

Informatik 2

Assignment Project Exam Help

<https://tutorcs.com>
01 – Bausteine von CO

WeChat: cstutorcs

Sommer 2021

Torsten Grust
Universität Tübingen, Germany

1 | Übersetzung (Compilation) von C0-Programmen

C0-Programmtext wird durch einen **Übersetzer** (auch: *Compiler*) in ein ausführbares Programm überführt:

1. Verfasse Programmtext (auch: Quelltext, *Source*) mittels regulärem **Texteditor**. Erzeuge Quelltext `program.c0`.
Assignment Project Exam Help
2. Übersetze Quelltext `program.c0` mittels **C0-Compiler**¹ `cc0` in ausführbaren Maschinen-Code. Erzeugt Programm `program`.
<https://tutorcs.com>
3. Führe `program` in der **Shell** des Systems aus. Die Ein-/Ausgabe erfolgt über das **Terminal** der Shell.
WeChat: cstutorcs

¹ Hinter den Kulissen wird `program.c0` zunächst in ein C-Programm `program.c0.c` übersetzt, das dann direkt mit dem C-Compiler `cc` (oft: `gcc` oder `clang`) des Systems in ausführbaren Code compiliert wird.

C0-Quelltext

Beispiel:

1. C0-Quelltext `bazinga.c0`:

```
#use <conio>
int main() {
    println("Bazinga!");
    return 0;
}
```

Assignment Project Exam Help
<https://tutorcs.com>
WeChat: cstutorcs

- **Hinweis:** Wir setzen in der *Informatik 2* keine IDEs ein.
 - Jeder Plain-Texteditor ist OK.
 - C0 ist ein Dialekt der Programmiersprache C. Support für C in Editoren (Syntax-Highlighting) hilft auch C0.

Compilation des C0-Quelltextes, Ausführung des Programms

2. Übersetzung mit C0-Compiler `cc0` (Shell-Prompt ist `$`):

```
$ cc0 -d -o bazinga bazinga.c0
$ ls -l
-rwxr-xr-x 1 grust staff 52968 Mar 23 14:26 bazinga
```

3. Ausführung des Programms `bazinga` in der Shell:²

<https://tutorcs.com>
WeChat: cstutorcs

```
$ ./bazinga
Bazinga!
0
$
```

← Ausgabe des Programms
← *return code* (0 ≡ OK)

² `./program` führt Programm `program` im aktuellen *working directory* der Shell (Notation: `./`) aus.

2 | Der C0-Compiler cc0

cc0 übersetzt C0-Quelltexte in ausführbaren Maschinen-Code.

Optionen (*-opt* bzw. *--opt*) beeinflussen die Compilation:

```
konkateniert zu einem Quelltext
$ cc0 -d -o program source1.c0 source2.c0 ... source_n.c0
      |  |
      |  |----- Name des ausführbaren Programms (Default: a.out)
      |  |----- während Ausführung von program (Vertrags)-
      |  |----- Überprüfungen durchführen (dynamic checks)
```

- cc0 meldet Art und Ort syntaktischer Fehler im Quelltext:


```
bazinga.c0: 4.3-4.22: error: undeclared function ...
           |
           |----- Quelltextregion: 「 ... 」 (Zeile 4, Spalten 3-22)
```

3 | C0: Anweisungen vs. Ausdrücke

Zur Laufzeit eines C0-Programms kommt es

- zur **Ausführung von Anweisungen** (*statements* \textcircled{S}), die einen *Effekt auf den Zustand des Programms/Systems* haben und
- zur **Auswertung von Ausdrücken** (*expressions* \textcircled{E}), die einen *Wert berechnen*.

<https://tutorcs.com>

 Algorithmen durch Effekte/Zustandsänderungen zu beschreiben, ist charakteristisch für das **imperative Programmierparadigma**. C0 ist eine **imperative Programmiersprache**.

- Funktionale Programmiersprachen—wie Racket—legen den Fokus hingegen allein auf die Ausdrucksauswertung (\rightsquigarrow).

