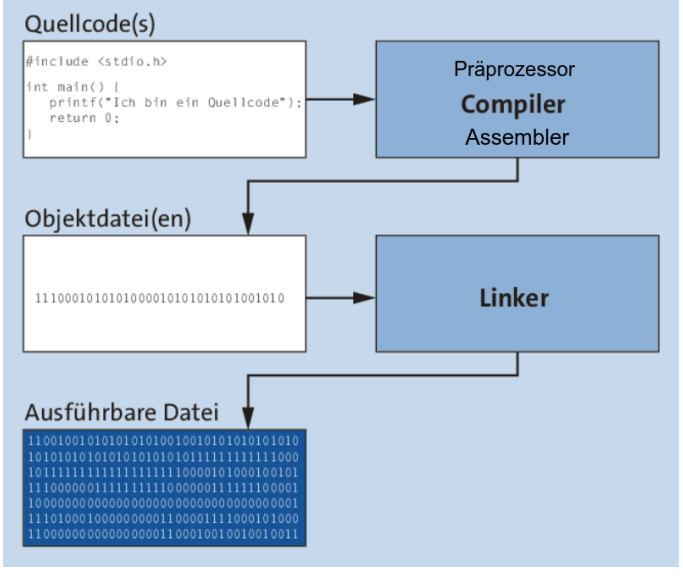
Grundlagen der Betriebssysteme

Praktische Einführung mit Virtualisierung

Stefan Bosse

Universität Koblenz - FB Informatik

C Programmierung



Die Grundlagen, um aus einem Quellcode mit einfachsten Mitteln eine ausführbare Datei zu generieren

Literatur

Jürgen Wolf, René Krooß

Grundkurs C

3., aktualisierte und überarbeitete Auflage
2020

C Compiler

Es gibt verschiedene Compiler. Folgende Auswahl sind bekannte und weniger bekannte C Compiler:

GCC aus GNU GCC

Der C-Compiler GCC aus der GNU Compiler Collection (kurz: GCC) wird auf vielen Plattformen verwendet und ist sehr beliebt, um neue Funktionen auszuprobieren. Unter Linux ist er für gewöhnlich die erste Wahl. Unter Windows steht die Portierung MinGW (Minimal GNU for Windows) zur Verfügung. Allerdings ist MinGW besonders unter Windows 10 nicht einfach zu installieren.

Clang der LLVM

Clang aus dem LLVM-Projekt ist der C-Standard-Compiler für die macOS-Entwicklung schlechthin, mit einer sehr guten Unterstützung für C11 und C18. Aber auch für Linux können Sie diesen Compiler kostenlos nachinstallieren.

C Compiler

Pelles C

Diese schlanke Entwicklungsumgebung für Microsoft Windows verwendet eine modifizierte und erweiterte Version des LCC-Compilers mit Unterstützung für C11 und C18 und dürfte somit die erste Wahl für die reine Programmierung in C unter Windows sein.

Microsoft Visual Studio

Diese Entwicklungsumgebung ist der Standard schlechthin für Windows-Anwendungen. Mittlerweile können Sie sogar Apps für macOS oder Smartphones erstellen. Visual Studio eignet sich nur bedingt für diesen Kurs (oder gar nicht), weil es für die einfachen Beispiele schlicht »überdimensioniert« ist und darüber hinaus vierstellige Beträge kostet. Eine Installation benötigt bis zu 10 GB Festplattenspeicher.

