variablen lassen sich nicht verknpüfen, aber ihre Belegungen

Typische Fehler in C

```
1 #include <stdio.h>
2 int main() {
3  int a = 0;
4  scanf( "%d", a );
5  printf( "a: %d", a );
6  return 0;
7 }
```

Typische Fehler in C

```
1 #include <stdio.h>
2 int main() {
3  int a = 0;
4  scanf( "%d", a );
5  printf( "a: %d", a );
6  return 0;
7 }
```

- Fehler: bei scanf & Operator für a vergessen!

Typische Fehler in C

```
1 char c;
2 scanf( "%c", &c );
3 if ( c = 'a' ) {
4   ...
5 }
```

Typische Fehler in C

```
1 char c;
2 scanf( "%c", &c );
3 if ( c = 'a' ) {
4   ...
5 }
```

- Fehler: Zuweisung mit “=” statt Vergleich mit “==”; korrekt wäre:

```
1 char c;
2 scanf( "%c", &c );
3 if ( c == 'a' ) {
4   ...
5 }
```

Typische Fehler in C

```
1 double x = 3/2;
```

Typische Fehler in C

```
1 double x = 3/2;
```

- Fehler: 3 und 2 sind integer Literale, daher ist das Ergebnis auch Integer
- Float oder double Literale nutzen

Typische Fehler in C

```
1  long int      a = -1;
2  unsigned int b = 1;
3  printf ("%d\n", a > b);
```

Typische Fehler in C

```
1   long int      a = -1;
2   unsigned int b = 1;
3   printf ("%d\n", a > b);
```

- Fehler: signed vs. unsigned Vergleiche sind (evtl.) architekturabhängig. Auf 64-Bit Systemen ok, aber auf 32-Bit Systemen falsch.
- Lösung: Explizite Casts

Typische Fehler in C

```
1 int k = 0;
2 for( int i = 0; i < 10; ++i ) {
3   k =+ 1;
4 }
```

Typische Fehler in C

```
1 int k = 0;
2 for( int i = 0; i < 10; ++i ) {
3   k =+ 1;
4 }
```

- =+ Sind zwei Operatoren, Zuweisung und unäres Plus

Typische Fehler in C

```
1 #define VALUE 5;
2
3 int main() {
4   int i = VALUE + 1;
5   printf( "%d\n", i );
6   return 0;
7 }
```

Typische Fehler in C

```
1 #define VALUE 5;
2
3 int main() {
4   int i = VALUE + 1;
5   printf( "%d\n", i );
6   return 0;
7 }
```

- Fehler: Semikolon am Ende zerteilt Zuweisung in zwei Stmts

Typische Fehler in C

```
1 // 10 mal Hallo Welt ausgeben
2 for( int i = 0; i < 10; i++ );
3     printf( "Hallo Welt!\n" );
```

Typische Fehler in C

```
1 // 10 mal Hallo Welt ausgeben
2 for( int i = 0; i < 10; i++ );
3     printf( "Hallo Welt!\n" );
```

- Fehler: Semikolon zu viel; korrekt wäre:

```
1 // 10 mal Hallo Welt ausgeben
2 for( int i = 0; i < 10; i++ )
3     printf( "Hallo Welt!\n" );
```

Typische Fehler in C

```
1 struct foo {
2   int x;
3 }
4
5 f()
6 {
7   //..
8 }
```

Typische Fehler in C

```
1 struct foo {
2  int x;
3 }
4
5 f()
6 {
7  //..
8 }
```

- Fehler: Fehlendes Semikolon am Ende der struct-Deklaration

Typische Fehler in C

```
1 int value;
2 do
3 {
4   //...
5   value=10;
6 }while(!(value==10) || !(value==20));
```

Typische Fehler in C

```
1 int value;
2 do
3 {
4   //...
5   value=10;
6 }while(!(value==10) || !(value==20));
```

- Fehler: value kann nicht zugleich 10 und 20 sein

```
1 int value;
2 do
3 {
4   //...
5   value=10;
6 }while(!(value==10) && !(value==20));
```

Typische Fehler in C

```
1 // führe Schleife 12x aus
2 for( int i = 0; i < 13; ++i ) {
3   //..
4 }
```

Typische Fehler in C

```
1 // führe Schleife 12x aus
2 for( int i = 0; i < 13; ++i ) {
3   //..
4 }
```

- Fehler: off-by-one, die Schleife beginnt mit 0 und endet damit im 13. Durchlauf

Typische Fehler in C

```
1 int main()
2 {
3   menu();
4 }
5 void menu()
6 {
7   //...
8 }
```

Typische Fehler in C

```
1 int main()
2 {
3   menu();
4 }
5 void menu()
6 {
7   //...
8 }
```

