Grundlagen von C

— Ausarbeitung von Jonas Gresens —

Proseminar C Grundlagen und Konzepte
Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

1 Allgemein

Die imperative Programmiersprache C wurde von dem Amerikanischen Informatiker Dennis Ritchie an den Bell Laboratorien (Bell Labs) entwickelt und 1972 veröffentlicht. C ist als direkte Sprachabstraktion für Assembler gedacht und wird daher mitunter als „Hoch-Assembler“ bezeichnet, wobei dies durchaus gerechtfertigt ist, da in C mit direkten Speicherzugriffen sehr hardwarenah programmiert werden kann.

Bei der Entwicklung von C ließ sich Ritchie von älteren Sprachen wie BCPL (1966 veröffentlicht), ALGOL 68 (1968) und die von ihm selbst mitentwickelte Sprache B (1969) beeinflussen. Im Gegenzug hat C aktuelle Sprachen wie C++ (1985), Java (1995), C# (2001) und D (2007) beeinflusst.

C besitzt einen sehr kleinen Sprachkern, wodurch die Funktionalität von C ohne zusätzliche Bibliotheken sehr eingeschränkt ist. Jedoch vereinfacht dies die Entwicklung eines Compilers so sehr, dass C Programme gut portierbar sind. C ist nicht typsicher und die Modularisierung findet auf Dateiebene (über Header-Dateien) statt.
Verwendung findet C in der Systemprogrammierung, bei Programmierung von Mikrocontrollern und Compiler, sowie Interpreter anderer Sprachen z.B. ist die Java Virtual Machine in C geschrieben.

2 Entwicklung in C

Die Programmierung in C kann auf verschiedene Arten erfolgen und gliedert sich idealerweise in zwei Teile auf:

- Das **Schreiben** des Quellcodes kann mit jedem beliebigen Texteditor erfolgen und benötigt keine speziellen Funktionen, da C-Quellcode aus normalen Textsymbolen besteht. Trotzdem ist es oft ratsam einen Editor mit Features wie Syntax Highligthing wie beispielsweise *Sublime Text* oder *Notepad++* oder falls mehr Funktionen notwendig sind eine IDE wie *Visual Studio*, *Eclipse*, *NetBeans*, *Anjuta* oder *Geany* zu verwenden.
- Das **Übersetzen** des Quellcodes erfolgt durch einen Prä-Prozessor und Compiler, welcher den Quellcode erst durch Textersetzung bearbeitet und daraufhin in direkt auf der Maschine ausführbaren Binärcode übersetzt. Oft werden Prä-Prozessor und Compiler gemeinsam verteilt, ein Beispiel hierfür ist der viel genutzte „GNU C Compiler“ *gcc*.

3 Hello-World!

Am folgenden „Hello-World!“-Beispiel werden ein paar Grundlagen erklärt:

```c
/*
 * C_Grundlagen_Hello_World!.c
 */

#include <stdlib.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    //Print Hello World
    printf("Hello World!");

    return EXIT_SUCCESS;
}
```

Erklärung	Zeile
einzeilige Kommentare stehen hinter `//` mehrzeilige Kommentare werden zwischen `/*` und `*/` geschrieben	10 1-3
Einbinden von weiterem Code durch `#include ...`	5, 6
`main`-Methode als Einstiegspunkt des Programms • die Parameter dienen zum Übergeben von Argumenten • die Rückgabe erfolgt an das Betriebssystem	8
als Blockklammern werden `{` und `}` verwendet	9, 14
Methodenaufruf haben die Form `identifier(args ...)`	11
am Ende jedes Statements steht ein Semikolon (`;`) • mehrere Statements können durch Kommata (`,`) getrennt	11, 13

4 Programmierung in C (Teil 1)

4.1 Variablen

Wie in anderen imperativen Programmiersprachen auch gibt es in C Variablen zur Speicherung eines veränderbaren Werts und obwohl C eine schwache Typisierung hat, gibt es zum Beispiel folgende verschiedene Typen für die entsprechenden Werte:

Typ	Werte
`int`	ganz Zahl
`double`	Zahl mit Nachkommastellen
`char`	einzelnes Zeichen
`bool`	boolscher Wert (seit C99)

Es gibt in C keinen primitiven Datentyp für Zeichenketten (Strings), stattdessen werden sie als `char`-Arrays realisiert (`char[]`).