C Workflow und Toolchain

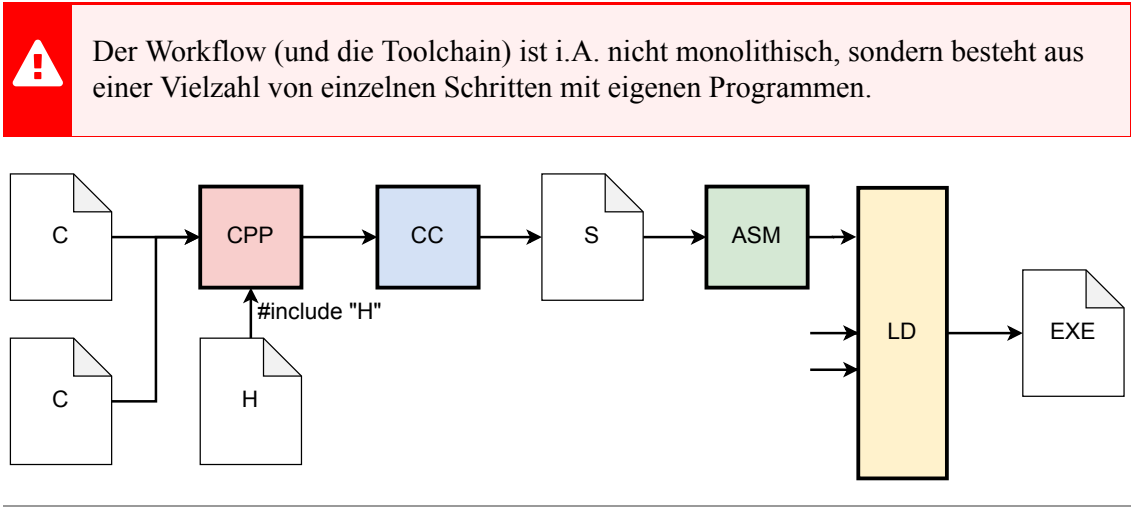


Abb. 1. Typischer C Workflow: Präprozessor CPP → C Compiler CC → Assembler ASM → Linker LD

C Workflow

Präprozessor

- Der Präprozessor ist eine Krücke um den C Programmcode anpassbar und konfigurierbar zu machen.
- Die Präprozessorsprache ist unabhängig von C und bietet Makrodefinition und Substitution sowie konditionale Codeblöcke

Makros

```
#define X 1
#define PRINT2(a,b) \
        printf("%d:%d",a,b)
#define LINUX 1
int x=X;
```

Konditionale 1

```
#ifdef LINUX
 XXX
#endif

#ifdef LINUX
 XXX
#else
 YYY
#endif
```

Konditionale 2

```
#if (X > 0)
  #define XSIGN 1
#else
  #define XSIGN 0
#endif

#if (X > 0)
  unsigned int x;
#else
  signed int x;
#endif
```

Präprozessor

Der C-JS Transpiler

In diesem Kurs wird C mittels eines C-JavaScript Transpilers geübt un erprobt.

Der C-JS Transpiler besteht aus folgenden Komponenten:

1. Ein gebündelter C Lexer und Parser (handwritten, Token und RegEx-basiert). Ausgabe ist ein Abstrakter Syntaxgraph.
2. Ein C-JS Generator mit Typprüfer. Ausgabe ist audführbarer JS Code.
3. Eine Ausführungsfunktion für den transpilierten Code.

Der generierte JS Code ist lesbar und kann eingesehen werden um ein tieferes Verständnis eines C Programms zu erhalten. Alles dreht sich um ein zentrales Speichermodell und Zugriff darauf.

Der C-JS Transpiler

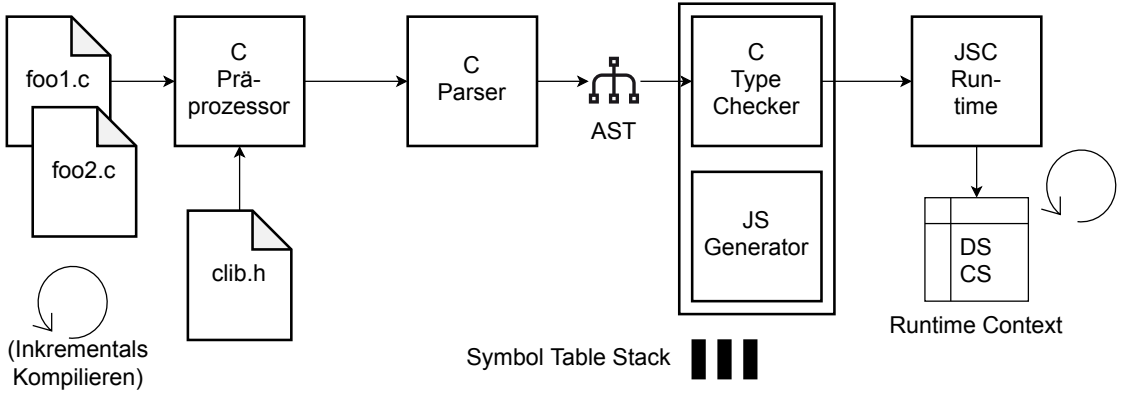
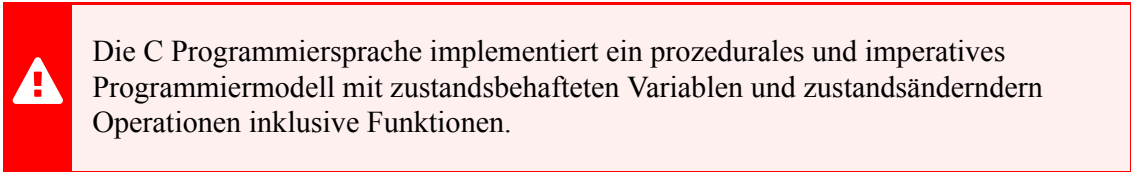


