Systemnahe Programmierung
Vorlesung

C – Tücken und Programmierkonventionen

Peter Puschner

Institut für Technische Informatik

26/11/03

Überblick

- C und generelle Fehlerquellen beim Programmieren
- Typische Schwierigkeiten und Fehler
- Aspekte der Portabilität
- Literatur

Gründe für die Verwendung von C

- Compiler Verfügbarkeit für viele Prozessoren
  - z.T. einzig verfügbare Hochsprache
  - Portierbarkeit von Code
- Schneller Code, low-level Programmierung
- Kleinerer und schnellerer Code als bei anderen Sprachen
- Viele Code-Generatoren erzeugen C Code (z.B., Matlab/Simulink RTW, dSPACE Targetlink)

C Standard

- ISO/IEC 9899, 1999
- Definiert Syntax und Semantik der Sprache
- Definition ist nicht vollständig
  - Anhang: Portability, Unspecified behaviour
  - Anhang: Portability, Undefined behaviour

Fehlerquellen beim Programmieren

- Programmierer macht Fehler
  - Stil und Ausdrucksmöglichkeiten der Sprache
  - Robustheit der Sprache gegen Fehler
  1. Wie schwer ist es, durch einen Fehler aus einem gültigen Konstrukt ein anderes gültiges Konstrukt zu erzeugen?
  2. Fehlererkennung durch die Sprache

Fehlerquellen beim Programmieren (2)

- Missverständnisse durch Sprache
  - z.B. Operator Precedence
- Compiler macht nicht das, was der Programmierer erwartet
  - undefiniertes Sprachverhalten
- Fehler im Compiler
  - Falsche Interpretation des Standards
  - Implementierungsfehler
  - Bewusste Abweichungen vom Standard

Kategorisierung von Fehlerquellen

- Lexikalische Tücken
- Syntaktische Fallen
- Semantische Fehlinterpretationen
- Unterschiedliche C Implementierungen und Portabilität

Lexikalische Interpretation

```
if (x = y)
  foo();
```

?

```
if (x == y)
  foo();
```

```
if ((x = y) != 0)
  foo();
```

= is not ==

==

```
While (c ✗ ' ' || c == 't' || c == '\n')
    c = getc(f);
```


Lexikalische Interpretation

& and | are not && and ||

"x" is not 'x'

"string" versus ~~'string'~~

Zahlen, die mit Ziffer 0 beginnen, sind Oktalzahlen
z.B. $10 \neq 010$

Lexikalische Analyse

Umwandlung der Zeichenfolge der Eingabe in Tokens:

repeatedly bite off the biggest possible piece

`y = x/*p`	`/*` leitet einen Kommentar ein
`y = x / *p`	Division, `p` zeigt auf Divisor
`a---b`	`a -- - b`

Syntaktische Eigenheiten

- Funktionsdeklarationen und Aufrufe

  `(*(void(*)())0)();` **??**

  Declare it the way you use it

- Deklaration = Typ + Deklaratoren
- Deklarator: Ausdruck, der angegebenen Typ retourniert

Deklarationen

`int f, ((g));` `f, ((g))` liefern `int`

⇨ `f, g` … Var. vom Typ `int`

`int ff();` `ff()` ist vom Typ `int`

⇨ `ff` … Funkion, retourniert `int`

`int *g(), (*h)();` `*g() = *(g())`

⇨ `g` … Funk., retourniert Pointer auf `int`

⇨ `h` … Pointer auf Funk., die `int` liefert

⇨ `*h` … Funktion

⇨ Aufruf: `(*h)()` oder Kurzschreibweise: `h()`

Deklaration und Typkonversion (Cast)

Cast: wie Deklaration, aber ohne Variablenname, ohne Strichpunkt und in runden Klammern

- Deklaration: `int (*h)();`
- Cast: `(int (*)())`

Aufgabe: Aufruf einer Funktion, deren Adresse an der Speicheradresse 0 gespeichert ist.

- 1. Versuch: `(*0)()` `0` hat falschen Typ
- Benötigtes Objekt: `void (*fp)();`
- Lösung mit Cast: `(*(void(*)())0)();`

Vorrangregeln bei Operatoren

- Auswertereihenfolge ist nicht immer intuitiv

```
if (flg & FLAG) ≠ if (flg & FLAG != 0)
⇨ if ((flg & FLAG) != 0)

r = hi << 4 + low;    /* hi << (4 + low) */
⇨ r = (hi << 4) + low;
```

- Tabelle der Operatorenprioritäten beachten!

