Der C-Präprozessor

Lukas Stabe

Inhaltsverzeichnis

1 Einleitung

Der C-Präprozessor übernimmt eine wichtige Rolle im Vorgang des Übersetzens von C-Code in Maschinensprache. Er erlaubt das Einbinden von Quelltextdateien in andere, bietet Makro-Funktionen, und kann zum bedingten Übersetzen von Codestücken genutzt werden.

Sehr abstrakt gesehen sucht und ersetzt der Präprozessor bestimmte Zeichenketten im Quellcode.

Während er früher als eigenständiges Programm genutzt wurde, wird er heutzutage meist mit Compilern mitgeliefert, und von diesen als Teil des Übersetzungsvorganges aufgerufen. Trotzdem kann der Präprozessor immer noch eigenständig genutzt werden, und damit auch für andere Texte als C-Quellcode verwendet werden. Benutzt man z.B. den GCC-Compiler, kann das Übersetzen mit der Option -E nach der Präprozessor-Phase abgebrochen werden.

2 Makros

Makros sind zugleich das Komplexeste, als auch das nützlichste Feature des Präprozessors. Es gibt zwei Arten von Makros: parameterlose und parameterbehaftete.

2.1 Parameterlose

Parameterlos Makros sind die einfachere der beiden Formen von Makros. Einmal definiert wird jedes Vorkommen des Namens durch den Inhalt ersetzt. Auch leerer Inhalt ist erlaubt.

```
#define MAKRO "hallo welt"
#define LEER
printf(MAKRO); LEER
```

Wird vom Präprozessor verarbeitet zu:

```
printf("hallo welt");
```

Es ist möglich anhand des Inhalts eines Makros, oder daran ob es überhaupt definiert ist, zu entscheiden ob teile des Quelltextes in die Ausgabe übernommen werden sollen oder nicht.

Dies ist besonders nützlich, da die Toolchain oft Makros mit Systemeigenschaften vordefiniert. Ebenso können Makros dem Compiler als Kommandozeilenoptionen übergeben werden.

```
#define VERBOSE = 3
#if VERBOSE >= 2
  #ifdef WIN32
    printf("I was compiled on Windows :(\n");
  #else
    printf("Not compiled on Windows :)\n");
  #endif
#endif
```

Obiges führt auf nicht-Windows PCs zu folgender Ausgabe:

```
    printf("Not compiled on Windows :)\n");
```

Makros können auch wieder aus der Liste der verwendeten Makros entfernt werden. Dies geschieht so:

```
#define FOO bar
FOO
#undef FOO
FOO
```

Nach dem Preprocessing wird aus diesem Code folgendes:

```
bar
FOO
```

2.2 Parameterbehaftete

Parameterbehaftete Makros sind die komplexere Art der Makros. Auf den ersten Blick sehen sie Funktionen recht ähnlich, sollten aber keinesfalls mit diesen verwechselt werden, da sie sich sehr unterschiedlich verhalten.

Ebenso wie parameterlose werden parameterbehaftete Makros definiert, und jedes darauf folgende Vorkommen wird ersetzt. Von ihrer einfacheren Alternative unterscheiden diese sich jedoch dadurch, dass sie eine Liste an Parametern übergeben bekommen, die in den substituierten Text eingesetzt werden.

Mit `...` kann eine Argumentliste von variabler länge (lt. Standard ein oder mehr Argumente, gcc und clang unterstützen auch kein Argument) angegeben werden, die im Substituttext mit `__VA_ARGS__` referenziert wird.

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

2.3 Verbreitete Fehler

Da Makros Variablen und Funktionen sehr ähnlich sehen, sich aber sehr anders verhalten, gibt es eine reihe beliebter Fehler, die man besser vermeiden sollte. Jedem dieser Fehler liegt zu Grunde, dass der Präprozessor nichts weiter tut, als den Makronamen durch seinen Inhalt zu ersetzen.

2.3.1 Operatorpäzedenz

```
#define FIVE_PLUS_FIVE 5 + 5
int a = FIVE_PLUS_FIVE * 2
```

In diesem Beispiel hat `a` nach Preprocessing, Kompilieren und Ausführung nicht etwa den Wert 20, so wie man es bei einer Variable erwarten würde, sondern 15. Dies liegt daran, dass zuerst `FIVE_PLUS_FIVE * 2` zu `5 + 5 * 2` wird, was dank Operatorpräzedenz zu 15 evaluiert.

Dieses Problem lässt sich lösen, indem man bei der Definition eines solchen Makros Klammern um den Wert setzt. Analog sollten auch um Parameter in Parameterbehafteten Makros immer Klammern gesetzt werden.

2.3.2 Mehrfaches Einfügen eines Ausdrucks

```
#define SQUARE(x) ( ( x ) * ( x ) )
int a = 2;
int b = SQUARE(++a);
```

Hier ist das Problem, dass in dem Makro `SQUARE(x)` zwar die Parameter ordentlich umklammert sind, allerdings taucht ein und der selbe Parameter mehrfach im Substituttext auf. Dies führt dazu, dass der Ausdruck der als Parameter übergeben wird mehrfach evaluiert wird. Funktionsaufrufe fänden also mehrfach statt, oder es wird, wie in diesem Beispiel, a zwei- anstatt nur einmal inkrementiert.