4 | C0: Funktionen gruppieren Anweisungen

Alle nicht-trivialen Probleme zerfallen in Teilaufgaben. Anweisungen, die gemeinsam ein solches Teilproblem lösen, werden in C0 in **Funktionen** gruppiert.

- **Definition einer Funktion**

`void f() {`
 `statement1`
 `statement2`
 `⋮`
 `statementn`
`}`

<https://tutorcs.com>
WeChat: cstutorcs

Body der Funktion *f*
(auch: Block)

Ausführungsreihenfolge

- **Typ** `void` von *f* zeigt an, dass uns nur die Effekte der `statementi` in *f* interessieren. *f* berechnet keinen Wert.

C0: Funktionsaufruf ⓘ

- Die Statements im Body von Funktion f kommen durch **Funktionsaufruf** zur Ausführung:

$f()$;

[Statements enden mit ‘;’]

- Assignment Project Exam Help
<https://tutorcs.com>
WeChat: cstutorcs
- Effekt von $f()$: Zustandsänderung nach Ausführung von $statement_1, statement_2, \dots, statement_n$. $statement_{i+1}$ sieht das System im Zustand nach $statement_i$.
- ⚠ Reihenfolge der **Sequenz von Anweisungen** relevant.
- Die Funktion $main$ existiert in jedem C0-Programm. Bei Programmstart führt C0 den Funktionsaufruf $main()$ durch.³

³ Funktion $main$ hat einen Effekt und berechnet einen Wert des Typs int , der an das Betriebssystem zurückgemeldet wird (*return code*). Dazu später mehr.

C0: Funktionsaufrufe ⑤ (Beispiel)

C0-Programm `f-and-g.c0`:

```
#use <conio>

void f() {
    println("Hi, this is f!"); /* Effekt: Terminal-Output */
}

void g() {
    println("Hi, this is g!"); /* Effekt: Terminal-Output */
}

int main() {
    f(); /* Effekt: die Effekte von f() */
    g(); /* und g() */

    return 0; /* return code */
}
```

C0: Funktionsdeklarationen (*Forward Declarations*)

Funktion g ist für Funktion f bekannt und aufrufbar, wenn

1. g im Programmtext vor f **definiert** oder
2. g im Programmtext vor f **deklariert** wurde.

Assignment Project Exam Help

Funktionsdeklaration für Funktion g :