Abb. 2. Aufbau des C-JS Transpiler mit optionalen inkrementellen Kompilieren

Variablen und das Speichermodell



Die C Programmiersprache implementiert ein prozedurales und imperatives Programmiermodell mit zustandsbehafteten Variablen und zustandsänderndern Operationen inklusive Funktionen.

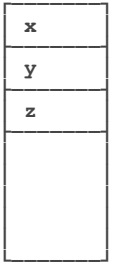
- Daten werden durch Variablen verwaltet. Eine Variable ist gekennzeichnet durch:
  - Wert ε
  - Name *name*
  - Speicheradresse (Beginn) σ
  - Datengröße (Länge) δ

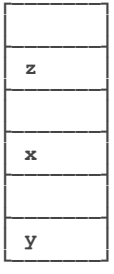
Eine Variable bezeichnet durch *name* befindet sich im Speicher Σ, formal ausgedrückt:

$$var(name) \Leftrightarrow \{\sigma, \delta, \epsilon\}$$
$$var(name) \Rightarrow Block(\sigma, \sigma + \delta)$$
$$Block(a, b) \Rightarrow [\epsilon]_a^b \in \Sigma$$

Variablen und das Speichermodell

- Eine Vielzahl von Variablen können hintereinander oder beliebig verteilt im Speicher Σ platziert werden.
  - Es gibt keine Regel wie Variablen im Speicher angeordnet werden
  - Die Platzierung kann zur Kompilierungszeit oder zur Laufzeit erfolgen
  - Statische Speicherbelegung versa dynamischer






```
int x,y,z;
```

Code 1. Zwei verschiedene Speicherallokationen für die Variablen x,y,z: Linear fortlaufend versa verstreut

Speichermodell C-JS Transpiler

DS

Das Datensegment welches den Heap (globale Variablen) und den Stack (lokale Variablen) beinhaltet.

CS

Anders als in C wo Funktionen Maschinenkode sind und im Speicher ebenfalls gespeichert werden, werden hier C Funktionen als normale JS Funktionen implementiert. Um Funktionszeiger (kommt noch) zu ermöglichen ist das Codesegment CS nur eine Indextabelle mit "virtuellen" Adressen von Funktionen, also eine Lookup Tabelle.

CSI

Reverse lookup Tabelle

Speichermodell C-JS Transpiler

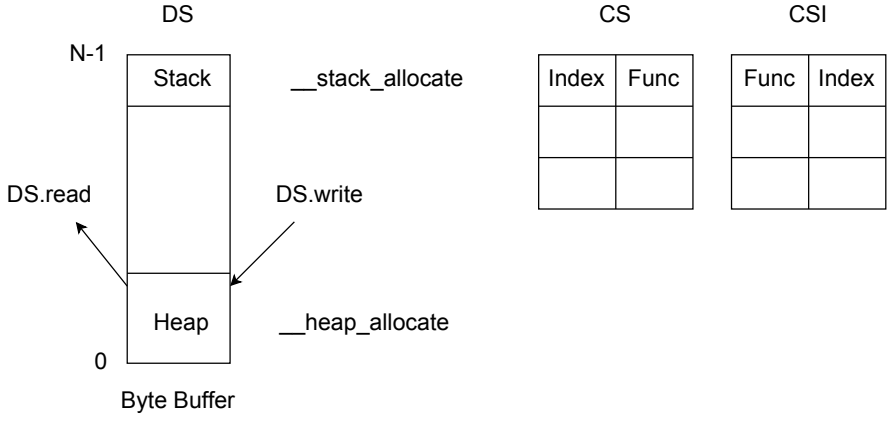


Abb. 3. Speichermodell des CJS Transpilers, DS: Datensegment, CS: Code Funktion Index Tabelle, CSI: Inverse CS Tabelle

