05 - Gültigkeitsbereiche

Peter Ulbrich

AG Systemsoftware

Veranstaltungswebseite

Rückblick: Trennung von Deklaration und Definition

- Analog zu Variablen: Vor der ersten Benutzung müssen Funktionen bekannt sein
- Unterscheidung zwischen Deklaration und Definition
- Deklaration macht Signatur bekannt
  - Rückgabetyp Bezeichner(Parameter);
  - Beispiel: `int sum(int a, int b);`
- Definition spezifiziert die Anweisungen

  → Funktionsrumpf, als Code-Block `{}`
- Deklaration und Definition können getrennt oder zusammen erfolgen

Rückblick: Trennung von Deklaration und Definition

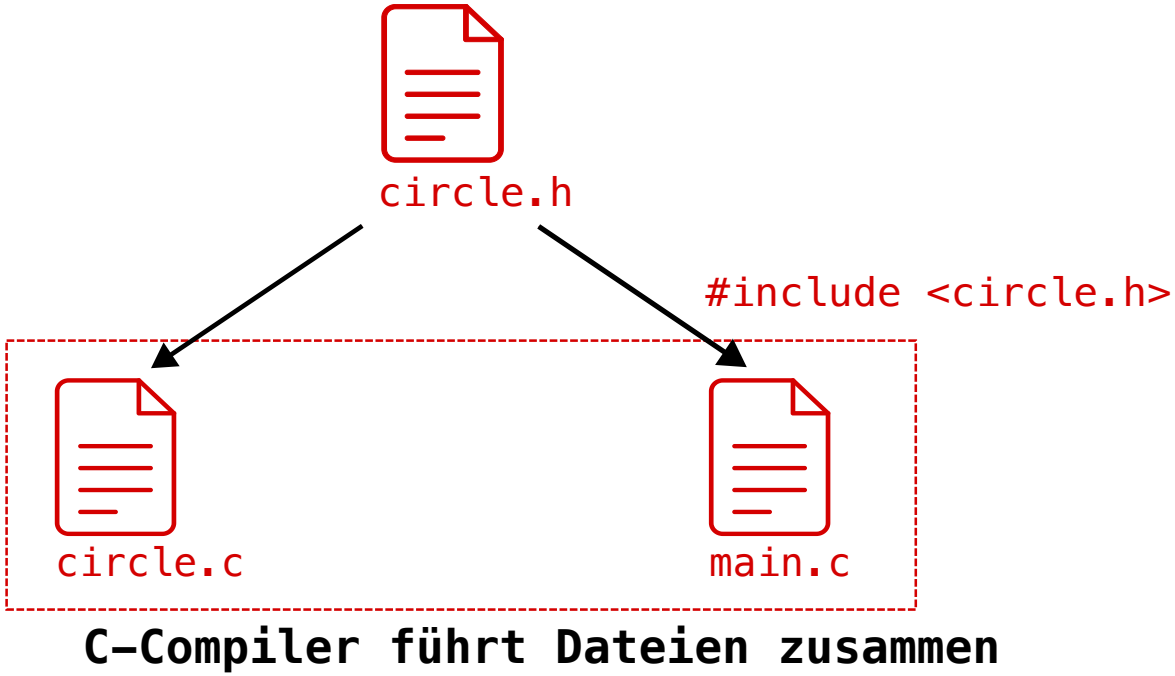
- Platzverbrauch: **Eine** Zeile pro Deklaration ⇔ **beliebig viele** pro Definition
- Vorteile einer Trennung
  1. Bessere Ordnung, Übersicht und Lesbarkeit für schnelles Nachschauen (Programmierer:innen)
  2. Beschleunigung des Übersetzens Aufteilung in Übersetzungseinheiten (*Compilation Units*) → Übersetzer muss nur Signaturen lesen
  3. Schnellere Analyse durch automatische Werkzeuge (Programmierumgebung)
- Für *kleine* Projekte ($<$ 10000 Zeilen Code) ist die Geschwindigkeit nachrangig
- Aber: Sehr wichtig bei *großen* Projekten mit mehreren Millionen Zeilen

