Chair of Embedded Systems
Department of Computer Science
University of Augsburg, Germany

Martin Frieb

Universität Augsburg University

Wintersemester 2020/2021

Informatik 1

Kapitel 8 – Mehrteilige Programme

Inhaltsverzeichnis

Version 20201204-2155

8.1 Übersetzungseinheiten

8.1.1 Modularisierung von Programmen

Größere Programme werden in mehrere sog. **Übersetzungseinheiten** aufgeteilt, die zuerst getrennt übersetzt und dann wieder zu einem Programm zusammengesetzt werden. Dabei enthält genau eine der Übersetzungseinheiten die `main`-Funktion.
Jede Übersetzungseinheit besteht aus einer `.c`- mit zugehöriger **namensgleicher** `.h`-Datei. Dabei können aber mehrere `.c`-Dateien für eine Header Datei existieren.

Eine solche Aufteilung des Programms nennt man auch *Modularisierung*. Ein Grund für diese Trennung ist die bessere Wartbarkeit und Übersichtlichkeit. Eine riesige Datei mit 100.000 Zeilen Code ist unübersichtlich. Teilt man sie in viele, kleinere Einheiten auf, kann man dadurch für eine bessere Übersichtlichkeit sorgen. Damit ergibt sich auch eine bessere Wartbarkeit.

Modularisierung
Will man nun eine Datei in mehrere, kleine Daten aufteilen – also modularisieren – teilt man sie nicht willkürlich auf. Die Dateien sollen nämlich sinnvoll und zusammenhängende Gruppen von Vereinbarungen und Definitionen bilden. Sie sollen für sich funktionieren und ein allgemein wiederverwendbares **Modul** bilden. Damit können mehrfach benötigte Vereinbarungen in nur wenigen Modulen verwaltet werden und Module in mehreren verschiedenen Programmen wiederverwendet werden.

Header- und C-Dateien
Wie oben schon kurz angesprochen nutzt man für eine Modularisierung *Header* (.h Dateien) und C-Dateien (.c Dateien). Die Header Datei enthält **symbolische Konstanten**, sog. **Makros** und die **Funktions-Prototypen** der Übersetzungseinheit. Eine genauere und ausführlichere Erklärung von Header, Makros, etc. werden im Lauf des Kapitels behandelt.
Die dazugehörige C-Datei enthält die Funktions-Definitionen der Übersetzungseinheit.

Aus den im letzten Kapitel entwickelten Funktionen lässt sich zum Beispiel die folgende allgemein wiederverwendbare Übersetzungseinheit bilden:

- `input.c` und `input.h` als Sammlung von Eingabefunktionen (`read_pos, read_string, read_plz, ...`).

Die C-Datei mit der `main`-Funktion heißt **Programmdatei**.

Übersetzungseinheiten: Header-Dateien

Beispiel: Header-Datei `input.h` für Eingabefunktionen

```
/* Bedingte Aktivierung */
/* Das ifndef schützt den Benutzer. Der Zweck
   hierbei ist eine Abfrage, ob diese Header
   Datei schon im genutzen Code definiert wurde
   oder nicht. Falls ja, bricht man hier gleich
   ab. */
```

```c
#ifndef INPUT_H_INCLUDED
#define INPUT_H_INCLUDED

/* Andere Header einbinden */
#include <stdlib.h>

/* Symbolische Konstanten können hier definiert
   werden. Man kann nun INVALID_INPUT und
   READ_ERROR im eingebundenen Code verwenden. (
   Die Konstante EOF ist beispielsweise auch
   eine symbolische Konstante - und wurde
   genauso wie hier definiert) */
#define INVALID_INPUT -1
#define READ_ERROR -2

/* Makros - Das sind sehr kurze "Programme", die
    man später verwenden kann */
#define ERR_END(msg) { ERR(msg); exit(
   EXIT_FAILURE); }
#define ERR(msg) printf(msg)

/* Funktions-Prototypen. Hier werden
   Funktionsprototypen definiert. Sie werden in
   der input.c Datei implementiert. */
int read_pos_buff(void);
int flush_buff(void);

/* Das endif gibt an, dass man mit der Header
   Datei fertig ist. */
#endif
```