Definition versa Deklaration

Definition

Einführung eines Bezeichners (Variable, Funktion) **mit Speicherallokation** und Festlegung des Datentyps oder der Typsignatur

Deklaration

Bekanntgabe eines Bezeichners (Variable, Funktion) **ohne Speicherallokation** und Festlegung des Datentyps oder der Typsignatur

Typdefinition

Einführung und Bekanntgabe eines benutzerdefinierten Datentyps (Structure, Enumeration)

```
int x;                          extern int x;
int foo(int x) { return x };    int foo(int x);
                                extern int foo(int x);
```

Code 2. (Links) Definition (Rechts) Deklaration

Symbole in C

Ein Symbol ist durch einen Namen (lexikalische Folge aus Textzeichen) und einer operationalen Semantik gekennzeichnet.

Es gibt folgende Symbolklassen:

1. Bezeichner (benutzerdefiniert)
2. Reservierte Schlüsselwörter
3. Literale
4. Begrenzer

Bezeichner in C

Bezeichner sind Namen für Objekte in einem Programm, die vom Programmierer festgelegt werden können, etwa Variablen, Funktionen usw. Für die Namen für gültige Bezeichner gelten folgende Regeln:

- Namen können aus Buchstaben aus dem Basis-Ausführungszeichensatz, Ziffern und Unterstrichen bestehen.
- Das erste Zeichen darf keine Zahl sein. Ein Bezeichner darf also nur mit einem Buchstaben oder einem Unterstrich beginnen. Zeichen wie @ oder der (Back-)Slash sind bei manchen Compilern nicht erlaubt.
- Es wird zwischen Groß- und Kleinbuchstaben unterschieden (engl.: case sensitive). Somit sind fun, Fun und FUN drei verschiedene Bezeichner.
- Als Bezeichner darf kein reserviertes Schlüsselwort von C verwendet werden.

```
int osMinorVersion2=1;
int main(int x) { .. };
struct node { .. };
```

Reservierte Bezeichner

Auf Bezeichner, die mit zwei sequenziellen Unterstrichen oder einem Unterstrich, gefolgt von einem Großbuchstaben beginnen, sollte verzichtet werden, weil sie für C-Implementierungen reserviert sind.

Bezeichner wie `__asdf` sind für gewöhnlich für Compiler-Zwecke, Bezeichner wie `_Asdf` für Betriebssystem- und Bibliotheken gedacht.

Bei gcc kann die Liste von vordefinierten Bezeichnern (mit Werten) mittels `echo | gcc -dM -E -` ausgegeben werden:

```
__UINT8_MAX__   0xff
__unix          1
__INT_WIDTH__   32
```

Reservierte Schlüsselwörter

Reservierte Schlüsselwörter dürfen nicht als Bezeichner verwendet werden (auch nicht in Feld- und Attributenamen),

```
auto     break    case    char    const  continue  default  do       double
else     enum     extern  float   for    goto      if       int      long
register return   short   signed  sizeof unsigned  void     volatile  while
```

Datentypen

Wir unterscheiden folgende Datentypen in C:

1. Nicht nutzerdefinierbare Kerndatentypen (zunächst Skalarwerte):
   - Ganzzahlige Werte (`long`,`int`,`short`)
   - Gleitkommawerte (`double`,`float`)
   - Bytes und Textzeichen (`char`)
   - Referenzen auf Variablen und Funktionen, Pointer (`*`)
2. Benutzerdefinierte Aggregate
   - Datenstrukturen/Records (`typedef`, `struct`)
   - Arrays von Datenstrukturen oder Kerndatentypen (`t[]`,`* t`)
3. Datentypmodifizierer
   - Vorzeichen (`signed`,`unsigned`)
   - Länge (`long`,`short`)

Boolescher Datentyp

In C gibt es keinen Booleschen Datentyp. Aber dieser kann einfach mit einer Ganzzahl implementiert werden. Dazu verwendet man eine Enumeration.