Trennung im Quellcode - *Header*

- Üblicherweise Trennung in zwei verschiedene Dateien: **Header** und **Source**
- Beinhaltet nur das absolut Notwendige: geteilte Funktionen und Variablen
- Dateiendung: **.h** (C) bzw. **.hh/.hpp** (C++)
- Enthält die Schnittstelle der Funktionen → Signatur
- Wird zur Einbindung der Funktionen an anderen Stellen in einem Programm verwendet

Trennung im Quellcode - *Source*

- Beinhaltet alle Definitionen (Funktionen + Variablen)
- Dateiendung
  - **.c** (C)
  - **.cc/.cpp** (C++)
- Verbindung von Header- und Sourcedatei mittels **`#include`** - Direktive



Beispiel - *Header*-Dateien

- `circle.c`

```c
#include "circle.h"

void draw_circle(int diameter) {
    // viele, viele Zeilen Programmcode
}
```

- `circle.h`

```c
void draw_circle(int diameter);
```

- `main.c`

```c
#include "circle.h"

int main() {
    draw_circle(42);
```

```
    return 0;
}
```

Verwendung von *Header*-Dateien

- Einbinden von System-*Header*-Dateien

```
    #include <math.h>
    double result = log(2.0);
```

- Selbst erzeugte Header werden durch Anführungszeichen markiert

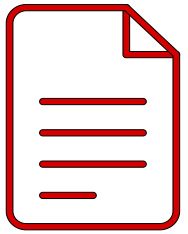
```
#include "circle.h"

int main() {
    draw_circle(42);
    return 0;
}
```

Konventionen im Umgang mit *Header*-*Source*-Dateien

- *Source*- und *Header*-Dateien …
- … müssen nicht zwingend den gleichen Namen haben
  - Dateinamen sind frei wählbar
  - Aber: Führt zu schl
- … können verschie
  Headers abbilden
  - Bsp.: `circle_with_opengl.c` oder `circle_with_x11.c`
  - Bei der Übersetzung darf nur eine Datei vorkommen
  - **Sehr** umständlicher Übersetzungsprozess
- Besser: `circle.h` ⇔ `circle.c`

Aber was ist jetzt dieses `#include` überhaupt?



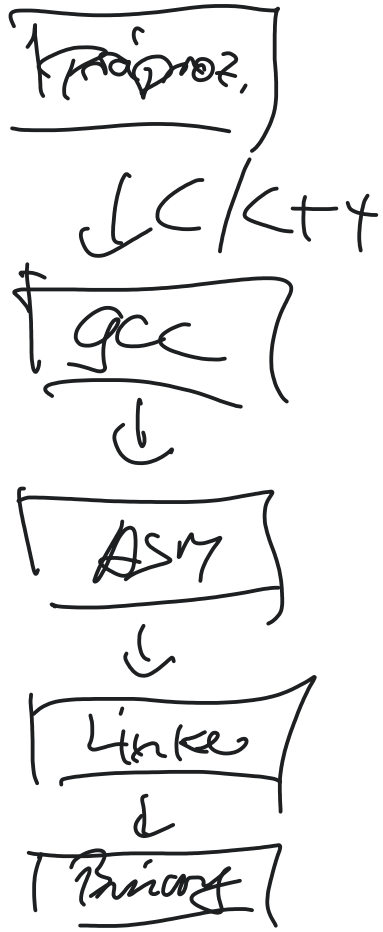
`circle.h`

`nclude <circle.h>`

`circle.c`

Einschub: C-Präprozessor

- Vorarbeitungsschritt **vor** dem Übersetzen; bearbeitet Quelltext
- Funktion: Einsetzen oder Ersetzen von Text
- Ablauf
  1. Präprozessor verarbeitet Quellcodedatei (*Source*)
  2. Wertet Direktiven aus und ersetzt sie
  3. Übergibt Ergebnis an Übersetzer
  4. *Compiler* übersetzt Quellcode in Maschineninstruktionen



