Systemnahe Programmierung in C (SPiC)

11 Präprozessor

Jürgen Kleinöder, Daniel Lohmann, Volkmar Sieh

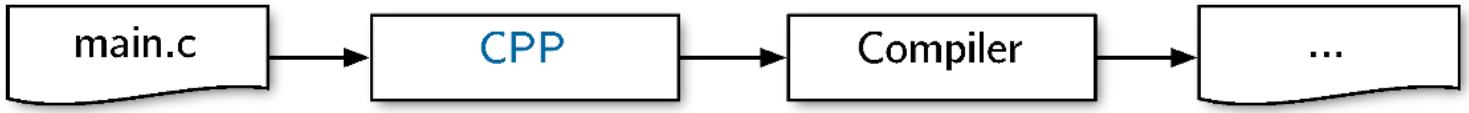
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Sommersemester 2022

http://sys.cs.fau.de/lehre/SS22/spic

Der C-Präprozessor [≠Java]



- Bevor eine C-Quelldatei übersetzt wird, wird sie zunächst durch einen Makro-Präprozessor bearbeitet
  - Historisch ein eigenständiges Programm (**CPP** = **C P**re**P**rocessor)
  - Heutzutage in die üblichen Compiler integriert
- Der CPP bearbeitet den Quellcode durch Texttransformation
  - Automatische Transformationen („Aufbereiten“ des Quelltextes)
    - Kommentare werden entfernt
    - Zeilen, die mit \ enden, werden zusammengefügt
    - ...
  - Steuerbare Transformationen (durch den Programmierer)
    - Präprozessor-Direktiven werden evaluiert und ausgeführt
    - Präprozessor-Makros werden expandiert

Präprozessor-Direktiven [≠Java]

- **Präprozessor-Direktive** := Steueranweisung an den Präprozessor

`#include <`*Datei*`>` **Inklusion:** Fügt den Inhalt von *Datei* an der aktuellen Stelle in den Token-Strom ein.

Präprozessor-Direktiven [≠Java]

- **Präprozessor-Direktive** := Steueranweisung an den Präprozessor

`#include` <*Datei*>	**Inklusion:** Fügt den Inhalt von *Datei* an der aktuellen Stelle in den Token-Strom ein.
`#define` *Makro Ersetzung*	**Makrodefinition:** Definiert ein Präprozessor-Makro *Makro*. In der Folge wird im Token-Strom jedes Auftreten des Wortes *Makro* durch *Ersetzung* substituiert. *Ersetzung* kann auch leer sein.

Präprozessor-Direktiven [≠Java]

- **Präprozessor-Direktive** := Steueranweisung an den Präprozessor

`#include` <*Datei*>	**Inklusion:** Fügt den Inhalt von *Datei* an der aktuellen Stelle in den Token-Strom ein.
`#define` *Makro Ersetzung*	**Makrodefinition:** Definiert ein Präprozessor-Makro *Makro*. In der Folge wird im Token-Strom jedes Auftreten des Wortes *Makro* durch *Ersetzung* substituiert. *Ersetzung* kann auch leer sein.
`#if` *Bedingung*, `#elif`, `#else`, `#endif`	**Bedingte Übersetzung:** Die folgenden Code-Zeilen werden in Abhängigkeit von *Bedingung* dem Compiler überreicht oder aus dem Token-Strom entfernt.
`#ifdef` *Makro*, `#ifndef` *Makro*	Bedingte Übersetzung in Abhängigkeit davon, ob *Makro* (z. B. mit `#define`) definiert wurde.

Präprozessor-Direktiven [≠Java]

- **Präprozessor-Direktive** := Steueranweisung an den Präprozessor

`#include` <*Datei*>	**Inklusion:** Fügt den Inhalt von *Datei* an der aktuellen Stelle in den Token-Strom ein.
`#define` *Makro Ersetzung*	**Makrodefinition:** Definiert ein Präprozessor-Makro *Makro*. In der Folge wird im Token-Strom jedes Auftreten des Wortes *Makro* durch *Ersetzung* substituiert. *Ersetzung* kann auch leer sein.
`#if` *Bedingung*, `#elif`, `#else`, `#endif`	**Bedingte Übersetzung:** Die folgenden Code-Zeilen werden in Abhängigkeit von *Bedingung* dem Compiler überreicht oder aus dem Token-Strom entfernt.
`#ifdef` *Makro*, `#ifndef` *Makro*	Bedingte Übersetzung in Abhängigkeit davon, ob *Makro* (z. B. mit `#define`) definiert wurde.
`#error` *Text*	**Abbruch:** Der weitere Übersetzungsvorgang wird mit der Fehlermeldung *Text* abgebrochen.