- Fehler: menu() undefined
- Forward declaration

Typische Fehler in C

```
1 void f( void ) {
2   printf( "Init done...\n" );
3 }
4
5 int main() {
6   f;
7   return 0;
8 }
```

Typische Fehler in C

```
1 void f( void ) {
2   printf( "Init done...\n" );
3 }
4
5 int main() {
6   f;
7   return 0;
8 }
```

- Fehler: f wird niemals aufgerufen! Klammern nicht vergessen: f()

Typische Fehler in C

- Was sagt uns folgende Fehlermeldung?

```
/tmp/cctiJhZL.o: In function ‘norm’:
./Vektorwinkel.c:19: undefined reference to ‘sqrtf’
/tmp/cctiJhZL.o: In function ‘angle’:
./Vektorwinkel.c:37: undefined reference to ‘acosf’
collect2: error: ld returned 1 exit status
```

Typische Fehler in C

- Was sagt uns folgende Fehlermeldung?

```
/tmp/cctiJhZL.o: In function ‘norm’:
./Vektorwinkel.c:19: undefined reference to ‘sqrtf’
/tmp/cctiJhZL.o: In function ‘angle’:
./Vektorwinkel.c:37: undefined reference to ‘acosf’
collect2: error: ld returned 1 exit status
```

- `undefined reference to ...` $\widehat{=}$ Fehler beim *Linken*
- Ursache: eine Bibliothek wurde nicht gefunden oder angegeben
  - In diesem Fall die math-Bibliothek, welche mit dem Parameter `-lm` hinzugefügt wird

Fehler finden mit KDbg

- Programm mit gcc-Option im Debug Modus kompilieren: `-g`
- Achtung Kombination mit `-O` (optimieren) möglich, **ABER**:
  - Variablen existieren vielleicht nicht
  - Konstante Berechnungen werden eventuell nicht ausgeführt
  - Reihenfolge der Anweisungen neu geordnet
  - Anweisungen werden aus Schleifen entfernt
- KDbg ist eine GUI für gdb
- KDbg-Aufruf: `kdbg ./Programmname`
- Möglichkeiten von KDbg:
  - Haltepunkte und bedingte Haltpunkte
  - Variablen beobachten
  - Schrittweises Abarbeiten des Codes
  - ...

Fehler finden mit KDbg - Beispiel

- Unsere Firma *LAE-Linear Algebra Enterprises* vertreibt eine C-Mathe-Bibliothek. Es häufen sich Beschwerden, dass Vektoren nach Aufruf der Funktion `angle` oft “verändert” wurden.
- Vorgehen mit KDbg:
  1. Kompilieren:
     ```
     gcc -Wall -pedantic -std=c99 -g ./main_kdbg.c
     ./Vektorwinkel_kdbg.c -o ./main_kdbg -lm
     ```
  2. Kdbg starten:
     ```
     kdbg ./main_kdbg
     ```
  3. Felder `a[0],a[1],a[2]` und `b[0],b[1],b[2]` als *beobachtete Ausdrücke hinzufügen*
  4. Programm schrittweise abarbeiten

Fehler vermeiden

- Problem: Computer macht nicht das was wir wollen, sondern das was wir ihm “sagen”.
- Guter Programmierstil vermeidet Fehler!
  - Keine (unnötige) Code-“Optimierung”; zum Beispiel:

```
1 int i = 0;
2 while( i++ < 100 )
3 // besser: for( i = 0; i < 100; i++ )
4 {
5     printf( "%i\n", i );
6 }
```

  - Const-Correctness
  - Stylechecker
  - Versionskontrollsystem, z.B.: Mercurial, git, svn, . . .
  - Unit testing

Assertions

- “crash early”: Wenn schon ein Fehler im Programm ist, dann sollte dieser so früh wie möglich bemerkt werden.
- Assert (Zusicherung): wenn nicht erfüllt ⇒ Programmabbruch
- `assert.h` muss inkludiert werden
- Assert-Funktion: `void assert( int expression )`
  - wenn *expression* 0 ist ⇒ Programmabbruch + Fehlermeldung
  - Fehlermeldung:
    `main: ./main.c:37: proc: Assertion ‘x <= 1.0f’ failed.`
    besteht aus:
    - `main` - Programmnamen
    - `main.c` - Quellcodedatei
    - `37` - Zeile in Quellcodedatei
    - `proc` - Name der Prozedur
    - `x <= 1.0f` - Zusicherung
- Welche Zusicherung könnten wir in der Funktion `angle` benutzen?

Unit tests

- Einzelne *Units* (Codeteile) werden getestet; Units können z.B. Klassen oder Funktionen sein.