Strichpunkt

```
if (x[i] > big)
  big = x[i];
big = x[i];
```

```
if (x[i] > big)
  big = x[i];
```

```
struct rec
{…
}
main() …
```

Definiert main als Funktion, die ein `struct rec` retourniert.

`;` fehlt

Dangling `else` Problem

```
if (x == 0)
   if (y == 0) err();
else
   f(x,y);
```

```
if (x == 0)
   if (y == 0) err();
   else
   {
       f(x,y);
   }
```

```
if (x == 0)
{
   if (y == 0) err();
}
else
{
   f(x,y);
}
```


Pointers und Arrays

- C kennt nur eindimensionale Arrays
- Arraygröße wird zur Compilezeit fixiert
- Zwei Arrayoperationen
  - Ermitteln der Größe des Arrays
  - Retournieren der Adresse von Element 0
- Alle anderen Operationen werden durch Pointeroperationen realisiert (z.B. Indizierung)
- (weitere Details: siehe Folien zum C-Block)

Arrays als Parameter

- Bei der Übergabe eines Arrays als Parameter wird der Parameter in einen Pointer auf das erste Element umgewandelt (keine Übergabe von Arrays!)
- C konvertiert Array-Parameterdeklarationen in entsprechende Pointerdeklarationen

```
main(int argc, char *argv[]) { … }
main(int argc, char **argv) { … }
```

=

Achtung: diese Konversion (Äquivalenz) gilt nur bei Parametern!

Auswertungsreihenfolge

- Auswertungsreihenfolge nur für vier Operatoren definiert: `&&`, `||`, `?:` und `,`
- Bei anderen Operatoren kann der Compiler die Auswertereihenfolge beliebig wählen

Bsp: `a < b` && `c < d`

1. 2., falls `a<b` true liefert

Auswertereihenfolge von `a` bzw. `b` ist compilerspezifisch

Bsp:
```
while (i < n)
    y[i] = x[i++];
```
falsch!

Portabilität

- Namen: In ANSI C sind die ersten sechs Zeichen für die Unterscheidung externer Namen signifikant (Keine Unterscheidung von Groß- und Kleinschreibung)
- Integer-Typen: short, plain, long
  - Größen: short ≤ plain ≤ long
    ANSI C: short, plain ≥ 16 Bit, long ≥ 32 Bit
  - Plain Integer ist groß genug für Array-Indizierung
  - Character: Größe entspricht "natürlicher", adressierbarer Einheit (8, 9, 16 Bit)

Portabilität (2)

- Characters: die meisten Compiler realisieren Characters als 8-bit Integers
- Konvertierung `char` → `int`: es ist undefiniert, ob ein Character als signed oder unsigned betrachtet wird.
- Shift-Operationen
  - Wie werden Bits beim Rechts-Shift aufgefüllt? → signed oder unsigned definieren.
  - Max. Shift-Count: `n-1` Bits bei Typ der Größe `n`

Portabilität (3)

Ganzzahlige Division

```
q = a / b;
r = a % b;
```

1. `q*b + r == a`
2. `a` negativ → `q` negativ, |q| unverändert
3. `b>0` → `r>=0` und `r<b`

Eigenschaften 1-3 nicht gleichzeitig erfüllbar

Bsp.: `(-3)/2`

- (2.): `q==-1` → (1.): `r==-1` → Widerspruch zu (3.)
- (3.): `r==1` → (1.): `q==-2` → Widerspruch zu (2.)

→ Die meisten Compiler geben Eig. 3 auf, einige Eigenschaft 2.

Programmierrichtlinien in der LU

- Checks beim Compilieren
- Fehlerabfragen im Programm
- Namenskonventionen
- Defensive Programmierung (keine komplizierten Ausdrücke, keine globalen Variablen, etc.)
- Ressourcensparend arbeiten
- Standardfunktionen verwenden

Literatur

- Andrew Koenig, *C Traps and Pitfalls*, Addison Wesley, 1988
- Les Hatton, *Safer C*, McGraw-Hill, 1994

Richtlinien für sicherheitskritische Anwendungen

- MISRA, *Guidelines for the Use of the C Language in Vehicle Based Software*, 1998
- Hecht, et al., *Review Guidelines for Software Languages for Use in Nuclear Power Plant Systems*, US Nucl. Reg. Comm., 1997 (WWW)

Was haben wir gelernt?

- C ist eine mächtige, weit verbreitete Programmiersprache
- Für das Schreiben korrekter und sicherer Programme ist eine gute Kenntnis der Sprache erforderlich
- Typische Probleme wurden vorgestellt
- Programming Guidelines