Der C-Präprozessor

Lukas Stabe

Universität Hamburg

Proseminar „C - Grundlagen und Konzepte“, 2013

Inhalt

Einführung

Was ist der C-Präprozessor?

- Früher eigenständiges Programm, heute Teil von Compilern
- Vor dem eigentlichen Übersetzen
- Ganz grob: Suchen und Ersetzen
- Kann man auch Quatsch mit machen:

```
%:include <stdio.h>

int ma\
in(int argc, char** argv) ??<
printf("He\
llo world!??/n");
%>
// Ausgabe: "Hello world!\n"
```

Inhaltslose Makros

- Definiert via `#define MAKRONAME`
- Entfernt via `#undef MAKRONAME`
- Bedingtes Übersetzen mit `#ifdef`, `#ifndef`, `#else` und `#endif`
- Oft auch von der Toolchain vordefiniert

```
#define VERBOSE

#ifdef VERBOSE
  #ifdef WIN32
    printf("I was compiled on Windows :(\n");
  #else
    printf("Not compiled on Windows :)\n");
  #endif
#endif
```

Parameterlose Makros

- Definiert via `#define MAKRONAME WERT`
- Entfernt via `#undef MAKRONAME`
- Jedes Vorkommen (ausser in String-Literals) von `MAKRONAME` wird durch `WERT` ersetzt
- Weitere Conditionals: `#if` und `#elif`

```
#define L printf
#define PI 3.141
#define VERBOSE 3

#if VERBOSE >= 2
  L("debug text %f\n", PI);
#endif
```

Parameterlose Makros (cont.)

Vorsicht: Makros werden wirklich *nur* ersetzt.

Falsch:

```c
#define L 1 + 1
int x = L * 2;
// x == 3
```

Richtig:

```c
#define L ( 1 + 1 )
int x = L * 2;
// x == 4
```

Parameterbehaftete Makros

- Sehen aus wie Funktionen, verhalten sich aber ganz anders
- Definiert wie folgt:

```
#define MAKRO_1(ARG1, ARG2) foo(ARG1, ARG2)
#define MAKRO_2(ARG1, ...)  ARG1(__VA_ARGS__)
#define MAKRO_3(...)        MAKRO_2(printf, __VA_ARGS__)
```

- Variable Anzahl von Argumenten mit „. . .“, benutzbar als `__VA_ARGS__`
  - Lt. Standard ein oder mehr Argumente, GCC und clang erlauben auch kein Argument
- Es wird so lange ersetzt, bis nichts mehr zu ersetzen ist

Parameterbehaftete Makros – beliebte Fehler

```c
#define TIMES_FIVE(x) ( x * 5 )
int a = TIMES_FIVE( 1 + 2 );
// => int a = ( 1 + 2 * 5 );
// => a != 15, a == 11
```

```c
#define SQUARE(x) ( ( x ) * ( x ) )
int a = 2;
int b = SQUARE(++a);
// => int b = ((++a) * (++a));
// => b != 9, b == 12
```

```c
#define SOMETHING(x) foo(x); bar(x)
if (baz != null) SOMETHING(baz);
// besser:
// #define SOMETHING(x) do { foo(x); bar(x); } while (0)
```

Makros – praktisches Beispiel

main.c

```c
#ifdef DEBUG
  #define LOG_FMT "%s#%d %s: "
  #define LOG_ARGS __FILE__, __LINE__, __func__
  #define LOG(format, ...) \
    printf(LOG_FMT format "\n", LOG_ARGS, ##__VA_ARGS__)
#else
  #define LOG(format, ...)
#endif

void some_function() {
    LOG("Hello world");
    // => main.c#11 some_function: Hello world
    LOG("Nummer %d lebt!", 5);
    // => main.c#13 some_function: Nummer 5 lebt!
}
```