Übersetzungseinheiten: C-Dateien

Beispiel: C-Datei input.c für Eingabefunktionen

```c
/* Einbindung zugehöriger Header-Datei*/
#include "input.h"
/* Einbindung Bibliotheks-Header-Dateien*/
#include <stdio.h>
#include <limits.h>
#include <ctype.h>
/* Liste der Funktions-Definitionen. In der
   Header Datei wurden zwei Funktionsprototypen
   angegeben. Diese müssen hier implementiert
```

```
    werden, damit sie später genutzt werden
    können. */
/* read_pos_buff sowie flush_buff sollten aus
    Kapitel 7 bekannt sein. */
int read_pos_buff(void)
{
    ...
}

int flush_buff(void)
{
    ...
}
```

Zur Benutzung der symbolischen Konstanten, Makros und Funktionen einer Übersetzungseinheit wird deren Header-Datei in die Programm-Datei eingebunden. **Achtung:** Bei symb. Konstanten und glob. Variablen sei zu beachten, dass wenn sie aus mehreren Dateien (Header) zusammengeführt werden nur die erste von gleichnamingen Ïmports"genutzt wird.

Beispiel: Programm-Datei

```
1 #include <stdio.h>
2 #include <limits.h>
3 /* Einbindung Übersetzungseinheit */
4 #include "input.h"
5 /* main-Funktion */
```

8.1.2 Mehrteilige Programme compilieren

Übersetzungseinheit ohne `main`-Funktion compilieren:
Eine Übersetzungseinheit `<modul>.c` ohne `main`-Funktion wird in eine sog. **Objektdatei** `<modul>.o` übersetzt durch Benutzung der `-c`-Option des `gcc`:
`gcc -c <modul>.c`

Beispiel:

In einer anderen Ansicht mit `input.c` und `input.h` veranschaulicht:

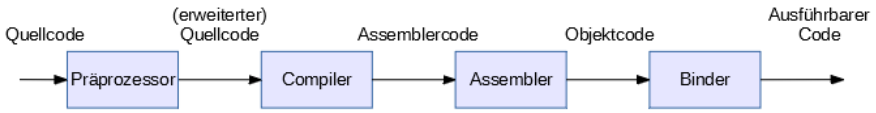
```
C:\Users\Edmin\Documents\Bsp>gcc -c input.c input.h
```

Programmdatei compilieren und mit Übersetzungseinheiten zusammensetzen:
Eine Programmdatei `<main>.c`, in der n Übersetzungseinheiten `<modul_1>.c`, ..., `<modul_n>.c` benutzt werden, wird wie folgt zusammen mit den zu den Übersetzungseinheiten gehörenden Objektdateien `<modul_1>.o`, ..., `<modul_n>.o` in Maschinencode übersetzt:
`gcc <modul_1>.o ... <modul_n>.o <main>.c`

> Beispiel:
>
> In einer anderen Ansicht mit `input.o` und `input_main.c` veranschaulicht:
>
> ```
> C:\Users\Edmin\Documents\Bsp>gcc input.o input_main.c
> ```

Der Compilierungsprozess besteht aus den folgenden Teilschritten:

1. Der **Präprozessor** übersetzt den **Quellcode** in den **erweiterten Quellcode**. Darin sind symbolische Konstanten und Makros aufgelöst (d.h. an die entsprechenden Stellen kopiert), ebenso werden Inhalte von Dateien, die via `#include` eingebunden werden, in die Datei kopiert.
2. Der **Compiler** übersetzt den **erweiterten Quellcode** in den **Assemblercode**, d.h. Maschinencode für die jeweilige Plattform – hier noch in *menschen*lesbarer Form.
3. Der **Assembler** übersetzt den **Assemblercode** in den **Objektcode**. Das ist eine 1:1 Umsetzung des Assemblercodes – nur diesmal in *maschinen*lesbarer Form. Dieser Objektcode lässt sich jedoch nicht ausführen – er enthält Verweise z.B. auf Bibliotheksfunktionen, mit denen er noch verbunden werden muss.
4. Der **Binder (Linker)** verbindet den Objektcode von Übersetzungeinheiten, Programmdatei und Bibliotheksfunktionen zu einer **ausführbaren Programmdatei**.