Präprozessor - Befehle

- Befehle des Präprozessors fallen in vier Kategorien:
  - Einschließen (*Include*): Fügt einen Text an der angegebenen Stelle ein
  - Ersetzen (*Replace*): Ersetzt einen Text (z.B. Variable) durch einen anderen
  - Konditional (*Conditional*): Konditionales einbinden von Quellcodeabschnitten
  - Fehler/Warnung: Ausgeben einer Fehlermeldung

Präprozessor - *include*

- Bereits bekannt: Mittels **`#include`** -Direktive
- „Suche nach der angegebenen Datei und fügt deren Inhalt an der Stelle ein” ein
- <> → „Suche im Systempfad nach der Datei ”
- "" → „Suche im an der aktuellen Stelle nach der Datei ”
- Auch Dateipfade sind hier möglich

```
#include <header_name.h>              // System-Header
#include "own_header_name.h"          // Eigener Quelltext
#include "sys/defs/header-name.h"     // Im Unterverzeichnis sys/defs/
```

Präprozessor - Ersetzen

- **`#define`** -Direktive legt Namen und Wert fest
- Sucht nach dem angegebenen Namen an **allen** Stellen und ersetzt
- 🚨 Achtung: Präprozessor ist im Prinzip **reines Textersetzungswerkzeug**

```
#define varname replacement
#define SENSOR_ID_BOTTOM_LEFT 12
```

- Unterschiede zu einer Variablen
  - An beliebiger Stelle verwendbar
  - Wird vor dem Kompilieren durch den Wert ersetzt

Präprozessor - Ersetzen

- Quellcode:

```c
#include <stdio.h>
#define MAX_VOLTAGE 250
#define MAX_CURRENT 16

int main() {
    printf("Max. Power: %d", MAX_CURRENT * MAX_VOLTAGE);
    return 0;
}
```

- Bearbeitung durch Präprozessor ergibt:

```c
// ... Sehr viel eingesetzter Text durch stdio.h

int main() {
    printf("Max. Power: %d", 16 * 250);
    return 0;
}
```

(Betrachtung des Ergebnisses mit `gcc -o _main.c -E main.c`)

Präprozessor - Ersetzen

- Auch ein funktionsähnlicher Block (*macro*) kann so erzeugt werden

```
#define ADD(sum1, sum2) (sum1 + sum2)
#define SELF_SUM(x)   x+x

int main() {
    ADD(1, 3);

    int square_sum = SELF_SUM(3) * SELF_SUM(3);

    return 0;
}
```

```
int main() {
    (1 + 2); // Klammern enthalten

    int square_sum = 3+3 * 3+3;

    return 0;
}
```

Präprozessor - *Include-Guards*

- Ausgangssituation
  - Präprozessor ersetzt bloß
  - `#include` -Direktiven können **gleichen** Header **mehrfach** einfügen

    → Compiler bricht ab, da zwei Definitionen vorhanden sind
- Anforderung
  - Programmierer:innen sollen **nicht** bei **jedem** `#include` darauf achten müssen
  - Vorgehen wäre unhandlich und zeitraubend
  - Nahezu umöglich mit steigender Komplexität des Programms

Präprozessor - *Include-Guards*

- Lösung
  - *Include-Guards* als Schranke für **einmaliges** Einbinden
  - Nutzt eigenes Feature: bedingtes Einbinden
  - Verantwortung liegt bei der Programmierer:in der *Header*-Datei
- Beispiel

```
#ifndef HEADER_NAME_H // Erste Zeile des Headers
#define HEADER_NAME_H
// ... Code
#endif // Letzte Zeile des Headers
```