Makros – praktisches Beispiel 2

main.c (cont.)

```
16 // LOG(format, ...) wie zuvor
17 #define VLOG(type, var) LOG("%s = %" #type, #var, var)
18
19 // Benutzung:
20 void other_function() {
21     int foo = 42;
22     char *string = "bar";
23     VLOG(d, foo);
24     // => file.c#23 other_function: foo = 42
25     VLOG(s, string);
26     // => file.c#25 other_function: string = bar
27 }
```

Makros – Zusammenfassung

- Makros werden per `#define` definiert, darauffolgende Vorkommen des Namens werden durch den Wert ersetzt
- Parameterlose und parameterbehaftete Makros
  - Spezielle Makros `__FILE__` und `__LINE__`
- Viele Möglichkeiten, Fehler zu machen
- Ermöglichen coole Sachen
- Können auch in Kommandozeilenparametern der Compilers definiert werden

Includes

- `#include` Wird durch Inhalt referenzierter Datei ersetzt
- `#include <datei>` durchsucht nur Systempfade,
  `#include "datei"` zusätzlich zuerst das aktuelle Verzeichnis
- Includes erlauben Modularität ohne Redundanz

Includes – Include guards

Problem: Mehrfaches includieren der selben Datei führt zu mehrfach definierten Symbolen:

grandparent.h

```
typedef int something;
```

parent.h

```
#include "grandparent.h"
```

child.h

```
#include "parent.h"
#include "grandparent.h"
// error: redefinition of typedef 'something'
```

Includes – Include guards (cont.)

Lösung: *include guards*:

grandparent.h

```
#ifndef _GRANDPARENT_H_
  #define _GRANDPARENT_H_
  typedef int something;
#endif
```

parent.h

```
#ifndef _PARENT_H_
  #define _PARENT_H_
  #include "grandparent.h"
#endif
```

child.h

```
#ifndef _CHILD_H_
  #define _CHILD_H_
  #include "parent.h"
  #include "grandparent.h"
#endif
```

Errors und Warnings

- Direktiven zum ausgeben von Compiler-Warnungen und Fehlern
- `#error TEXT` und `#warn TEXT`

```
#include <some_library/version.h>

#if __some_library_version < 4
  #error Must have some_library version 4 or newer
#endif

#if __some_library_version == 5
  #warning Bad performance with some_library version 5
#endif
```

Pragmas

- `#pragma ...` oder `_Pragma ("...")`
- Viele viele Anwendungen, wenige sinvoll
  - "Note that in general we do not recommend the use of pragmas"[1]
  - "It was basically a mistake to use `#pragma` for anything."[2]
- Nützliche Anwendung: diagnostics

```
#ifdef DEBUG
  #pragma GCC diagnostic warning "-Wall"
#else
```

[1] http://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/Pragmas.html
[2] http://gcc.gnu.org/onlinedocs/gcc-4.8.0/gcc/Function-Attributes.html

Pragmas (cont.)

- #pragma GCC diagnostic warning "-Wsomething"
- #pragma GCC diagnostic error "-Wsomething"
- #pragma GCC diagnostic ignored "-Wsomething"
- #pragma GCC diagnostic push
- #pragma GCC diagnostic pop

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused"
void some_function() {
    int unused_var = 5;
    do_something();
}
#pragma GCC diagnostic pop
```

Diverses

Was tut der Präprozessor sonst noch?

- Kommentare durch Whitespace ersetzen
- Tokenization
- Zusammenführen von Zeilen mit \ am Ende
- Ersetzen von Di- und Trigraphen

```
%:include <stdio.h>

int ma\
in(int argc, char** argv) ??<
printf("He\
llo world!??/n");
%>
// Ausgabe: "Hello world!\n"
```

Quellen

- http://en.wikipedia.org/wiki/C_preprocessor
- http://de.wikipedia.org/wiki/C-Präprozessor
- http://gcc.gnu.org/onlinedocs/gcc/

```
printf("Goodbye, world");
```