Präprozessor-Direktiven [≠Java]

- **Präprozessor-Direktive** := Steueranweisung an den Präprozessor

`#include` <*Datei*>	**Inklusion:** Fügt den Inhalt von *Datei* an der aktuellen Stelle in den Token-Strom ein.
`#define` *Makro Ersetzung*	**Makrodefinition:** Definiert ein Präprozessor-Makro *Makro*. In der Folge wird im Token-Strom jedes Auftreten des Wortes *Makro* durch *Ersetzung* substituiert. *Ersetzung* kann auch leer sein.
`#if` *Bedingung*, `#elif`, `#else`, `#endif`	**Bedingte Übersetzung:** Die folgenden Code-Zeilen werden in Abhängigkeit von *Bedingung* dem Compiler überreicht oder aus dem Token-Strom entfernt.
`#ifdef` *Makro*, `#ifndef` *Makro*	Bedingte Übersetzung in Abhängigkeit davon, ob *Makro* (z. B. mit `#define`) definiert wurde.
`#error` *Text*	**Abbruch:** Der weitere Übersetzungsvorgang wird mit der Fehlermeldung *Text* abgebrochen.

Der Präprozessor definiert letztlich eine eingebettete **Meta-Sprache**. Die Präprozessor-Direktiven (Meta-Programm) verändern das C-Programm (eigentliches Programm) vor dessen Übersetzung.

Präprozessor – Verwendungsbeispiele [≠Java]

Einfache Makro-Definitionen

Leeres Makro (Flag)	`#define USE_7SEG`
Quelltext-Konstante	`#define NUM_LEDS (4)`
„Inline“-Funktion	`#define SET_BIT(m, b) (m | (1 << b))`

Präprozessor-Anweisungen werden **nicht** mit einem Strichpunkt abgeschlossen!

Präprozessor – Verwendungsbeispiele [≠Java]

Einfache Makro-Definitionen

Leeres Makro (Flag)	`#define USE_7SEG`
Quelltext-Konstante	`#define NUM_LEDS (4)`
„Inline“-Funktion	`#define SET_BIT(m, b) (m | (1 << b))`

Präprozessor-Anweisungen werden **nicht** mit einem Strichpunkt abgeschlossen!

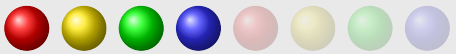
Verwendung

```
#if NUM_LEDS < 0 || 8 < NUM_LEDS
#  error invalid NUM_LEDS            // this line is not included
#endif

void enlighten(void) {
  uint8_t mask = 0, i;
  for (i = 0; i < NUM_LEDS; i++) {  // NUM_LEDS --> (4)
    mask = SET_BIT(mask, i);        // SET_BIT(mask, i) --> (mask | (1 << i))
  }
  sb_led_setMask(mask);             // -->

#ifdef USE_7SEG
  sb_show_HexNumber(mask);          // -->
#endif

}
```



Präprozessor – Gefahren [≠Java]

- Funktionsähnliche Makros sind keine Funktionen!
  - Parameter werden nicht evaluiert, sondern textuell eingefügt
    Das kann zu unangenehmen Überraschungen führen

```
#define POW2(a) 1 << a
n = POW2(2) * 3
```

<< hat geringere Präzedenz als *

```
⇝ n = 1 << 2 * 3
```

Präprozessor – Gefahren [≠Java]

- Funktionsähnliche Makros sind keine Funktionen!
  - Parameter werden nicht evaluiert, sondern textuell eingefügt
    Das kann zu unangenehmen Überraschungen führen

```
#define POW2(a) 1 << a
n = POW2(2) * 3
```

<< hat geringere Präzedenz als *

```
⇝ n = 1 << 2 * 3
```

  - Einige Probleme lassen sich durch korrekte Klammerung vermeiden

```
#define POW2(a) (1 << a)
n = POW2(2) * 3
```

```
⇝ n = (1 << 2) * 3
```


Präprozessor – Gefahren [≠Java]

- Funktionsähnliche Makros sind keine Funktionen!
  - Parameter werden nicht evaluiert, sondern textuell eingefügt
    Das kann zu unangenehmen Überraschungen führen

```
#define POW2(a) 1 << a
n = POW2(2) * 3
```

<< hat geringere Präzedenz als *

⇝ `n = 1 << 2 * 3`

  - Einige Probleme lassen sich durch korrekte Klammerung vermeiden

```
#define POW2(a) (1 << a)
n = POW2(2) * 3
```

⇝ `n = (1 << 2) * 3`

  - Aber nicht alle

```
#define max(a, b) ((a > b) ? a : b)
n = max(x++, 7)
```

a++ wird ggf. zweimal ausgewertet

⇝ `n = ((x++ > 7) ? x++ : 7)`

Präprozessor – Gefahren [≠Java]

- Funktionsähnliche Makros sind keine Funktionen!
  - Parameter werden nicht evaluiert, sondern textuell eingefügt
    Das kann zu unangenehmen Überraschungen führen

```
#define POW2(a) 1 << a
n = POW2(2) * 3
```
<< hat geringere Präzedenz als *

⇝
```
n = 1 << 2 * 3
```

  - Einige Probleme lassen sich durch korrekte Klammerung vermeiden

```
#define POW2(a) (1 << a)
n = POW2(2) * 3
```
⇝
```
n = (1 << 2) * 3
```

  - Aber nicht alle

```
#define max(a, b) ((a > b) ? a : b)
n = max(x++, 7)
```
a++ wird ggf. zweimal ausgewertet

⇝
```
n = ((x++ > 7) ? x++ : 7)
```

- Eine mögliche Alternative sind `inline`-Funktionen C99
  - Funktionscode wird eingebettet ⇝ ebenso effizient wie Makros

```
inline int max(int a, int b) {
  return (a > b) ? a : b;
}
```