Mittels verschiedener `gcc`-Optionen lassen sich die Teilschritte des Compilierungsprozesses auch einzeln ausführen:

1. Ausführung des **Präprozessors**:
   Quellcode `<code>.c` in erweiterten Quellcode überführen:
   `gcc -E <code>.c`
2. Ausführung von **Präprozessor und Compiler**:
   Quellcode `<code>.c` in Assemblercode `<code>.S` überführen:
   `gcc -S <code>.c`
3. Ausführung von **Präprozessor, Compiler und Assembler**:
   Quellcode `<code>.c` in Objektcode `<code>.o` überführen:
   `gcc -c <code>.c`

Alternative Möglichkeit zum Compilieren mit Header Dateien
Alternativ kann eine Programmdatei `<main>.c` und n verschiedene Header Dateien in einem Befehl kompiliert werden:
`gcc <header_1>.h ... <header_n>.h <main>.c`

> Beispiel:
>
> Auch hier wieder in einer anderen Darstellung veranschaulicht:
>
> ```
> C:\Users\Edmin\Documents\Bsp>gcc input.c input.h input_main.c
> ```

8.2 Header-Dateien

Was ist eine Header-Datei?
Was steht in der Header-Datei?
Eine **Header-Datei** ist eine Textdatei mit der Endung `.h`, die folgende Informationen zu einer Übersetzungseinheit enthält:

- Funktions-Prototypen
- Symbolische Konstanten
- Makros

Was steht **nicht** in der Header-Datei?

- Funktionsdefinitionen: stehen in der c-Datei der Übersetzungseinheit
- Bibliotheksfunktionen: liegen als vorübersetzte Programme vor und werden mit dem Compiler installiert

Wie wird eine Header-Datei benutzt?
Einbindung von Header-Dateien:
Eine **Header-Datei** `beispiel.h` wird wie folgt in eine Quellcode-Datei **eingebunden**:

- Falls sich `beispiel.h` im include-Verzeichnis des Compilers befindet (Standard-Bibliothek):
  `#include <beispiel.h>`
- Falls sich `<Header>.h` im Programmverzeichnis befindet (eigener Header):
  `#include "beispiel.h"`

Verarbeitung:

- Das Einbinden von Header-Dateien wird durch den Präprozessor verarbeitet
- Veranlasst den Präprozessor, die Header-Datei in den Quellcode **zu kopieren**

Wie ist eine Header-Datei aufgebaut?
Inhalt einer Header-Datei `Header.h`:

```
1 #ifndef HEADER_H_INCLUDED
2 #define HEADER_H_INCLUDED
3
4 <Liste anderer Header-Dateien>
5 <Liste der symbolischen Konstanten>
6 <Liste der Makros>
7 <Liste der Funktions-Prototypen>
8
9 #endif
```

- Zeilen 1, 2, 9: **Bedingte Aktivierung**:
  Die Datei wird **genau einmal** in den erweiterten Quellcode kopiert, auch wenn er im Quellcode mehrmals mit `#include` eingebunden wird (`ifndef` = if not defined)
- Zeile 2: Fasst die nachfolgenden Deklarationen unter dem Namen `HEADER_H_INCLUDED` zusammen

Es können aber auch `.h`-Dateien existieren, die nicht zu genau einer `.c`-Datei gehören. Somit kann man beispielsweise Konstanten mittels `#define` für mehrere C-Dateien verwenden und einbinden.

> Beispiel:
>
> Ein kurzes Beispiel wäre beispielsweise die eigene Defintion von π. Will man ein ungenaues Leben führen, kann man bspw. in einer Header Datei $\pi = 3$ definieren. Bindet man nun diese Header Datei in den eigenen `.c`-Code ein, lässt sich dieses eigens definierte Pi überall verwenden.

8.3 Symbolische Konstanten

Symbolische Konstanten wurden im Wesentlichen schon in Kapitel 6.2.4 im Skript behandelt, nicht jedoch im Foliensatz.

Was ist eine Symbolische Konstante?
Vereinbarung:
Eine **symbolische Konstante** `<Konstante>` mit **Wert** `<Wert>` wird wie folgt vereinbart:
`#define <Konstante> <Wert>`

Verarbeitung:

- Symbolische Konstanten werden durch den Präprozessor verarbeitet.
- Jedes Vorkommen von `<Konstante>` im Quellcode außer in Namen und Zeichenketten wird durch `<Wert>` ersetzt.

