System-Level Programming

11 Preprocessor

J. Kleinöder, D. Lohmann, V. Sieh, P. Wägemann

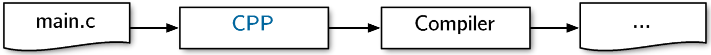
Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Summer Term 2025

http://sys.cs.fau.de/lehre/ss25

The C Preprocessor [≠Python]



- Before a C source file is compiled, it is processed by the macro preprocessor
  - in the past, a stand-alone program (**CPP** = **C** **P**re**P**rocessor)
  - nowadays, integrated into compilers
- The CPP edits the source code by text transformations
  - automatic transformation ("clean-up" of the source code)
    - comments are deleted
    - lines ending with \ are put together
    - ...
  - controllable transformations (by the programmer)
    - preprocessor directives are evaluated and executed
    - preprocessor macros are expanded

Preprocessor Directives [≠Python]

- **preprocessor directive** := control expression for the preprocessor

`#include` <*file*>	**Inclusion:** The contents of *file* are included at this exact place into the token stream.

Preprocessor Directives [≠Python]

- **preprocessor directive** := control expression for the preprocessor

`#include` <*file*>	**Inclusion:** The contents of *file* are included at this exact place into the token stream.
`#define` *macro replacement*	**Definition of macros:** Defines a preprocessor macro *macro*. In the following token stream, each occurrence of *macro* will be replaced by *replacement*. The *replacement* can also be empty.

Preprocessor Directives

- **preprocessor directive** := control expression for the preprocessor

`#include` *<file>*	**Inclusion:** The contents of *file* are included at this exact place into the token stream.
`#define` *macro replacement*	**Definition of macros:** Defines a preprocessor macro *macro*. In the following token stream, each occurrence of *macro* will be replaced by *replacement*. The *replacement* can also be empty.
`#if` *condition*, `#elif`, `#else`, `#endif`	**Conditional compilation:** Following lines of code are handed to the compiler or are deleted from the token stream dependent on *condition*.
`#ifdef` *macro*, `#ifndef` *macro*	Conditional compilation dependent on (defined/not defined) *macro* (e. g., with `#define`).

Preprocessor Directives [≠Python]

- **preprocessor directive** := control expression for the preprocessor

`#include` <*file*>	**Inclusion:** The contents of *file* are included at this exact place into the token stream.
`#define` *macro replacement*	**Definition of macros:** Defines a preprocessor macro *macro*. In the following token stream, each occurrence of *macro* will be replaced by *replacement*. The *replacement* can also be empty.
`#if` *condition*, `#elif`, `#else`, `#endif`	**Conditional compilation:** Following lines of code are handed to the compiler or are deleted from the token stream dependent on *condition*.
`#ifdef` *macro*, `#ifndef` *macro*	Conditional compilation dependent on (defined/not defined) *macro* (e. g., with `#define`).
`#error` *text*	**Abort:** The compilation procedure is aborted with the error message *text*.

Preprocessor Directives [≠Python]

- **preprocessor directive** := control expression for the preprocessor

`#include` <*file*>	**Inclusion:** The contents of *file* are included at this exact place into the token stream.
`#define` *macro replacement*	**Definition of macros:** Defines a preprocessor macro *macro*. In the following token stream, each occurrence of *macro* will be replaced by *replacement*. The *replacement* can also be empty.
`#if` *condition*, `#elif`, `#else`, `#endif`	**Conditional compilation:** Following lines of code are handed to the compiler or are deleted from the token stream dependent on *condition*.
`#ifdef` *macro*, `#ifndef` *macro*	Conditional compilation dependent on (defined/`n`ot defined) *macro* (e. g., with `#define`).
`#error` *text*	**Abort:** The compilation procedure is aborted with the error message *text*.

The preprocessor defines an embedded **meta language**. All preprocessor directives (i.e., the meta program) modify the C program (i.e., actual program) prior to actual compilation.

Preprocessor – Example

- Simple macro definitions

empty macro (flag)	`#define USE_7SEG`
source-code constant	`#define NUM_LEDS (4)`
“inline” function	`#define SET_BIT(m, b) (m | (1 << b))`

Preprocessor directives are **not** followed by a semicolon!

Preprocessor – Example [≠Python]

- Simple macro definitions

empty macro (flag)	`#define USE_7SEG`
source-code constant	`#define NUM_LEDS (4)`
“inline” function	`#define SET_BIT(m, b) (m | (1 << b))`

Preprocessor directives are **not** followed by a semicolon!

- Usage

```
#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);          // --> 0F
#endif

}
```

Preprocessor – Risks

- Function-like macros are indeed no functions!
  - Parameters are not evaluated, rather they are expanded textually. Since CPP misses C semantics, expansions can lead to unwanted surprises.

```
#define POW2(a) 1 << a                      << has lower precedence than *
n = POW2(2) * 3                          ⤳ n = 1 << 2 * 3
```

Preprocessor – Risks

[≠Python]

- Function-like macros are indeed no functions!
  - Parameters are not evaluated, rather they are expanded textually. Since CPP misses C semantics, expansions can lead to unwanted surprises.

```
#define POW2(a) 1 << a                    << has lower precedence than *
n = POW2(2) * 3                         ⤳ n = 1 << 2 * 3
```

  - Some problems can be avoided by the correct use of brackets

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

Preprocessor – Risks [≠Python]

- Function-like macros are indeed no functions!
  - Parameters are not evaluated, rather they are expanded textually. Since CPP misses C semantics, expansions can lead to unwanted surprises.

```
#define POW2(a) 1 << a                       << has lower precedence than *
n = POW2(2) * 3                           ⤳ n = 1 << 2 * 3
```

  - Some problems can be avoided by the correct use of brackets

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

  - However, not all

```
#define max(a, b) ((a > b) ? a : b)      a++ will be potentially evaluated twice
n = max(x++, 7)                           ⤳ n = ((x++ > 7) ? x++ : 7)
```

Preprocessor – Risks [≠Python]

- Function-like macros are indeed no functions!
  - Parameters are not evaluated, rather they are expanded textually. Since CPP misses C semantics, expansions can lead to unwanted surprises.

```
#define POW2(a) 1 << a                    << has lower precedence than *
n = POW2(2) * 3                        ⇝ n = 1 << 2 * 3
```

  - Some problems can be avoided by the correct use of brackets

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

  - However, not all

```
#define max(a, b) ((a > b) ? a : b)       a++ will be potentially evaluated twice
n = max(x++, 7)                        ⇝ n = ((x++ > 7) ? x++ : 7)
```

- A possible alternative are real `inline` functions C99
  - function's body is directly inserted ⇝ as efficient as macros

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