<https://tutorcs.com>

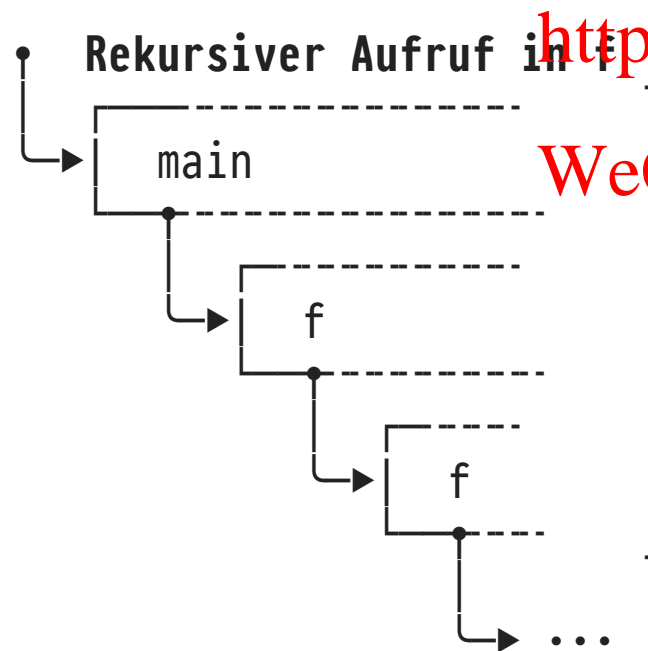
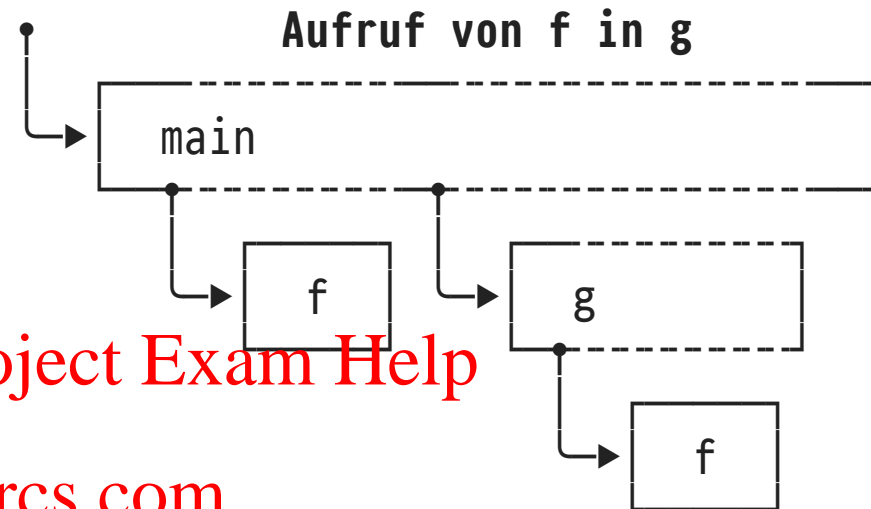
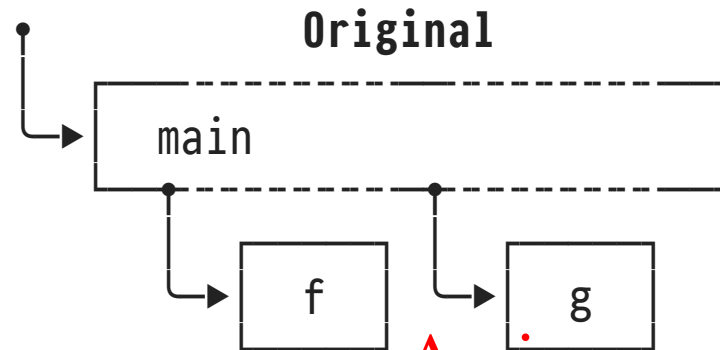
```
void g();
```

[(noch) kein Body für g]

WeChat: cstutorcs

- Die **Deklaration** führt den Namen der Funktion (auch: Typ, Variable) ein, *ohne* bereits eine Definition vorzunehmen.
- Solche *forward declarations* erlauben dem Compiler, den Quelltext strikt vorwärts ↓ und nur einmal zu lesen.

Call Chains (Funktionsaufrufe in `f-and-g.c0`)



Assignment Project Exam Help

<https://tutorcs.com>

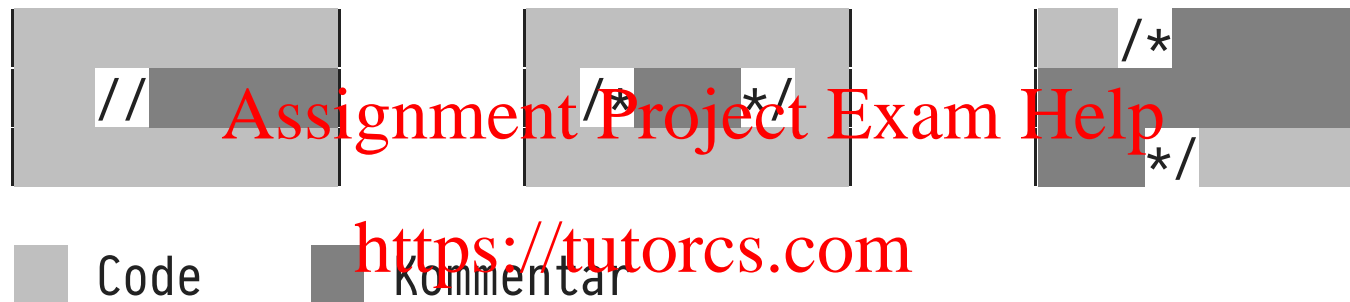
WeChat: cstutorcs

⌚ Laufzeit →

nicht beendete Aufrufe
(*stack overflow* ↗)

5 | C0: Kommentare

Kommentare `//...CR` und `/*...*/` annotieren Code und markieren Quelltextregionen, die bei der Compilation ignoriert werden:



Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

- Kommentare `//...CR` reichen bis zum Zeilenende ^{C_R}.
- Kommentare `/*...*/` sind schachtelbar und eignen sich zum **Auskommentieren** ganzer (evtl. kommentierter) Regionen von Quelltext.