Beispiel:

`#define PI 3.14`
Ersetzt jedes Vorkommen von `PI` durch `3.14`

Wie werden Symbolische Konstanten benutzt?
Benutzung:
`#define <Konstante> <Wert>`

- Benutze `<Konstante>` als lesbaren Namen für den konstanten Wert `<Wert>`.
- Namenskonvention: Verwende nur Großbuchstaben und das '_'-Zeichen.
- Einsatzmöglichkeiten: Längen von Feldern und Zeichenketten, besondere Rückgabewerte von Funktionen für Erfolgs- oder Fehlerfälle, Konstanten für mathematische Berechnungen

Beispiel:

Sei nun das PI vom oberen Beispiel in einer `beispiel.h` Datei definiert worden. Dann kann man es, wie uns schon bekannt, aufrufen.

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

int main(void)
{
    /* Gibt den Wert für PI aus: 3.14 */
    printf("PI: %f", PI);

    return 0;
}
```

Bewertung:

- Konstanten muss man bei Bedarf nur an einer Stelle ändern
- Programm ist besser lesbar
- Konfiguration mehrerer Programme durch einmalige Deklaration in Header-Datei möglich
- Symbolische Konstanten können für die Angabe von Feldlängen verwendet werden – im Gegensatz zu konstanten Variablen (z.B. `const int variable`)

8.4 Makros

Was ist ein Makro?
Vereinbarung:
Ein **Makro** `<Name_des_Makros>` mit **Parametern** `p_1,...,p_n` und **Ausdruck** `<Ausdruck>` wird

wie folgt vereinbart:
`#define <Name_des_Makros>(p_1,...,p_n)<Ausdruck>`

Man kann ein Makro als eine Art „sehr kurze Funktion“ ansehen, welche auch als solche verwendet werden kann.

Verarbeitung:

- Makros werden durch den Präprozessor verarbeitet.
- Jedes Vorkommen von `<Name_des_Makros>(a_1,...,a_n)` mit Ausdrücken `a_1,...,a_n` im Quellcode außer in Namen und Zeichenketten wird durch `<Wert>` ersetzt. Dabei wird jedes Vorkommen von `p_i` in `<Ausdruck>` durch `a_i` ersetzt.

Es gilt hierbei zu beachten: Die verwendeten Parameter sind Typ-unabhängig. Das bedeutet, es ist nicht festgelegt welchen Datentyp die Parameter `p_1,...,p_n` haben müssen. Es kann also ein `int`, `float`, etc. übergeben werden (vgl. Programmiersprache Python: Dort müssen auch keine Datentypen für Variablen angegeben werden).

Wie werden Makros benutzt?

Benutzung: `#define <Name_des_Makros>(p_1,...,p_n)<Wert>`

- Benutze `<Name_des_Makros>(a_1,...,a_n)` als Aufruf einer Funktion, die durch `<Wert>` definiert ist
- **1. Klammerungsregel**: Jeden Parameter `p_i` in `<Wert>` einklammern
  → Der für `p_i` eingesetzte Ausdruck `a_i` wird zuerst ausgewertet
- **2. Klammerungsregel**: `<Wert>` einklammern
  → Das Makro wird als Teil eines größeren Ausdrucks zuerst ausgewertet

> Beispiel: Gut definiertes Makro
>
> ```
> #define maximum(a, b) (((a) > (b)) ? (a) : (b))
> ```
>
> - Ersetzt `maximum(-x,0)` im Quellcode durch:
>   `(((-x) > (0)) ? (-x) : (0))`
> - Die Ersetzung erfolgt auch in Ausdrücken:
>   Ersetzt `1 + maximum(-x,0)` im Quellcode durch:
>   `1 + (((-x) > (0)) ? (-x) : (0))`

Schlecht definiertes Makro

`#define quad(a) a * a`
Ersetzt `quad(1 + 2)` im Quellcode durch:
`1 + 2 * 1 + 2` (Auswertung führt zu falschem Ergebnis)
⇒ Klammerung verbessern: `#define quad(a) ((a) * (a))`

Makros müssen keine Parameter haben

Beispiel:

Ein sehr kurzes, komplettes Beispiel könnte folgendermaßen aussehen:

`beispiel.h`:

```
#ifndef BEISPIEL_H_INCLUDED
#define BEISPIEL_H_INCLUDED

/* Die leeren runden Klammern sind nicht
   zwingend nötig und können auch weggelassen
   werden. Wir lassen sie jedoch dran, um sie
   besser von symbolischen Konstanten
   unterscheiden zu können. */
#define MAKRO_PRINT() printf("Hello world")

#endif
```

`main.c`:

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

int main(void)
{
    /* Gibt die Zeichenkette "Hello world" aus
       */
    MAKRO_PRINT();

    return 0;
}
```

- Makros können alle Arten von Anweisungen enthalten.
- Ein Makro kann ein anderes (vorher definiertes) Makro benutzen.

