C - Grundlagen und Konzepte

C-Präprozessor & Compiler

Jan Niklas Hennings

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

24. April 2014

Vom Quellcode zum Executable

Motivation

- Präprozessor und Compiler?
  - Große Unterschiede zu Java!
  - Vorverarbeitung vor dem Kompilieren (Präprozessor)
  - Keine VM, sondern erzeugter Maschinencode
  - Viele unterschiedliche Compiler und Möglichkeiten
    $\rightarrow$ direkter Einfluss auf das endgültige Programm

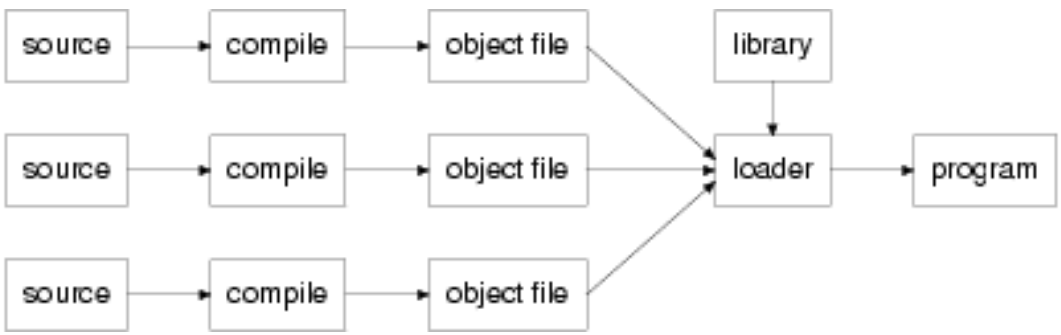

Abbildung: "Separate compilation" [2]

Der Präprozessor

- Erste Instanz vor dem Kompilieren
  - Früher: Eigenständiges Programm
  - Heute: Bestandteil jedes modernen Compilers
- Grob: Code-Ersetzung
- Mächtiges Werkzeug
  - Sehr fehleranfällig für unerfahrene User
  - Verarbeiteter Code muss keineswegs sinnvoll oder richtig sein

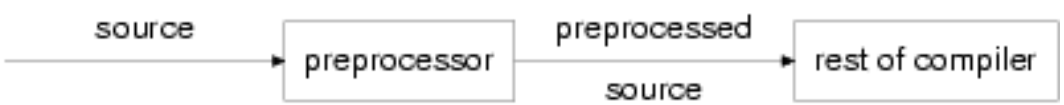

Abbildung: "The preprocessor" [4]

Elementare Aufgaben

- Entfernen von Kommentaren
  - nicht relevant für den Compiler
  - Verkleinerung des Sourcecodes
- **#include**
  - *Hello World*:

```
#include<stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

#include

- Ersetzen durch Inhalt der *Header Files* (.h)
  - Viele solcher *Header Files* in jeder C-Compiler Standardbibliothek
  - Im *Hello World*-Beispiel: Befehle für die Stadard-Ein- und Ausgabe (printf())
- Etwa vergleichbar mit (*import **) -Anweisung in Java

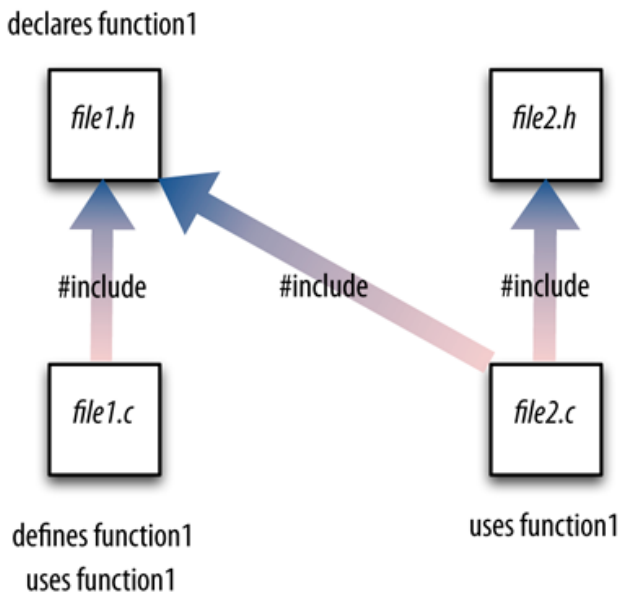

Abbildung: "How a large C program is divided into files" [13]

#include (cont.)

- Einbinden eigener Dateien möglich
  - *funktionen.h*:

```
int add(int a, int b) {
    return a + b;
}
```

  - *funktionen.h*:

```
#include<stdio.h>
#include "funktionen.h"

int main() {
    printf("Summe: %d\n", add(4, 5));
    return 0;
}
```

  - Ausgabe: **9** (Quelle: [6])
- häufig gebrauchte Funktionen in eigene *Header Files* auslagern

Makros

- Definiert durch **#define BEZEICHNER wert**
- Entfernen durch **#undef BEZEICHNER**
  - Im Quelltext wird **BEZEICHNER** nun durch **wert** ersetzt.
  - Quelltext:

```
#define PI 3.1415926f

float umfang(float d) {
return PI * d;
}
```

  - nach Präprozessor:

```
float umfang(float d) {
    return 3.1415926f * d;
}
```

  - (teilweise nach: [17])

parameterbehaftete Makros

- Definiert durch **#define BEZEICHNER(Args) ausdruck**
  - z.B. Bool'scher Ausdruck:

```
#define KLEINER_100(x) ((x) < 100)

void klHundert(int zahl) {
    if(KLEINER_100(zahl))
    printf("Ja! Die Zahl ist kleiner als 100!\n");
    else
    printf("Die Zahl ist größer als 100!\n");
}
```

  - (Quelle: [17])

bedingte Kompilierung

- **#if** und **#endif** schließen Quelltextblöcke ein, die je nach erfüllter Bedingung dem Compiler übergeben werden.

```
#if SIZE < 128
    #undef SIZE
    #define SIZE 256
#endif
```

- (Quelle: [16])

bedingte Kompilierung (cont.)

- Ideal für Debugging
  - **#ifdef**:

```
#define DEBUG

#ifdef DEBUG
    printf("aktueller Wert von SIZE: %d", SIZE);
#endif
```

- Portierbarkeit zwischen unterschiedlichen Systemen
  -

```
#ifdef WINDOWS
    printf("Programm läuft unter Windows\n");
#elif UNIX || LINUX
    printf("Programm läuft unter UNIX/LINUX\n");
#else
    printf("Unbekanntes Betriebssystem!!\n");
#endif
```

- (Quelle: [16])

Zusammenfassung

- Präprozessor ist ein mächtiges Werkzeug
  - Header Files einfügen
  - Konstanten definieren
  - funktionsähnliche Makros definieren
  - Portierbarkeit und Debugging unterstützen
- Vorsicht: Code wird immer nur ersetzt!
  Verlust der Übersichtlichkeit
- "We would suggest that at least six months experience is the minimum prerequisite for a full attack." [1]

Compiler

- Erhält vorbereiteten Code vom Präprozessor
- Übersetzt von Hochsprache in semantisch äquivalenten Maschinencode
- Arbeitet in mehreren unterteilten Phasen
  - Ergebnis in C: *Object-Files* (.o)
  - *Object-Files* werden vom *Linker* endgültig zusammengeführt

Phasen-Modell

- Analyse-Phase (Front-end)
  - Lexikalische Analyse
  - Syntaktische Analyse
  - Semantische Analyse
- Synthese-Phase (Back-end)
  - Zwischencode-Erzeugung
  - Programmoptimierung
  - Maschinencode-Erzeugung

I. Lexikalische Analyse

- Zerlegung des Quellcodes in *Tokens*
  - *Token*: Bezeichner, Zahlen, Schlüsselworte (*if, while...*) etc.
- umfang = radius^2 * 3,14;