Ausdrücke (*expressions* \mathbb{E}) berechnen **Werte**.⁴

- **Atomare Ausdrücke** (auch: Konstante, Literale) sind Werte, die ohne weitere Berechnung für sich selbst stehen:

Assignment Project Exam Help
<https://tutorcs.com>
WeChat: cstutorcs

Literale	Typ	
42, 0, -1138	int	ganze Zahlen
'x', '\n'	char	Zeichen
"Bazinga!", ""	string	Zeichenketten
true, false	bool	Wahrheitswerte

Atomare Ausdrücke (Konstante, Literale)

- C0 kennt *keine* Fließkommazahlen (C: ~~Typ~~ ~~float~~). Damit ist jegliche Arithmetik in C0 exakt.

⁴ Sogenannte **Seiteneffekte** dürfen bei der Ausdruckswertung auftreten (\rightarrow später).

C0: Zusammengesetzte Ausdrücke

Zusammengesetzte Ausdrücke werden mittels Operatoren gebildet:

Ausdrücke	Wert	Typ	
<code>42 + 0, 43 - 1</code>	<code>42</code>	<code>int</code>	Addition, Subtraktion (binär)
<code>-1138</code>	<code>-1138</code>	<code>int</code>	Negation (unär)
<code>28 * 68</code>	<code>1904</code>	<code>int</code>	Multiplikation
<code>20 / 3</code>	<code>6</code>	<code>int</code>	ganzzahlige Division
<code>1 / 2, (-1) / 2</code>	<code>0</code>	<code>int</code>	• <code>/</code> rundet Richtung 0
<code>20 % 3</code>	<code>2</code>	<code>int</code>	Modulo
<code>1 % 2</code>	<code>1</code>	<code>int</code>	• generell: $a == (a/b)*b + a\%b$
<code>(-1) % 2</code>	<code>-1</code>	<code>int</code>	
<code>0 == 0, false != true</code>	<code>true</code>	<code>bool</code>	Vergleiche ⁵ (Relationen)
<code>1 <= -1, 'X' > 'x'</code>	<code>false</code>	<code>bool</code>	• nur auf <code>int</code> , <code>char</code> : <code><</code> , <code><=</code> , <code>>=</code> , <code>></code>

Zusammengesetzte Ausdrücke (Ausschnitt)

⁵ C0 vergleicht Werte des Typs `char` auf der Basis des ASCII-Codes der Zeichen (https://de.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange).

CO-Operatoren: Priorität, Assoziativität und Klammerung

- Die **Priorität** und **Assoziativität** der Operatoren bestimmt die Interpretation zusammengesetzter Ausdrücke:

$2 * 3 + 4$	$== 10$	[Priorität von $*$ höher als $+$]
$42 < 40 + 1$	$== \text{false}$	[Priorität von $+$ höher als $<$]
$2 - 3 - 4$	$== -5$	$[-$ assoziiert nach links]
$\vdots \dots = \dots \vdots$		

Assignment Project Exam Help

<https://tutorcs.com>

WeChat: cstutorcs

- Klammerung** (\dots) macht die Interpretation explizit:

$(2 * 3) + 4$	$== 10$
$2 * (3 + 4)$	$== 14$
$(2 - 3) - 4$	$== -5$
$2 - (3 - 4)$	$== 3$

C0-Operatoren: Priorität und Assoziativität

Die Familie der C0-Operatoren ist groß. Im Laufe des Semesters vervollständigen wir die **Operator-Tabelle**:

Priorität	Operatoren	Assoziativität	
13	-	rechts	Negation (unär)
12	*	links	
11	+, -	links	
9	<, <=, >=, >	links	
8	==, !=	links	

C0-Operator-Tabelle (Ausschnitt)

❓ <, <=, >=, > sind links-assoziativ, aber Ausdrücke der Form $x < y < z$ sind illegal in C0 (aber legal in C).
Q: Wieso illegal?