- Eine Enumeration definiert eine Wertemenge die i.A. einen Bezeichner mit einem Ganzzahlwert verknüpft.
- Enumerationen sind i.A. autonummeriert.

```
enum bool_vals {false,true};
typedef enum bool_vals bool;
bool b1=false;
```

- Hier ist `enum` die eigentliche Enumeration, mittels `typedef` wird aus der Wertemenge `bool` ein neuer Datentyp.

Enumerationen

- Eine Enumeration definiert eine Wertemenge die i.A. einen Bezeichner mit einem Ganzzahlwert verknüpft.
  - Der "echte" Datentyp einer Ennumeration ist i.A. Integer (int)

```
enum <ennumname> {
  C1,
  C2=<expressio>,
  ..
};
```

Def. 1. Eine Eumeration führt konstante benannte Werte ein

```
#define C1 0
#define C2 <expression>
...
```

Code 3. Alternative mit Präprozessor Makrodefinitionen

Variabledefinition

```
<datentypmod> <datentyp> <name>;
<datentypmod> <datentyp> <name> = <initval>;
```

Begrenzer

Um einzelne Symbole voneinander zu trennen, werden sogenannte Begrenzer benötigt.

Begrenzer	Bedeutung
Semikolon (;)	Dient als Abschluss einer Anweisung. Jeder Ausdruck, der mit einem Semikolon endet, wird als Anweisung behandelt. Der Compiler weiß dann, dass hier das Ende der Anweisung ist, und fährt nach der Abarbeitung dieser Anweisung mit der nächsten fort.
Komma (,)	Mit dem Komma trennen Sie für gewöhnlich gleichartige Elemente, z. B. können Sie mehrere Variablen für Ganzzahlen so definieren: `int minSp, maxSp, startSp;`
geschweifte Klammern ({})	Zwischen den geschweiften Klammern wird ein Anweisungsblock zusammengefasst. In diesem befinden sich alle Anweisungen (abgeschlossen mit einem Semikolon), die ausgeführt werden sollen.
Gleichheitszeichen (=)	Das Gleichheitszeichen = steht in C für eine Zuweisung, z. B. in `int maxSpieler = 500;`

Tab. 1. Einfache Begrenzer in C

Speicherbedarf

Eine Variable belegt immer einen Teil des Datenspeichers, auch bei Zeigervariablen (kommen noch).

- Speicherbedarf mit *sizeof* ermitteln
- Wenn die Größe eines Typs oder einer Variable benötigt wird, wird der sizeof-Operator verwendet.

Dieser gibt in der Regel die Größe des Operanden in Bytes zurück und wird beispielsweise bei der dynamischen Speicherreservierung verwendet, oder wenn Sie Programme schreiben, die auf andere Plattformen portierbar sind.

- Wie viel Speicherplatz ein Variablentyp letztlich benötigt, ist implementierungsabhängig.

```
int x;
int szx=sizeof(x);
int szint=sizeof(int);
```

Wertebereiche von Datentypen

Alle Datentypen können anders als in der Mathematik nur eine diskrete und endliche Menge von Werten darstellen (kodieren).

[num2017]

m	Speicher	Zahlbereich	42 (Big Endian)	42 (Little Endian)
8	1 Byte	$0,\ldots,2^8-1(=255)$	0010 1010	0101 0100
16	2 Bytes	$0,\ldots,2^{16}-1(=65535)$	0000 0000 0010 1010	0101 0100 0000 0000
32	4 Bytes	$0,\ldots,2^{32}-1(=4294967295)$	...	
64	8 Bytes	$0,\ldots,2^{64}-1(\approx 1.84\cdot 10^{19})$	...	

Eigenschaften	single precision/float	double precision/double
Speichergröße	32 bit = 4 byte	64 bit = 8 byte
Bits der Mantisse	23	52
Stellen der Mantisse	24	53
Bits des Exponenten	8	11
Zahlbereich des Exponenten	$-126,\ldots,127$	$-1022,\ldots,1023$
Zahlbereich normalisiert	$1.2\cdot 10^{-38},\ldots,3.4\cdot 10^{38}$	$2.2\cdot 10^{-308},\ldots,1.8\cdot 10^{308}$
kleinste positive denorm. Zahl	$1.4\cdot 10^{-45}$	$4.9\cdot 10^{-324}$
Genauigkeit für Dezimalzahlen	7 – 8 Stellen	15 – 16 Stellen

Abb. 4. Wertebereiche von Ganzzahl- und Gleitkommazahlen in C. Die Bytereihenfolge ist prozessor- und betriebssystemabhängig!