- `#ifndef` -Block umschließt Code vollständig

  → Kein Einfügen nach dem ersten Auftreten

Präprozessor - `#pragma`

- Alternative zu *Include-Guards*: **`#pragma once`** (Nur C++)

```
// Erste Zeile des Headers
#pragma once
// Code...
```

- Probleme mit **`#pragma`**
  - **Komplett implementierungsabhängig** → jeder *Compiler* darf die Umsetzung frei entscheiden
  - *Compiler* muss ein bestimmtes **`#pragma`** nicht implementieren
  - **`#pragma once`** wird aber von den meisten Compilern unterstützt

Präprozessor - Gegenseitige Abhängigkeiten

- Szenario: Zwei Header `a.h` und `b.h` bauen auf einander auf

```c
// a.h
#include "b.h"

void func_a() { func_b(); };
```

```c
// b.h
#include "a.h"

void func_b() { func_a() };
```

- Problem: Gegense*cies*)
  - Sind ein unerfreulic
  - Führen potentiell zu unbeschränkter, rekursiven Inklusion

Nun wieder zurück zur Trennung von Quellcode.

- Allgemein gilt: Sind ein starker Indikator für schlechte Software-Architektur

  → Gedanken über die Projektstruktur machen und restrukturieren

Grundlagen zur Aufteilung des Quellcodes

- Sichtbarkeit von Variablen: lokal vs. global
- Neue Schlüsselwörter
  - `extern`
  - `static`

→ Beginn mit lokalen Variablen

Lokale Variablen

- Bisher: Sichtbarkeit von Variablen und Funtkionen halbherzig behandelt
- Das ändert sich jetzt! Wir schauen genauer hin. 🔍
- Zunächst einmal für lokale Variablen
  - Innerhalb eines Blocks definiert
  - **Nur** im jeweiligen Block und alle untergeordneten Blöcke sichtbar

  → Bei Verlassen des Blocks werden sie zerstört

```
int sum(int a, int b) {
    int sum;
    if (a > 0){
        int c = b + b;
        return c;
    } else {
        sum = a + b;
    }
```

```
    int foo(int i) {

        int j = 2;

        return i + j;
    }
```

gkeitsbereiche

```
        return sum;
    }
```

Globale Variablen

- Deklaration außerhalb von Funktionen
- Existieren während der gesamten Lebensdauer des Programms
- Prinzipiell überall gültig und sichtbar, außer sie werden *lokal* überdeckt

C Run ▶

```c
#include <stdio.h>

int i = -10;     // Global
int j;           // Global

int main() {
    printf("i = %d\n", i);
    printf("j = %d\n", j);
    return 0;
}
```

Beispiel - Variablenüberdeckung

C Run ▶

```c
#include <stdio.h>

int i = 0;

int main() {
    printf("(1) i = %d\n", i);
    float i = 2.5;
    printf("(2) i = %.1f\n", i);
    {
        char i = 'c';
        printf("(3) i = %c\n", i);
    }
    ++i;
    printf("(4) i = %.1f\n", i);
    return 0;
```

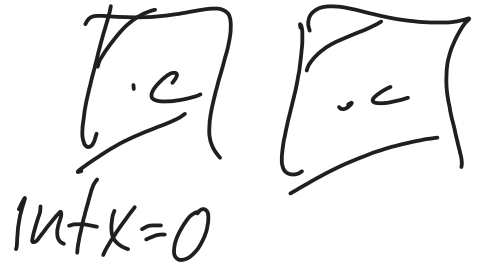
Variablenüberdeckung

- Situation: Variable mit gleichem Namen wird in untergeordnetem Block deklariert
- Variable im aktuellem Block **überdeckt** Variable aus übergeordnetem Block
- Überdeckte Variablen existieren weiterhin, aber nicht sichtbar
- Beachtet: Typ der Variable darf anders sein

Schlüsselwort `extern`

„Wie teile ich eine Variable mit zwei (oder mehr) *Source*-Dateien?"