```
 umfang   =    radius   ^   2   *   3,14   ;
<umfang> <=>  <radius> <^> <2> <*> <3,14> <;>
```

  - Fehler: ungültiges Zeichen "^", und die Real-Zahl "3,14" enthält ein Komma statt einem Punkt
  - Lexikalische Analyse in etwa eine **Rechtschreibkorrektur**
  - (Quelle: [8])

I. Syntaktische Analyse

- Prüfung des lexikalisch korrekten Codes auf syntaktische Korrektheit
- *Parsing* der zerlegten linearen Tokens in einen *Syntax-Baum*
- `flaeche = pi * r * ;`
  - lexikalisch korrekt
  - Aber Fehler: auf zweiten Operator "*" folgt kein Operand
  - (Quelle: [8])

Überblick

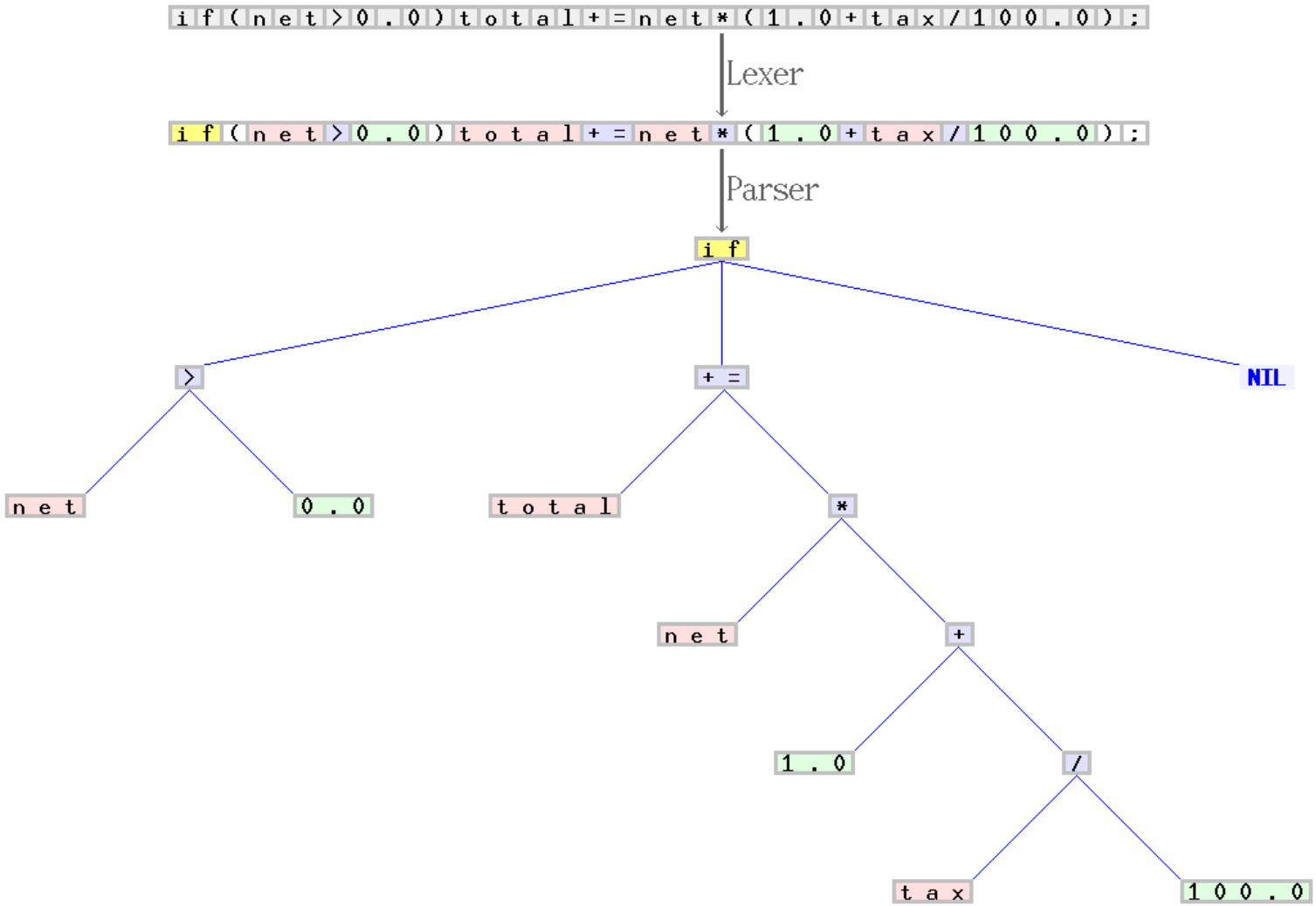

Abbildung: "Scanner and parser example for C" [5]

I. Semantische Analyse

- Prüfung auf *Funktionsweise* des Quellcodes
  - Datentyp-Überprüfung
  - *Object binding* (Variablen mit Typen verknüpfen, Methodenaufrufen folgen und verknüpfen)
  - Initialisierung aller Variablen Prüfen [7]
- Zentrale Prüfung: *"Funktioniert der Code?"*
- `double a = 3 * wurzel(5);`
  - *Könnte* semantisch inkorrekt sein, bspw. wenn *wurzel()* ein String-Literal zurückgibt.
  - (Quelle: [8])

II. Zwischencode-Erzeugung

- Erstes Übersetzen des Quellcodes
- maschinennahe Repräsentation
- Eine fundamentale Operation pro Zeile (keine Schleifen notwendig)
- unbegrenzte Speicheradressierung und Registeranzahl
  - Folge: langer, sehr unoptimierter Code

II. Zwischencode-Beispiel

- Quellcode:

```
p = i + r * 60;
```

- Zwischencode:

```
temp1 := 60
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

- $r \Leftrightarrow id3, i \Leftrightarrow id2, p \Leftrightarrow id1$
- (Quelle: [8])

II. Zwischencode-Optimierung

- semantisch äquivalenter, jedoch schnellerer bzw. kürzerer Code
  - toten Code entfernen
  - Schleifen abwickeln und Kontrollfluss ändern
  - Konstanten auflösen (Code: $(14/2)$, Optimiert: 7)
  - ggf. Anweisungen ersetzen [14]
- Optimierungsmöglichkeiten von verschiedenen Compilern stark unterschiedlich
- Vorsicht: Eingriff in die Semantik könnte Pragmatik verändern!

II. Zwischencode-Optimierung (cont.)

- Zwischencode:

```
temp1 := 60
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

- Optimierungsmöglichkeit:

```
temp1 = id3 * 60
id1 = id2 + temp1
```

- Reduktion auf die Hälfte der Anweisungen
- (Quelle: [8])

II. Maschinencode-Erzeugung

- Bestimmung der benötigten Register und Speichergröße
  - Abhängig von: Architektur des unterliegenden Systems
  - verwendeten Variablen
- Erzeugung der Maschinen-Befehle für die entsprechende System-Architektur basierend auf dem Zwischencode
- das fertige *Object-File* (.o) wird an den Linker weitergegeben

II. Maschinencode-Erzeugung (cont.)

- optimierter Zwischencode:

```
temp1 = id3 * 60
id1 = id2 + temp1
```

- Maschinencode (repräsentiert als Assembler):