Konstanten

Benötigen Sie einen unveränderbaren Wert, können Sie eine Konstante verwenden.

- Der Sinn und Zweck einer solchen Konstanten ist es, dass der Wert zur Laufzeit des Programms nicht mehr verändert werden kann.
- Das Gegenstück zu einer Konstanten ist eine Variable.
- Eine Konstante definiert man, indem vor den eigentlichen Datentyp das Schlüsselwort const gesetzt wird (**Typqualifikator** als Modifizierer, als Alternative zur Makrowertdefinition für den Präprozessor)

```
#define N2 100
const int N1 = 100;
int x1 = N1;
int x2 = N2;
```

Vorteil: "read only" Semantik, der Wert von konstanten "Variablen" kann zur Kompilierungszeit substituiert werden und benötigt keinen (Heap/Stack) Speicher.

Lebensdauer und Sichtbarkeit von Variablen

Die Lebensdauer einer Variablen gilt immer bis zum Ende des Anweisungsblocks (Bezeichnerkontext, Scope), in dem sie definiert wurde. Dazu ein einfaches Beispiel:

```
int x=1,y=2;
if (x>0) {
  int x=2;
  printf("x=%d y=%d\n",x,y);
}
printf("x=%d y=%d\n",x,y);
```

- Die Variable *x* gibt es tatsächlich zweimal.
- Am Ende vom Blockkontext ist nur der äußere Kontext sichtbar, d.h. hier die ersten *x* und *y* Variablen, im tieferen Kontext wird *x* überlagert (überschattet), aber *y* kann aus dem äußeren Kontext verwendet werden.

Lebensdauer und Sichtbarkeit von Variablen

- Lokaler Blockscope

Referenz- versa Wertsemantik

Wir unterscheiden:

1. **Wertsemantik** ⇒ Eine Variable liefert immer ihren Wert, aber nicht ihre Kontainereferenz

```
x=100
function foo(u) {
  u=200
}
foo(x)
// x==100!
```

2. **Referenzsemantik** ⇒ Eine Variable wird über ihre Referenz ihres Kontainers (oder Wertes), also letztlich einer Speicheradresse benutzt.

```
x=100
function foo(reference u) {
  u=200
}
```

Die Unterscheidung beider Semantiken kann verwirrend sein. Insbesondere in C. Hier unterscheiden wir Wert- und Zeigervariablen.

- Das pardoxe in C ist aber: Auch Zeigervariablen (werden noch eingeführt) unterliegen der Wertsemantik. **In C gibt es nur Wertsemantik!**. Das ist i.A. typisch für statisch typisierte Programmiersprachen.
- Dynamisch typisierte Programmiersprachen können teils wert- teil referenzbasiert sein.

Welche Programmiersprachen unterstützen echte Referenzsemantik bei Funktionsparametern?

Kontrollanweisungen

Bedingte Verzweigung

```
if (ε) {
  ..
}
if (ε) {
 ..
} else {
 ..
}
```

Mehrfachauswahl

```
switch (ε) {
  case c1:
    ..
    break;
  case c2:
    ..
    break;
  case c3:
    ..
    break;
  default :
    ..
}
```

Schleifen

```
for(i=ε;i<ε;i=ε(i)) {
  ..
}
while (ε) {
  ..
}
do {
  ..
} while (ε)
```

Schleifen

Man unterscheidet:

1. Zählschleife `for(start;condition;update)` mit drei integrierten Anweisungen: Einer Initialisierung (einer oder mehrere Zählvariablen), einer Schleifenbedingung die vor jedem Durchlauf getestet wird (i.A. der Zählvariable), und einer Update Anweisung die nach jedem Schleifendurchlauf ausgeführt wird (i.A. Veränderung der Zählvariable).

2. Abweisende Schleife `while(condition)` die abhängig von der Schleifenbedingung niemals, einmal oder mehrmals den Schleifenkörper ausführt (Test vor Schleifendurchlauf).

3. Eine nicht abweisende Schleife `do {} while(condfition)` die mindestens einmal den Schleifenkörper ausführt (Test nach jedem Schleifendurchlauf).