- Lösung 1: Zweimal `int foo = 42;` hinschreiben
  - ❌ Funktioniert nicht. *Compiler* beschwert sich über „*multiple definitions*"
  - Sorgt für **zwei** `int` -Variablen mit dem Namen `foo`



- Lösung 2: **Ein**mal `int foo = 42;` hinschreiben und darauf verweisen
  - Wird funktionieren. ✅
  - Aber wie setzen wir das um? 🤔

Schlüsselwort `extern`

- Datei `a.c`
  - Enthält die Deklaration (und Initialisierung)
  - `int foo = 42;`
- Datei `b.c`
  - Enthält **nur** die Deklaration
  - `extern int foo;`
- Schlüsselwort `extern` sagt dem Compiler:

  „Hey, es gibt hier irgendwo eine `int` -Variable namens `foo`"
- Erst später im Übersetzungsprozess wird nach der Variablen gesucht

Beispiel - Schlüsselwort `extern`

```c
// main.c

int counter = 0; // Initialisierung

int main() {
    while(true) {
        increment();
        printf("%d", counter);
    }
}
```

```c
// counter.c

extern int counter;


void increment() {

    counter += 1;

}
```

Noch besser: Deklaration als `extern` in eine *Header*-Datei verschieben

```c
// counter.h
#ifndef __COUNTER_H__
#define __COUNTER_H__
extern int counter;

#endif
```

Schlüsselwort `extern`

- Bei Funktionen ist `extern` standardmäßig „aktiviert” – auch ohne explizite Nennung
- Verwendung ist optional
- Folgende Funktionen sind gleichwertig:

```c
extern void foo();

int main() {
    foo();

    return 0;
}
```

```c
void foo();

int main() {
    foo();

    return 0;
}
```

Wie behalte ich meine Variablen und Funktionen für mich?

Statische Funktionen

- Mittels `static` „bleiben" Funktionen in einer *Source*-Datei
- Sind damit **nur** in der jeweiligen *Source*-Datei sichtbar

```c
void foo();          // Deklaration als extern

int main() {
    foo();

    return 0;
}
```

```c
#include <stdio.h>



static void foo() {          // Definition/Imple
    printf("Hallo Welt!\n");
}
```

→ *Compiler* wird einen Fehler melden!

Statische Variablen

- Auch Variablen können statisch sein
- Sind damit **nur** in der jeweiligen *Source*-Datei sichtbar

```c
extern int sum;          // Deklaration als extern

int main() {
    sum = sum + 3;

    return 0;
}
```

```c
#include <stdio.h>




static int sum = 3;          // Definition
```

→ *Compiler* wird einen Fehler melden!

Spezialfall: statische lokale Variablen

- Auch lokale Variablen können statisch sein
- Existieren während der **gesamten** Programmlaufzeit
- Initialisierung nur ein einziges Mal **vor** dem Start der `main()`

C Run ▶

```c
#include <stdio.h>
 invisible int c=0;
int counter() {
    static int c = 0;
    return ++c;
}

int main() {
    for (int i = 0; i < 5; ++i) {
        printf("%d ", counter());
    }
}
```

Globalen Variablen: Auf ein Wort!

- Auf den ersten Blick sehr nützlich: Überall benutzbar und nur einmal initialisiert
- **Aber:** Je größer das Programm, desto problematischer werden sie
- Warum?
  - Variablen, insbeson
  - Weil sie global sind
  - Zustand des Progra
  - Unklarer Zusammenhang: Werteänderung ⇆ Wo wirkt sich die Änderung aus?

Vermeidet bitte globale Variablen.
Nutzt Funktionsparameter.

→ **Unnötige Komplexität und unübersichtliche**

Zusammenfassung

- C-Präprozessor
  - Manipulation des Quellcodes
  - Z. B. Einbinden von anderen *Header*-Dateien mit Deklarationen
- Lokale und globale Variablen
- Neue Schlüsselwörter
  - `extern` dient dem Teilen von Variablen zw. Dateien
  - `static` beschränkt Variablen/Funktionen auf eine Datei