Bevor eine Variable z.B. im Programm modifiziert und oder als Parameter übergeben werden kann, durchläuft sie normalerweise die folgenden 3 Schritte:

- Die **Deklaration** dient der Vergabe eines Namens und Typs, sodass die Variable über ihren Namen im Programm referenziert werden kann und Typprüfungen schon zur Übersetzungszeit stattfinden können
  - bei Verwendung des Schlüsselworts `extern` findet die reine Deklaration der Variablen statt, dies dient der Modularisierung des Programms
- Die **Definition** ist die Reservierung des vom Typ abhängigen Speicherplatzes der Variablen, sodass sie zwingend immer die vorherige Deklaration voraussetzt.
- Die **Initialisierung** ist die Zuweisung eines ersten Werts

Der folgende Beispielcode zeigt die 3 genannten Schritte:

```
1 int i;        //Deklaration und Definition
2 extern int j; //nur Deklaration
3 i = 42;       //Initialisierung
4 char c = 'i'; //alles in einer Zeile
```

4.2 Ausgabe

Oftmals ist es notwendig Text auf dem Terminal oder der Konsole auszugeben, da wie bereits erwähnt der Sprachkern von C sehr klein ist und keine Methoden für die Textausgabe bereitstellt, erfordert dies das Einbinden der Standard-Input-Output Bibliothek. Das Einbinden erfolgt über die Prä-Prozessor-Anweisung `#include <stdio.h>`.

Die Ausgabe von Text erfolgt durch den Aufruf der Methode `printf`, wobei die Ausgabe durch die Parameter bestimmt wird:

- Soll ein **einfacher String**, wie z.B. `"Hello World"` ausgegeben werden, so erhält `printf` diesen String als einziges Argument:
  `printf("Hello World!");` ⇒ `Hello World!`
- Ist das Ziel die **Ausgabe einer Variablen** sein, so werden `printf` 2 Argumente übergeben: Das erste Argument ist ein String mit einem `%` als Platzhalter für die Variable und einem passenden conversion modifier (siehe unten). Das Zweite ist die Variable die ausgegeben werden soll:
  `printf("%d", i);` ⇒ `42`
- Kombinierte Ausgabe: Soll die Ausgabe mehrerer Variablen und weiterer Zeichen erfolgen, so lässt sich dies anstatt in entsprechend vielen Aufrufen von `printf` auch verkürzt in einem einzigen erledigen. Hierzu müssen an den jeweiligen Stellen im ersten Argument (im String) die entsprechenden Platzhalter eingefügt und daraufhin in der gleichen Reihenfolge, wie sie in der Ausgabe vorkommen sollen, die jeweiligen Variablen übergeben:
  `printf("%c hat den Wert %d", c, i)` ⇒ `i hat den Wert 42`

Über die bereits erwähnten **conversion modifier** lässt sich angeben, als welcher Typ die Variablen ausgegeben werden soll - dieser Typ muss nicht mit dem Typ der Variablen übereinstimmen:

Zeichen	Typ
`%d` oder `%i`	`int`
`%c`	`char`
`%f`	`double`

Manchmal ist es notwendig Steuerzeichen oder bereits in C verwendete Zeichen zu verwenden, hierzu ermöglicht C das **Escapen** mit dem Backslash (\). Oft genutzte zu escapende Zeichen sind:

Zeichen	Erklärung
`\n`	für einen Zeilenumbruch (newline)
`\t`	für einen Tab
`\\`	für das Zeichen \
`\"`	für das Zeichen "
`\0`	für das Ende des Strings

5 Programmierung in C (Teil 2)

Ein wichtiges Element in der Programmierung ist die Steuerung des Kontroll- oder Steuerflusses (control flow) welcher über Fallunterscheidungen und Schleifen beeinflusst werden kann.