Beispiel:

```
#define ERR_END(msg) { ERR(msg); exit(
   EXIT_FAILURE); }
#define ERR(msg) printf(msg)
```

- `void exit(int status)`
  Funktion aus `stdlib.h`
  Beendet das Programm normal mit Rückgabewert `status`
- Falls das Makro nicht Teil eines Ausdrucks sein kann (da bestehend

aus einer Anweisungssequenz):
Eigenen Gültigkeitsbereich definieren durch **Einschluss in geschweifte Klammern**.

Beispiel: Beispiele aus `stdio.h` und `stdlib.h`

Im Kapitel 8.3 haben wir symbolische Konstanten kennengelernt. Es gilt jedoch zu beachten, dass auch unter solchen symbolischen Konstanten ein Makro verstanden wird. So sind beispielsweise die Konstanten `EOF`, `EXIT_SUCCESS`, `...` auch Makros.

- `EOF`
  Signalisiert Pufferfehler.
  Möglicher Rückgabewert von `scanf` und `getchar`.
  Systemabhängig definiert (hat oft den Wert -1).
- `EXIT_SUCCESS`
  Signalisiert erfolgreichen Programmverlauf.
  Wird als Rückgabewert von `main` benutzt.
  Systemabhängig definiert (hat oft den Wert 0).
- `EXIT_FAILURE`
  Signalisiert fehlerhaften Programmverlauf.
  Wird als Rückgabewert von `main` benutzt.
  Systemabhängig definiert (hat oft den Wert 1).

Worauf man noch achten sollte:
Beispiel: `#define quad(a) ((a) * (a))`
Vergleich zu Funktionen:

- Auf **Nebeneffekte** achten: Da ein Ausdruck an mehreren Stellen für denselben Paramter eingesetzt werden kann, wird dieselbe Rechnung öfter ausgeführt.
  Beispiel: Aufruf `quad(++i)` wird ersetzt durch `((++i) * (++i))` (`i` wird zweimal erhöht).
- Es entsteht **mehr Programmcode**, da **alle** Vorkommen des Makros ersetzt werden.
- Es ist **keine Speicherverwaltung** nötig.

8.5 Lokale Variablen (Wiederholung)

Definition: 8.8 Anweisungsblock

Ein Anweisungsblock in C ist von der Form

```
{
    <Anweisungen>
}
```

Wir kennen schon verschiedene Anweisungsblöcke:

- `if`-Block, `else`-Block, `else if`-Block
- `for`-Block, `while`-Block, `do-while`-Block
- Funktionsrümpfe

Anweisungsblöcke können ineinander **verschachtelt** werden.

> Definition: 8.9 Gültigkeitsbereich
>
> Jeder Anweisungsblock erzeugt einen sog. **Gültigkeitsbereich für lokale Variablen**

Was ist eine lokale Variable? (Wiederholung)

> Definition: 8.10 Lokale Variable
>
> - In einem Gültigkeitsbereich deklarierte Variablen heißen für diesen Bereich **lokal**.
> - Lokale Variablen existieren **nur während der Ausführung** des Anweisungsblocks – die Speicherverwaltung erfolgt **automatisch**:
>   - Reservieren von Speicherplatz durch Deklaration
>   - Freigeben von Speicherplatz am Ende des Blocks
>
>   Sie können also nur in ihrem Block und dessen inneren Blöcken verwendet werden.
> - Vor der ersten Wertzuweisung hat eine lokale Variable einen **zufälligen Wert** (**alte Bits** am zugewiesenen Speicherplatz).
> - Lokale Variablen werden auf dem **Stack** gespeichert.

