Vorlesung Informatik 1 (Wintersemester 2020/2021)

Kapitel 8: Mehrteilige Programme

Martin Frieb
Johannes Metzger

Universität Augsburg
Fakultät für Angewandte Informatik

9. Dezember 2020

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

Modularisierung

- Übersetzungseinheiten werden als sinnvoll zusammenhängende Gruppen von Vereinbarungen und Definitionen gebildet
- Eine Übersetzungeinheit soll so ein funktional allgemein wiederverwendbares **Modul** bilden
- Damit können mehrfach benötigte Vereinbarungen in nur wenigen Modulen verwaltet werden und Module in mehreren verschiedenen Programmen wiederverwendet werden

Modularisierung von Programmen

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

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

`.h`-Dateien (Header-Dateien)

Die Header-Datei enthält **symbolische Konstanten**, sog. **Makros** und die **Funktions-Prototypen** der Übersetzungseinheit.

`.c`-Dateien (C-Dateien)

Die C-Datei enthält die Funktions-Definitionen der Übersetzungseinheit.

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

Übersetzungseinheiten: Header-Dateien

Beispiel 8.1 (Header-Datei `input.h` für Eingabefunktionen)

```
1  /* Bedingte Aktivierung */
2  #ifndef INPUT_H_INCLUDED
3  #define INPUT_H_INCLUDED
4  /* Andere Header einbinden */
5  #include <stdlib.h>
6  /* Symbolische Konstanten */
7  #define INVALID_INPUT -1
8  #define READ_ERROR -2
9  /* Makros */
10 #define ERR_END(msg) { ERR(msg); exit(EXIT_FAILURE); }
11 #define ERR(msg) printf(msg)
12 /* Funktions-Prototypen */
13 int read_pos_buff(void);
14 int flush_buff(void);
15 #endif
```

Übersetzungseinheiten: C-Dateien

Beispiel 8.2 (C-Datei `input.c` für Eingabefunktionen)

```
1 /* Einbindung zugehöriger Header-Datei*/
2 #include "input.h"
3 /* Einbindung Bibliotheks-Header-Dateien*/
4 #include <stdio.h>
5 #include <limits.h>
6 #include <ctype.h>
7 /* Liste der Funktions-Definitionen */
```

Zur Benutzung der symbolischen Konstanten, Makros und Funktionen einer Übersetzungseinheit wird deren Header-Datei in die Programm-Datei eingebunden

Beispiel 8.3 (Programm-Datei)

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

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
```

Programmdatei compilieren und mit Übersetzungseinheiten zusammensetzen

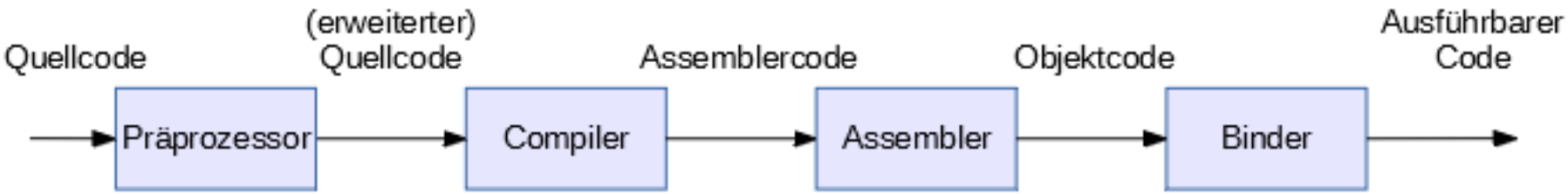
Eine Programmdatei `<main>.c`, in der *n* Übersetzungseinheiten `<modul_1>.c`, ..., `<modul_n>.c` benutzt werden, wird wie folgt zusammen mit den zu den Übersetzungseinheiten gehörenden Objektdateien `<modul_1>.o`, ..., `<modul_n>.o` in Maschinencode übersetzt:

```
gcc <modul_1>.o ... <modul_n>.o <main>.c
```

Mehrteilige Programme compilieren

Der Compilierungsprozess besteht aus den folgenden Teilschritten:

1. Der **Präprozessor** übersetzt den **Quellcode** in den **erweiterten Quellcode** (siehe folgende Folien)
2. Der **Compiler** übersetzt den **erweiterten Quellcode** in den **Assemblercode**
3. Der **Assembler** übersetzt den **Assemblercode** in den **Objektcode**
4. Der **Binder (Linker)** verbindet den Objektcode von Übersetzungeinheiten, Programmdatei und Bibliotheksfunktionen zum **Maschinencode**



Mehrteilige Programme compilieren

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`

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** `<Header>.h` wird wie folgt in eine Quellcode-Datei **eingebunden**:

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

8. Mehrteilige Programme

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 8.4

```
#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

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

8. Mehrteilige Programme

Was ist ein Makro?

Vereinbarung

Ein **Makro** `<Makro>` mit **Parametern** `p_1,...,p_n` und **Wert** `<Wert>` wird wie folgt vereinbart:

```
#define <Makro>(p_1,...,p_n)<Wert>
```

Verarbeitung

- Makros werden durch den Präprozessor verarbeitet
- Jedes Vorkommen von `<Makro>(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 `<Wert>` durch `a_i` ersetzt

Wie werden Makros benutzt?

Benutzung

```
#define <Makro>(p_1,...,p_n)<Wert>
```

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

Wie werden Makros benutzt?

Beispiel 8.5 (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)

Wie werden Makros benutzt?

Makros müssen keine Parameter haben

Beispiel 8.6 (Beispiele aus `stdio.h` und `stdlib.h`)

- `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)

Wie werden Makros benutzt?

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

Beispiel 8.7

```
#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**.

Bewertung

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)
- Parameter sind **nicht typ-abhängig**: Ein Makro kann Parameter unterschiedlichen Typs verarbeiten (im Gegensatz zu Funktionen)
  Beispiel: Aufrufe `quad(2)` und `quad(2.5)` möglich
- Es entsteht **mehr Programmcode**, da **alle** Vorkommen des Makros ersetzt werden
- Es ist **keine Speicherverwaltung** nötig

Anweisungsblöcke (**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

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.

Lokale und globale Konstanten

```
#define KONSTANTE N
```

- Kann als globale Konstante in C-Dateien eingebunden werden
- 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;
```

- 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 späteres Kapitel zu Zeigern)

8. Mehrteilige Programme

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)

Beispiel für lokale statische Variablen

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.

Beispiel für lokale statische Variablen

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

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:

`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 (nächste Folie)

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

Beispiel für globale statische Variablen

Bibliotheksfunktionen für die Generierung von Pseudo-Zufallszahlen:

srand

Die `stdlib.h`-Bibliotheksfunktion
`void srand(unsigned int seed)`
setzt `seed` als Startwert für eine neue Folge von Zufallszahlen

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 }
```