```
MOV id3,R2
MUL 60,R2
MOV id2,R1
ADD R2,R1
MOV R1,id1
```

- Compiler wie *GNU GCC* übersetzen zuerst in Assembler-Sprache
  Andere Compiler hingegen direkt von Zwischencode in Maschinen-Sprache
- (Quelle: [8])

Zusammenfassung

- Analyse-Phase (Front-end): Quellcode auf Korrektheit prüfen
  - Lexikalische Analyse: *"Rechtschreibkorrektur"*
  - Syntaktische Analyse: Prüfung auf Syntax-Fehler
  - Semantische Analyse: *"Funktioniert das Programm?"*
- Synthese-Phase (Back-end): Maschinencode-Generierung
  - Zwischencode-Erzeugung: grobe Übersetzung
  - Programmoptimierung: Optimierung des Zwischencodes
  - Maschinencode-Erzeugung: Maschinensprache als *Object-File*

Linker

- Bindet nach Kompilierung alle *Object-Files* zu einem zentral ausführbaren Programm zusammen.
  - manchmal auch *Binder* genannt, früher in etwas anderer Funktionsweise auch *Loader*

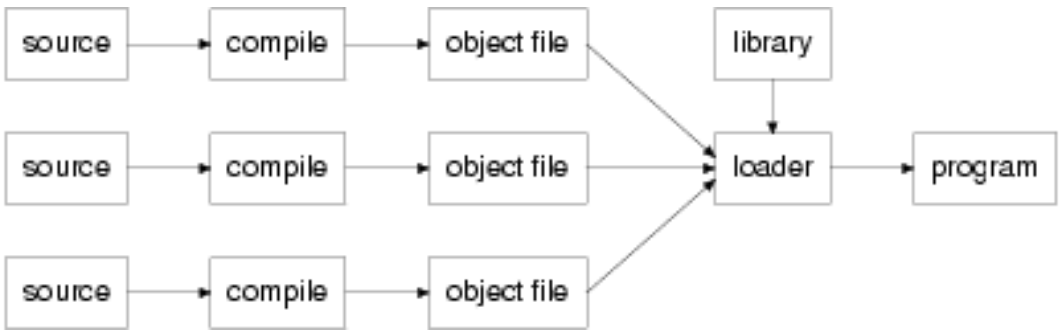

Abbildung: "Separate compilation" [2]

- Vorteil: Kein aufwendiges re-compiling des gesamten Projektes nötig, bei Veränderung einer Datei
- erneutes Linken ist wesentlich schneller [3]

Linker (cont.)

- Statisches Linken
  - einmaliges Linken
  - eine einzelne ausführbare Datei (portabel)
  - sämtliche Programmmodule in der Datei fest enthalten
  - Aber: hoher Speicherplatzbedarf & unflexibel [11]

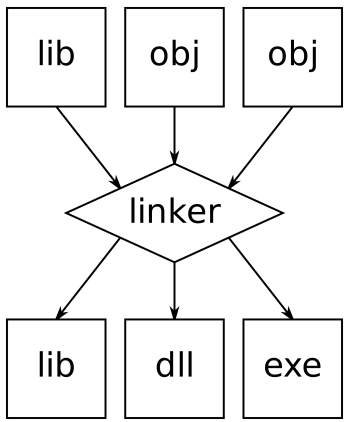

Abbildung: "Input and output file types of the linking process." [15]

Linker (cont.)

- Dynamisches Linken
  - *dynamically linked libraries* (DLL) werden bei Programmstart eingebunden
  - Speicherplatz wird nur einmal benötigt
  - leichtes Austauschen von DLLs (Flexibilität)
  - Aber: jedes System muss die entsprechenden DLLs fest installiert haben (nicht mehr portabel)
- Mischformen sind der Regelfall [11]

versch. Compiler und IDEs

- GNU GCC
  - verbreiteter Open-Source Compiler
  - in den meisten Linux-Distributionen mitgeliefert
  - auf Windows z.B. mit MinGW [12]
  - leichte Integration in IDEs (z.B. Eclipse CDT)
- Executable:

  ```
  gcc hello.c -o hello
  ```
- Optionen:
  - -c: nur *Object-Files* ohne Linken
  - -E: nur pre-processed Code ausgeben
  - -o: nachfolgender Parameter ist der Name der Executable
  - -O2: Optimierungsoptionen für die Code-Optimierung
- sehr flexibel und in aktiver Entwicklung!

Intel ICC

- sehr ausgereifter, kommerzieller Compiler
- viele automatisierte Optimierungsmöglichkeiten
- Aber: sehr teuer (mind. 700$)
  - in Linux kostenlos nutzbar unter *NonCommercial-Use License* [10]
- optimale Ergebnisse werden nur auf Intel CPUs erzielt [9]

Quellenverweise und Literatur I

[1] The C Book. *"Effect of the Standard"*. URL: http://publications.gbdirect.co.uk/c_book/chapter7/effect_of_the_standard.html (besucht am 23. 04. 2014).

[2] The C Book. *"Separate compilation"*. URL: http://publications.gbdirect.co.uk/c_book/figures/1.1.png (besucht am 23. 04. 2014).

[3] The C Book. *"The form of a C program"*. URL: http://publications.gbdirect.co.uk/c_book/chapter1/form_of_a_c_program.html (besucht am 23. 04. 2014).

Quellenverweise und Literatur II

[4] The C Book. *"The preprocessor"*. URL: http://publications.gbdirect.co.uk/c_book/figures/7.1.png (besucht am 23. 04. 2014).

[5] Jochen Burghardt. *"Scanner and parser example for C"*. URL: http://en.wikipedia.org/wiki/File:Xxx_Scanner_and_parser_example_for_C.gif (besucht am 23. 04. 2014).

[6] C-HowTo. *"Dateien einbinden"*. URL: http://www.c-howto.de/tutorial-praeprozessor-dateien-einbinden.html (besucht am 23. 04. 2014).

[7] *"Compiler"*. URL: http://en.wikipedia.org/wiki/Compiler#Front_end (besucht am 23. 04. 2014).

Quellenverweise und Literatur III

[8] Ulrich Helmich. *"Was ist ein Compiler?"* URL: http://www.u-helmich.de/inf/BlueJ/kurs131/Seite25/seite25-1.html (besucht am 23. 04. 2014).

[9] *"Intel C++ Compiler"*. URL: http://en.wikipedia.org/wiki/Intel_C++_Compiler (besucht am 23. 04. 2014).

[10] *"Intel Developer Zone"*. URL: https://software.intel.com/en-us/non-commercial-software-development (besucht am 23. 04. 2014).

[11] *"Linker (Computerprogramm)"*. URL: http://de.wikipedia.org/wiki/Linker_(Computerprogramm) (besucht am 23. 04. 2014).

Quellenverweise und Literatur IV

[12] *"MinGW"*. URL: http://www.mingw.org/ (besucht am 23. 04. 2014).

[13] Matt Neuburg. *"How a large C program is divided into files"*. URL: http://www.apeth.com/iOSBook/figs/pios_0102.png (besucht am 23. 04. 2014).

[14] *"Optimizing compiler"*. URL: http://en.wikipedia.org/wiki/Compiler_optimization (besucht am 23. 04. 2014).

[15] Qef. *"Input and output file types of the linking process"*. URL: http://commons.wikimedia.org/wiki/File:Linker.svg (besucht am 23. 04. 2014).

Quellenverweise und Literatur V

[16] Sommergut.de. *"C-Buch"*. URL: http://c-buch.sommergut.de/Kapitel12/Bedingte-Kompilierung.shtml (besucht am 23. 04. 2014).

[17] Jürgen Wolf. *"C von A bis Z"*. URL: http://openbook.galileocomputing.de/c_von_a_bis_z/010_c_praeprozessor_001.htm (besucht am 23. 04. 2014).