C0: Seiteneffekte



Seiteneffekte (*side effects*) dürfen bei der Auswertung von C0-Ausdrücken auftreten.

- Seiteneffekte können in unerwarteter Häufigkeit und Reihenfolge ausgelöst werden.
- Seiteneffekte sind oft Zeichen fragwürdigen Stils. Separiere effektbehaftete Statements und pure Ausdrücke.
- Kontrollierte Seiteneffekte können u.a. bei der Fehlersuche nützlich sein (“*printf debugging*”).

Ausdrucksauswertung © (nur Seiteneffekte, ignoriere Wert):

`e;`

[Wert von `e` berechnet, aber verworfen]

7 | Der C0-Interpreter **coin**

Der **C0-Interpreter** **coin** (auch: REPL, *read-execute/eval-print loop*) wertet Anweisungen \textcircled{S} und Ausdrücke \textcircled{E} interaktiv aus:

read \rightarrow execute/eval \rightarrow print
Assignment Project Exam Help

<https://tutorcs.com>

- **coin** platziert den **WebChat** **tutorcs** innerhalb des Bodys einer (unbenannten) Funktion, die nicht verlassen werden kann.
- Effekte werden sofort ausgeführt, Ausdrücke sofort ausgewertet und der resultierende Wert (mit Typ) in der REPL ausgegeben (intern: C0 VM, keine Compilation).

Der C0-Interpreter `coin`

- Aufruf von `coin` in der Shell (`coin`-Prompt ist `-->`):

```
$ coin -d -l lib [ source1.c0 ... sourcen.c0 ]
      |
      |  L- Funktionen der Bibliothek lib laden
      |  L- dynamic checks durchführen
      |
      | https://tutorcs.com
      | WeChat: @tutorcs
C0 interpreter (coin) 0.3.3 'Nickel' (r793, ...)
Type '#help' for help or '#quit' to exit.
--> 6 * (3 + 4);
42 (int)                                     ← Antwort der REPL
--> main();
```

- Option `-l lib` macht die Funktionen der C0-Bibliothek `lib` in `coin` verfügbar (vgl. `#use <lib>`, → später).

8 | C0: Typen

Typen definieren Mengen von Werten. In C0 besitzt *jeder* Wert *genau einen* Typ.⁶

Typ	Wertemenge
int	ganze Zahlen ($\subseteq \mathbb{Z}$)
bool	{true, false}
char	ASCII-Zeichen (Codes 0–127)
string	Zeichenketten (Sequenzen von char)
void	\emptyset

WeChat: cstutorcs
C0-Typen (Ausschnitt)



C0 ist **streng** und **statisch typisiert**: Ausdrücke können nur mit Werten korrekten Typs formuliert werden. Typfehler werden bereits bei Übersetzung erkannt.

⁶ Werte besitzen *einen* Typ, es gibt *keine* Typvariablen: C0 ist **monomorph** (vs. Polymorphie in Racket).

C0: Typen von Operatoren und Funktionen (Signaturen)

Wenn Operator \oplus zusammen mit den Typen seiner Argumente/seines Resultats (auch: **Funktionstyp**, **Signatur**) kommuniziert wird, ist bereits eine Menge über \oplus bekannt:

Resultatstyp
↓
`int + (int, int)`
↑ ↑ ↑
Operator Argumenttypen

```
bool == (int, int)
bool == (bool, bool)
```

```
int  / (int, int)
int  - (int)
int  - (int, int)
bool < (int, int)
```

```
void println(string)
void printint(int)
void printbool(bool)
```

↑
kein Resultatwert,
interessant sind
nur die Effekte

9 | C0: Variablen

Variablen repräsentieren den *änderbaren* Speicher des Rechners: der Wert einer Variablen kann über die Zeit geändert werden.