Beispiel:

```c
int main(void)
{
    /* i ist eine lokale Variable und gilt für
       die komplette main */
    int i = 3;

    while (i < 10) {
        /* x ist eine lokale Variable und gilt
           nur innerhalb der while-Schleife. Sie
            wird mit jeder Iteration neu
           erstellt. */
        int x = 1;

        /* Man kann erfolgreich mit x arbeiten.
           Es wird immer 1 ausgegeben. */
        printf("%i", x);
```

```
            /* Sowie auch erfolgreich mit i arbeiten
                */
            i++;

            /* Jede Änderung von x wird mit einer
               neuen Iteration zurückgesetzt. Die
               folgende Zeile kann daher weggelassen
                werden. */
            x++;
        }

        /* Man kann außerhalb der Schleife NICHT
           mehr erfolgreich mit x arbeiten. Gibt
           eine Fehlermeldung aus. */
        printf("%i", x);
        /* Aber mit i kann man erfolgreich arbeiten
           */
        printf("%i", i);

        return 0;
    }
```

8.6 Globale Variablen

Was ist eine globale Variable?

> Definition: 8.11 Globale Variable
>
> Eine **globale Variable** ist eine Variable, die in der C-Datei einer Übersetzungseinheit oder in der Programmdatei **außerhalb aller Funktionsrümpfe** deklariert wird. Globale Variablen
>
> - haben nach der Deklaration automatisch den Wert Null ihres Datentyps
> - werden nicht im Stack, sondern im **Datenteil** gespeichert
> - können in **allen lokalen Gültigkeitsbereichen** verwendet und manipuliert werden und behalten ihren Wert während der **kompletten Programmlaufzeit**

So wenig wie möglich benutzen (Übersichtlichkeit, Speicherverwaltung): Benutze eine globale Variable **nur**, um **funktionsübergreifende Daten** zu speichern und zu manipulieren.

Externe Variablen

Globale Variablen können Übersetzungseinheit-übergreifend benutzt werden.

Definition: 8.12 Externe Variable

Eine in einer C-Datei `code1.c` deklarierte globale Variable `v` vom Typ `T` kann **in eine andere** C-Datei `code2.c` wie folgt als **externe Variable** eingebunden werden:
`extern T v;`

- Variablenname und Typ müssen in beiden C-Dateien gleich sein.
- Es wird nur einmal Speicher für `v` reserviert, nämlich durch die Deklaration in `code1.c`. Die `extern`-Deklaration in `code2.c` führt zu keiner weiteren Speicherreservierung.
- Anwendungsfall: Deklariere eine globale Variable in einer Übersetzungseinheit und verwende diese in der Programmdatei als externe Variable.

Beispiel:

```c
/* Eine globale Variable. Kann in jeder Funktion
    innerhalb dieser Datei genutzt werden */
int N = 0;

/* Eine externe Variable. Kann in jeder Datei
   benutzt werden */
extern int M;

void foo(void)
{
    /* N und M kann man bspw. in dieser Funktion
        nutzen */
    printf("%i, %i", N, M);
}

int main(void)
{
    /* Oder in einer anderen Funktion nutzen */
    printf("%i, %i", N, M);

    return 0;
}
```

Lokale und globale Konstanten

`#define KONSTANTE N`:

- Symbolische Konstante

- Kann als globale Konstante in C-Dateien eingebunden werden
- lob. Variable vs symbolische Konstante: Variable überall aenderbar; symbolische Konst. immer fest
- Ist **keine** Variable: es wird kein Speicher belegt, Adressoperator kann nicht angewendet werden
- Ist nicht typsicher: für `N` können Konstanten verschiedenen Typs eingesetzt werden
- Wird vom Präprozessor verarbeitet: Verändert den Quellcode

`const T konstante = N;`:

- „echte“ Variable mit konstantem Wert
- Kann als lokale und globale Konstante benutzt werden
- Ist eine Variable: es wird Speicher belegt, Adressoperator kann angewendet werden, ist typsicher
- Kann u.a. nicht für Feldlängen benutzt werden
- Wird vom Compiler verarbeitet
- Ist in C *keine wirkliche Konstante*: es ist möglich den Wert über andere Variablennamen zu ändern (siehe Kapitel 9 zu Zeigern)

8.7 Statische Variablen

Was ist eine statische Variable?

Definition: 8.13 statische Variable

Eine **statische Variable** ist eine Variable, die wie folgt deklariert wird:
`static <T> <Variable> = N;`

Statische Variablen

- müssen in der Deklaration mit einem konstanten Wert `N` initialisiert werden.
- werden nicht im Stack, sondern im **Datenteil** gespeichert.
- können lokal oder global deklariert werden.
- behalten ihren Wert während der kompletten Programmlaufzeit.

Lokale vs. globale statische Variablen
Lokale statische Variablen:
Lokale statische Variablen werden **in einem Funktionsrumpf** deklariert.