  $\Rightarrow$ Vorteil: Fehler werden früh gefunden.
- Möglichst automatisiert; bspw. bei Übernahme in Versionsverwaltung
- Tests sollten sowohl typische Fälle, als auch Grenzfälle abdecken
- Frameworks für verschiedene Programmiersprachen[5]
  - C: *CUnit, MinUnit, Check, . . .*
  - C++: *CppUnit, CxxTest*
  - Java: *JUnit, TestNG, . . .*
  - Python: *Unittest*
- Wie könnte ein Test für unsere *angle*-Funktion aussehen?

[5] Wikipedia

Versionskontrollsysteme

- Ermöglichen kollaborativ Arbeit an Quellcode
  - Je nach Komplexität geschieht die Vereinigung der Codevarianten (Mergen) automatisch oder manuell
- sind für Textdateien (*.txt, .java, .c, .h, . . .*) ausgelegt, Binärdateien (*.doc, .bmp, .png, . . .* nur als Ganzes ersetzt werden
- Fehler vermeiden durch
  - Nachvollziehbarkeit von Änderungen
  - Möglichkeit auf eine frühere funktionierende Version zu wechseln
  - Entwickeln von neuen Softwarefunktionen in eigenen Branches; Pflegen der Software in Default-Branch (auch Master-Branch genannt)
- Beispiele:
  - Client-Server System: *Subversion (SVN)*
  - Distributed Systeme: *Git, Mercurial (hg),*
    CRE130 - Verteilte Versionsverwaltung (Podcast für Interessierte)

Versionskontrolle mit Mercurial

- Mercurial, (engl. für Quecksilber)
- Befehlsübersicht:
  - `hg init .` - Repository im aktuellen Verzeichnis anlegen
  - `hg add F` - Datei `F` zur Versionskontrolle hinzufügen
  - `hg ci` - Änderungen committen
  - `hg pull` - Repository updaten
  - `hg up` - Auf neueste Revision updaten
  - `hg up -r X` - Auf Revision X updaten
  - `hg branch NEW_FEATURE` - Branch (Zweig) NEW_FEATURE anlegen
  - `hg log` - Changeset History
  - `hg log -p` - Changeset History mit Änderungen (Quelltext)
  - `hg diff .` - Unterschiede zur versionierten Version anzeigen