- **Variablendeklaration** für Variable v des Typs τ :

Assignment Project Exam Help

```
 $\tau$   $v$ ;
```

<https://tutorcs.com>

Keine Nutzung von Variablen ohne vorherige Deklaration.

WeChat: cstutorcs

- Variablen sind **typisiert**: v kann ausschliesslich Werte des Typs τ annehmen.
- Gültige Variablennamen v (auch: *Identifizier*):
 - im Block eindeutig, Groß-/Kleinschreibung relevant,
 - Zeichen $\in \{A-Z, a-z, _, 0-9\}$, kein $0-9$ am Anfang.

C0: (Block-)Lokale Variablen



Alle C0-Variablen sind **lokal**: ab Ort der Deklaration in einem Block `{...}` lediglich bis zum Blockende sichtbar.

```
int x;
```

```
void f() {  
    int x;  
    :  
}
```

```
void g() {  
    int y;  
    :  
    { int z;  
      :  
    }  
    :  
}
```

! keine Variablen ausserhalb eines Blocks

Assignment Project Exam Help

] Variable x sichtbar
(auch: Scope von x)

WeChat: cstutorcs

] nur y sichtbar
(äußerer Scope)

] y, z sichtbar
(innerer Scope)

C0: Uninitialisierte Variablen und Zuweisung

Nach Deklaration $\tau \ v$; ist Variable v **uninitialisiert**. Auswertung von uninitialisiertem v führt zu Übersetzungsfehler:

```
{ int x;  
  x;  
}
```



uninitialized variable 'x'
Assignment Project Exam Help

- **Zuweisung** \textcircled{S} (auch: *Assignment*) von Ausdruck e an v :
<https://tutorcs.com>
WeChat: cstutorcs

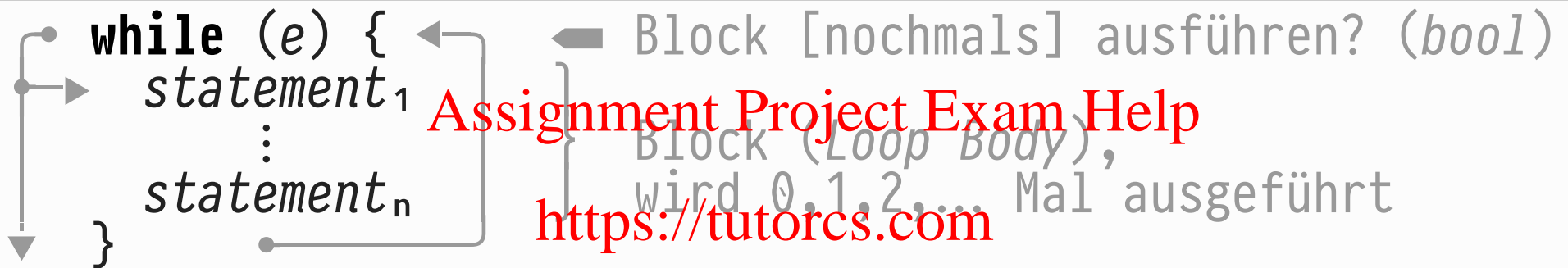
```
 $v = e$ ;
```

Zuweisung ist ein Statement mit Effekt auf den Speicher:

1. Werte e zu Wert z des Typs τ aus (Seiteneffekte),
2. **überschreibe** bisherigen Wert von v mit z (Effekt).

10 | C0: Iteration (**while**-Loop)

Iteration mittels **while** ⑤ wiederholt die Ausführung eines Blocks von Statements, solange Prädikat **e** erfüllt ist:



- Effekt des **while**-Statements: **WeChat: cstutorcs**

1 Werte Prädikat **e** aus (Seiteneffekte), ergibt **z** (*bool*).

2 $\left\{ \begin{array}{ll} \text{Führe } statement_1 \dots_n \text{ aus, zu } \mathbf{1} & \text{falls } z == \text{true} \\ - & \text{sonst} \end{array} \right.$