- Sie **behalten ihren jeweils letzten Wert** nach Ende der Abarbeitung der Funktion
- Sie können nur in der Funktion verwendet und manipuliert werden, in der sie deklariert wurden

Globale statische Variablen:
Globale statische Variablen werden **außerhalb aller Funktionsrümpfe** deklariert.

- Sie können in jeder Funktion der C-Datei verwendet und manipuliert werden
- Sie **behalten ihren jeweils letzten Wert**
- Sie können **nicht Übersetzungseinheit-übergreifend** verwendet werden (Unterschied zu normalen globalen Variablen)

Beispiele für lokale statische Variablen

Beispiel:

```c
void foo(void)
{
    /* n ist eine statische Variable. Dadurch
       bleiben Veränderungen von n erhalten */
    static int n = 0;
    printf("%i", n);
    n++;
}

int main(void)
{
    /* Der erste foo Aufruf. n ist beim ersten
       Aufruf 0 und es wird 0 ausgegeben */
    foo();

    /* Der zweite foo Aufruf. n wurde schonmal
       erstellt und mit dem ersten Aufruf um 1
       erhöht. Somit wird hier 1 ausgegeben */
    foo();

    /* Selbiges wie oben. n wurde von 1 auf 2
       erhöht, weshalb hier 2 ausgegeben wird.
       */
    foo();

    return 0;
}
```

Zufallszahlen-Generator:
Ein **Zufallszahlen-Generator** für eine Zahlenmenge M besteht aus

- einem Startwert $x_1 \in M$.
- einer **Nachfolger-Funktion** $f : M \to M$.

Durch die Regel

$$x_{n+1} := f(x_n) \ (n \in \mathbb{N})$$

wird eine Folge von sog. **Pseudo-Zufallszahlen** generiert (d.h. aus der n-ten Zahl wird die $(n+1)$-te Zahl berechnet).

Die Nachfolger-Funktion f wird so gewählt, dass sich die Pseudo-Zufallszahlen möglichst willkürlich (zufällig) über M verteilen.

Für die Generierung einer Folge von Pseudo-Zufallszahlen verwendet man eine statische Variable. Jeder Funktionsaufruf generiert aus der letzten Zufallszahl die nächste Zufallszahl:

Beispiel: Zufallszahlen-Generator mit festem Startwert

```
1 unsigned int myintrand()
2 {
3     static unsigned int number = 45644641; /*
          Startwert */
4     number = number * (number + 3); /*
          Nachfolger-Funktion */
5     return number;
6 }
```

- Zeile 3: Initialisierung – wird **nur einmal** ausgeführt
- Zeile 4: Ein Aufruf von `myintrand` erzeugt, beginnend mit dem Startwert, aus dem vorherigen Wert den nächsten Wert bzgl. der Nachfolgerfunktion $f(n) := n \cdot (n+3)$ (mit zyklischem Bereichsüberlauf)
- Für die Erzeugung von n Zufallszahlen muss man `myintrand` n-mal aufrufen.
- Startwert ist hier fest

Beispiel für globale statische Variablen
Bibliotheksfunktionen für die Generierung von Pseudo-Zufallszahlen:

Beispiel: `rand`

Die `stdlib.h`-Bibliotheksfunktion
`int rand(void)`
liefert bei jedem Aufruf eine ganzzahlige Pseudo-Zufallszahl im Bereich von `0` bis `RAND_MAX`

Da man mit `srand` den Startwert neu setzen kann, wird hier in der Implementierung eine **globale statische Variable** benutzt.

`RAND_MAX` ist eine symbolische, systemabhängige Konstante mit dem Mindestwert `32767`

Bibliotheksfunktionen für die Generierung von Pseudo-Zufallszahlen:

Beispiel: `srand`

Die `stdlib.h`-Bibliotheksfunktion
`void srand(unsigned int seed)`
setzt `seed` als Startwert für eine neue Folge von Zufallszahlen. (Beachte: Das ist nicht die echte rand-Funktion, wie sie in C implementiert ist. Sie wurde der Anschaulichkeit vereinfacht.)

In der Implementierung wird wie folgt eine **globale statische Variable benutzt**:

```
1  static unsigned int next = 1U;
2  void srand (unsigned int seed)
3  {
4      next = seed;
5  }
6
7  int rand ()
8  {
9      next = (next * (next + 3)) % RAND_MAX;
10     return next;
11 }
```