5.1 Fallunterscheidung

In C existieren drei verschiedene Möglichkeiten zur Fallunterscheidung, die im folgenden alle erläutert werden:

5.1.1 if-else

Die einfachste Form einer **if**-Anweisung dient der bedingten Ausführung einer Anweisung und hat die folgende Syntax:

```
if (expr) statement;
```

Das `statement` wird ausgeführt, wenn die Prüfung erfolgreich ist (`expr` zu *true* ausgewertet wird).

Die nächste Stufe ist die bedingte Ausführung mit einer Alternative **if-else**, falls die Prüfung *false* zurückgibt:

```
if (expr)
    statement;
else
    statement;
```

Dies lässt sich beliebig oft **schachteln**, wodurch ein Entscheidungsbaum aufgespannt werden kann:

```
if (expr1)
    statement1;
else if (expr2)
    statement2;
else
    statement3;
```

In diesem Fall wird, wenn `expr1` *true* ist, `statement1` ausgeführt, ansonsten wird, wenn `expr2` *true* ist, `statement2` ausgeführt und wenn weder `statement1` noch `statement2` *true* sind, wird `statement3` ausgeführt.

Soll in einem Fall eine mehrzeilige Anweisung ausgeführt werden, so müssen diese Anweisungen in Blockklammern stehen.

5.1.2 conditional

Sollen in einem Fall nicht mehrere Anweisungen ausgeführt werden, sondern ein Wert zurückgeliefert werden, so eignet sich hierfür der ternärere conditinal-Operator `?`. Das conditional verknüpft 3 Elemente miteinander und stellt eine Art Kurzform von if-else dar: Das erste Element ist die zu prüfende Bedingung und die beiden folgenden die Werte, welche bei true oder false zurückzuliefer werden sollen. Die Syntax lautet:

```
bedingung ? wenn_wahr : wenn_falsch;
```

Im folgenden Beispiel wird der Wert, der ausgegeben werden soll, erst durch das conditional bestimmt und die Zahl 200 ausgegeben, da $5 >= 5$ gilt:

```
1 int i = 5;
2 printf("%d", (i >= 5) ? 200 : 300;
```

5.1.3 switch-case

Manchmal sind die genauen Werte einer Variablen im Voraus bekannt, an dieser Stelle ermöglicht das `switch-case` die bedingte Ausführung nach dem Vergleich mit verschiedenen konstanten Werten. Ein solcher `case` besteht dabei aus normalerweise drei einzelnen Elementen:

1. Dem Schlüsselwort `case` gefolgt von einem konstanten Wert
2. Den in diesem Fall auszuführenden Anweisungen
3. Dem Schlüsselwort `break`, welches das Ende des Falls markiert und ebenfalls das Verlassen des switch-case zur Folge hat.

Folgt auf ein `case` kein `break`, wird nach der Ausführung vom ersten `case` auch der nächste darauffolgende ausgeführt – der Vergleich am Anfang des nächsten `case` ist dabei irrelevant, wie folgendes Syntaxbeispiel zeigt:

```
switch (var) {
    case const1:    // case mit Wert
        statement1; // Anweisungen
        break;      // Ende des Cases
    case const2:
    case const3:
        statement2;
        break;
    default:
        statement3;
        [break;]    // optional
}
```

Gilt `var == const1`, so wird nur `statement1` ausgeführt.
Da auf `case const2` kein `break` folgt wird `statement2` ausgeführt, wenn `var == const2` oder `var == const3` gilt.
Gilt keiner der obigen Fälle, so wird `statement3` ausgeführt.