2.3.3 Kontrollstrukturen ohne Block-Klammern

```
#define SOMETHING(x) foo(x); bar(x)
if (baz != null) SOMETHING(baz);
```

Bei diesem Beispiel liegt der Fehler darin, dass `if (...)` ohne folgende Blockklammern nur die erste Anweisung hinter dem `if` selbst ausführt. Es wird also `foo(baz)` nur ausgeführt wenn `baz != null`, die Bedingung also erfüllt ist. `bar(baz)` wird hingegen in jedem Fall ausgeführt, was im schlimmsten Fall zu Abstürzen führen kann.

Um solche Fehler zu vermeiden sollte Makros, die mehr als eine Anweisung enthalten wie folgt definiert werden:

```
#define SOMETHING(x) do { foo(x); bar(x); } while (0)
```

3 Includes

Der Präprozessor ermöglicht es einem mit der `#include`-Direktive, den Inhalt einer Datei in eine andere einzuschliessen. Dabei wird die Direktive durch den Inhalt der angegebenen Datei ersetzt. Es gibt zwei verschiedene Schreibweisen, die leicht unterschiedlich wirken: `#include <datei>` durchsucht nur Sy-

stempfade, während `#include "datei"` vor den Systempfaden noch im aktuellen Verzeichnis nach Dateien sucht.

3.1 Include Guards

Da der Inhalt der includierten Datei einfach in die aktuelle Datei kopiert wird, ergeben sich Probleme wenn eine Datei mehrfach, meist über andere Dateien, eingebunden wird. Zum Beispiel taucht dann die selbe Typdefinition zwei mal in der Datei auf, was einen Fehler beim Übersetzen hervorruft.

grandparent.h

```c
typedef int something;
```

parent.h

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

child.h

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

Die Lösung für dieses Problem baut auf Makros auf, und nennt sich Include guards. In jeder Header-Datei wird am Anfang überprüft, ob ein bestimmtes Makro bereits definiert ist. Ist das nicht der Fall, wird das Makro definiert, und der Inhalt der Datei wiedergegeben.

grandparent.h

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

parent.h

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

child.h

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

4 Errors und Warnings

Der Präprozessor bietet auch Direktiven, die dazu führen, dass während des Übersetzungsvorganges Compiler-warnings und -errors ausgegeben werden.

Während dies an sich noch nicht all zu viel Nutzen bringt, ist es in Kombination mit Makros, speziell Makros die vom System oder verwendeten Libraries definiert wurden, sehr nützlich, in bestimmten Fällen das Kompilieren zu stoppen.

```
#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
```

5 Pragmas

`#pragma` (auch: `_Pragma ("...")`) ist eine Direktive, hinter der sehr viel Funktionalität versteckt ist. Viel dieser Funktionalität ist allerdings über andere Wege besser zu erreichen, weswegen die Nutzung von Pragmas selten empfohlen wird.

So schreibt beispielsweise die GCC-Compiler-Dokumentation:

> "It was basically a mistake to use #pragma for anything."[1]

Wenigstens eine sinnvolle Anwendung gibt es dennoch, und zwar die Kontrolle der Warnungen und Fehlermedungen die vom Compiler ausgegeben werden. Möglich wird das mit den folgenden Anweisungen:

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

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

So ist es beispielsweise möglich, in einem Projekt in dem normalerweise die Option `-Werror` zum Einsatz kommt während der Entwicklung, für Debug-Builds also, wieder auf normale Warnings umzusteigen, indem man die folgende Direktive benutzt:

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

Weiterhin ist es möglich, einzelne Warnungen zu unterdrücken. Ist man sich zum Beispiel bei einer bestimmten Funktion im klaren darüber, dass eine lokale Variable ungenutzt bleibt, möchte diese aber aus irgendeinem Grund nicht herausnehmen, kann man mit folgendem Code die Warnung deaktivieren:

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

6 Diverses

Neben den bisher vorgestellten Funktionalitäten übernimmt der Präprozessor noch weitere Aufgaben. So ersetzt er Di- und Trigraphen durch die entsprechenden Symbole. Di- und Trigraphen sind Symbolketten, die verwendet werden können, wenn die eigene Tastatur ein bestimmtes Zeichen nicht beinhaltet. `??/` steht beispielsweise für einen Backslash, `%>` für eine schliessende geschweifte Klammer.

Mit heutigen Tastaturen ist diese Funktion kaum mehr von Nutzen, so dass GCC zum Beispiel Trigraphen nur noch mit einer speziellen Kommandozeilenoption unterstützt.

Weiterhin hängt der Präprozessor an jede Zeile die mit einem Backslash endet die darauffolgende Zeile an, ersetzt Kommentare durch Whitespace und bricht dann noch den verarbeiteten Quellcode in für den Compiler einfacher zu übersetzende Tokens auf.

Diese weniger bekannten (und weniger nützlichen) Funktionen führen dazu, dass das folgende Beispiel komplett gültiger C-Code ist.

```
%:include <stdio.h>

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