Beispiele von Kontrollanweisungen

Goto Hell

- Goto ist eine reine Sprunganweisung zu einem symbolischen Label an anderer Stelle im Programm (unbedingter Sprung).
- Goto ist ein Relikt aus alten (Assembler) Zeiten und reduziert die Lesbarkeit und erhöht die Fehlerwahrscheinlichekit von Programmen.
- Goto kann vorwärts oder rückwärts im Programmfluß verzweigen.

Goto sollte nur sparsam (wenn überhaupt) verwendet werden und **nut vorwärts verzweigende**.

- Modernes (vorwärts) Goto: Exceptions!
- Aber: Goto kann die Lesbarkeit und Nachvollziehbarkeit von Programmen auch erhöhen **wenn es keine Exceptions gibt**, wie das in C der Fall ist.

Goto Heaven

- Die Bearbeitung von Ein-Ausgabe Aufgaben, d.h. die Kommunikation mit Peripheriegeräten, ist i.A. mehrschrittig.
  - Bei den einzelnen Schritten kann es zu Fehlern kommen (z.B. Timeout).
  - Nach jeder Teiloperation wird der Status abgefragt, und wenn ein Fehler aufgetreten ist dann wird zum Ausgang der Hauptoperation und einer Fehlerbehandlung verzweigt.

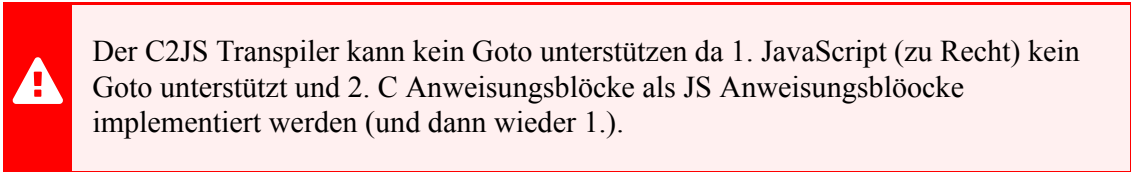
Goto nur als Ersatz für Exceptions benutzen.

- Eine Ausnahme: Virtuelle Machinen und deren Instruktionsinterpreter. Hier kann die Verwendung von Goto vorwärts und rückwärts sinnvoll sein.

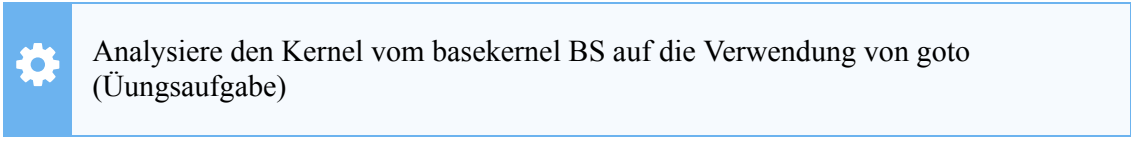
```
#define EOK 0
#define EIO -1
int IOOP(...) {
  int status;
  status=dev_open(...);
  if (status!=0) goto error;  // raise Error
  status=dev_read(...);
  if (status!=0) goto error;  // raise Error
  status=dev_write(...);
  if (status!=0) goto error;  // raise Error
  status=dev_close(...);
  if (status!=0) goto error;  // raise Error
  return EOK;
error:
  dev_close(...);  // handle Error locally
  return EIO;  // raise EIO
}
```

Alg. 1. Teilschrittige Ausführung einer EA Operation mit Fehlerbehandlung

Goto Earth



- Um C Goto zu unterstützen bräuchte man eine VM Loop die adressierte (Bytecode) Instruktionen verarbeitet - haben wir hier aber nicht.
- Eine Krücke wäre nur die o.g. Struktur in eine try-catch-raise Struktur umzusetzen.



Funktionen

> Eine Funktion besteht aus einem Namen, einer (optionalen) Liste von Funktionsparametern, dem Rückgabetyp, und dem Funktionsrumpf aus Anweisungen.

- Der Funktionsrumpf ist ein Blocklevel Scope und kann neue lokale Variablen einführen, aber keine weitere Funktionen oder Typdefinitionen.

```
==== Definition ====
<rettype> <name>(<partype> <parname>,...) { ... };
==== (Vorwärts)Deklaration ====
extern <rettype> <name>(<partype> <parname>,...);
==== Aufruf ====
<name>(<argexpr>,<argexpr>,..)
```

Def. 2. Funktionsdefinition in C

Beispiele von Funktionen