5.2 Schleifen

Bisher lassen sich mit Fallunterscheidungen nur Abzweigungen in den Kontrollfluss einbauen, manchmal ist es jedoch notwendig ein gewissen Codeabschnitt mehrfach auszuführen, hierfür stehen in den meisten Programmiersprachen Schleifen zur Verfügung:

5.2.1 for-Schleife

Die `for`-Schleife ermöglicht eine bedingte wiederholte Ausführung und bietet eine oder mehrere eingebettete Laufvariable(n), welche in den Anweisungen innerhalb der Schleife verwendet werden. Die Syntax einer `for`-Schleife ist die folgende:

```
for (vor_Beginn; Test_vor_Durchlauf; nach_Durchlauf) {
    statement;
}
```

Hinter dem `for` befindet sich der dreiteilige Schleifenkopf:

- `vor_Beginn` wird einmalig vor dem ersten Durchlauf der Schleife ausgeführt und enthält standardmäßig die Definition und Initialisierung der Laufvariable(n).
- `Test_vor_Durchlauf` ist die Prüfung, die vor jedem Durchlauf der Schleife durchgeführt wird: Bei *true* werden die Anweisungen innerhalb der Schleife ausgeführt und bei *false* der darauffolgende Code ausgeführt.
- `nach_Durchlauf` besteht aus den Anweisung zum verändern der Laufvariable(n), oft wird hierbei die Präfixform des Inkrement- oder Dekrement-Operator (`++i` und `--i`) verwendet.

Beispiel zur Ausgabe der Zahlen von 0 bis 9 mit einer `for`-Schleife:

```
1 for (int i = 0; i < 10; ++i) {
2     printf("%d ", i);
3 }
```

5.2.2 while-Schleife

Die gleichmächtige `while`-Schleife ist einfacher aufgebaut als die `for`-Variante:

```
while (Test_vor_Durchlauf) {
    statement;
}
```

Da es keine Laufvariable gibt besteht der Schleifenkopf nur aus dem Test, der vor jedem Durchlauf durchgeführt wird – ansonsten verhält sie sich gleich.

Beispiel zur Ausgabe der Zahlen von 0 bis 9 mit einer `while`-Schleife:

```
1 int i = 0;
2 while (i < 10) {
3     printf("%d ", i);
4     ++i;
5 }
```

5.2.3 do-while-Schleife

Die `do-while`-Schleife ist die fußgesteuerte Version der `while`-Schleife, d.h. die Prüfung findet anstatt vor dem Durchlauf erst vor dem Nächsten statt, so ist mindestens ein Durchlauf der Schleife garantiert, da vor dem allerersten Durchlauf keine Prüfung stattfinden kann:

```
do {
    statement;
} while (Test_nach_Durchlauf);
```

Beispiel zur Ausgabe der Zahlen von 0 bis 9 mit einer `do-while`-Schleife:

```
1 int i = 0;
2 do  {
3     printf("%d ", i);
4     ++i;
5 } while (i < 10);
```

Für diesen Schleifentyp gilt wie für jeden anderen auch: Wie beim `if` sind die Blockklammern nur notwendig, wenn die Anweisungen im Schleifeninneren mehrzeilig sind.

5.2.4 continue und break

Der Kontrollfluss durch Schleifen lässt sich mit den folgenden Schlüsselwörtern noch genauer steuern:

- `continue` ermöglicht das vorzeitige Beenden des aktuellen Durchlaufs und kann beispielsweise dazu verwendet werden, um zu verhindern dass eine unnötige Berechnung stattfindet, deren Ergebnis erst im nächsten Schleifendurchlauf verwendet wird, weil die Bedingung nicht mehr erfüllt ist.
- `break` ist mächtiger und verursacht das vorzeitige Verlassen der gesamten Schleife.

Literatur

„C-Programmierung“, http://de.wikibooks.org/wiki/C-Programmierung, 23.04.2014
„C (programming language)“, http://en.wikipedia.org/wiki/C_(programming_language), 01.05.2014
„Dennis Ritchie“, http://en.wikipedia.org/wiki/Dennis_Ritchie, 01.05.2014