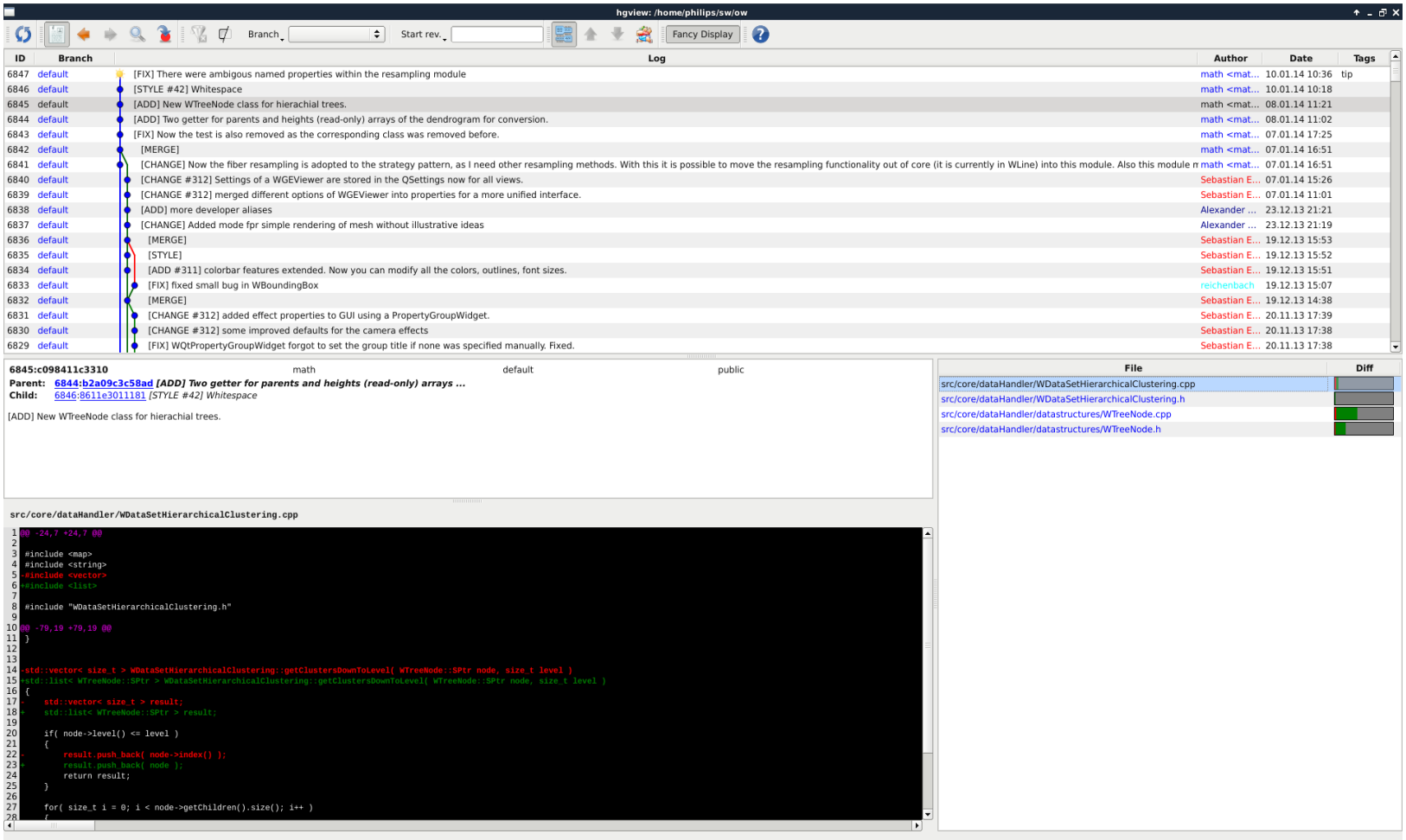

Abbildung: *hgview* - Tool zur Anzeige eines Mercurial Repository

goto fail [6]

- Fehler in Apples SSL/TLS Bibliothek
- Server-Zertifikate sollen sicherstellen, dass man mit dem “richtigen” Computer verbunden ist
  - *goto fail*-Fehler hebelt diesen Schutz aus, weil die Verifikation der Zertifikate nicht stattfindet.
- Wie kann er ausgenutzt werden?
  - Anfragen an einen zertifizierten Server werden beispielsweise in einem gehackten W-LAN an einen eigenen Server weitergeleitet.
  - Der eigene Server könnte sich z.B. als Onlinebanking-Server ausgeben.

[6] Anatomy of a "goto fail"

goto fail

```
1  static OSStatus SSLVerifySignedServerKeyExchange(SSLContext *ctx, \dots )
2  {
3      ...
4      if ((err = SSLFreeBuffer(&hashCtx)) != 0)
5          goto fail;
6      if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
7          goto fail;
8      if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
9          goto fail;
10     if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
11         goto fail;
12     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
13         goto fail;
14         goto fail;
15     if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
16         goto fail;
17
18     err = sslRawVerify(ctx, ctx->peerPubKey, dataToSign, dataToSignLen, signature,
           signatureLen);
19     if(err) {
20         sslErrorLog("KeyExchange: sslRawVerify returned %d\n", (int)err);
21         goto fail;
22     }
23
24     fail:
25     SSLFreeBuffer(&signedHashes);
26     SSLFreeBuffer(&hashCtx);
27     return err;
28 }
```

goto fail - vermeiden

- Wie hätte solch ein Fehler vermieden werden können?
  - Unit-Tests
    - Nicht trivial, andererseits bei solch wichtigen Code sehr empfehlenswert
  - Vollständige Klammerung der *If*-Anweisungen
  - Keine Verwendung von `goto`
    - Die meisten Sprachen sind ohne `goto` nicht weniger mächtig

  ⇒ KEIN `goto` IST KEIN VERLUST!
    - Java hat z.B. kein klassisches `goto`
    - “Go To Statement Considered Harmful” (1968, Edsger W. Dijkstra)

Abbildung: xkcd.org - goto

Heartbleed-Bug

- OpenSSL
  - Ermöglicht Transport Layer Security (TLS), z.B. für HTTPS
  - *Heartbeat*:
    - Soll TLS Verbindung aufrecht erhalten
    - Client sendet Paket mit sogenannter Payload (Datenpaket) und erwartet eine Antwort mit dem selben Inhalt
    - Payload soll das feststellen der *Maximum Transmission Unit* (MTU) ermöglichen
- Heartbleed - Fehler in OpenSSL 
  - Sogenannter “buffer over-read” - Speicher wird über die eigentlichen Grenzen des zugewiesenen Speichers hinaus gelesen
  - Client sendet eine Payload der Länge n, gibt aber an eine Payload mit der Länge m geschickt zu haben $m > n$

  $\Rightarrow$ Folge: Server liest über den Buffer des eigentlichen Payload hinaus und antwortet mit Speicherinhalten
  - OpenSSL verwendet eigene malloc/free-Methoden, daher ist der Speicherinhalt mit hoher Wahrscheinlichkeit interessant

Heartbleed-Bug

- Von xkcd.org: *How the Heartbleed bug works*