Proseminar

C-Programmierung
Grundlagen und Konzepte

Der Präprozessor

von:
Svenja Neef

Inhaltsverzeichnis

1 Der C-Präprozessor

1.1 Was ist der C-Präprozessor

Grundlegend ist ein Präprozessor ein Programm, das vorhandenen Code umschreibt, um ihn für die Maschinelle Verarbeitung vorzubereiten.

Der C-Präprozessor kann hierfür eine Reihe von Befehlen befolgen. Anhand dieser Befehle führt er jedoch lediglich Textersetzungen vor; es gibt keinerlei Überprüfungen, ob nach den Ersetzungen und Einfügungen valider und sinnvoller Quelltext entsteht.

1.2 Präprozessor-Befehle

1.2.1 Zusammenführen von Zeilen

Eine Präprozessor-Anweisung endet normalerweise am Ende der Zeile. Um die Lesbarkeit zu erhöhen kann es jedoch Wünschenwert sein eine Anweisung über mehrere Zeilen zu verteilen.

Um dies zu tun

1.2.2 Trigraph-Zeichen

Als C zuerst entwickelt wurde konnte es vorkommen, das einige Sonderzeichen, die C benutzt, vom Entwickler nicht direkt eingegeben werden konnten. Dies konnte zum Beispiel vorkommen wenn ein Zeichensatz benutzt benutzt wurde, der das entsprechende Zeichen nicht enthielt, oder ein Zeichen auf der Tastatur nicht zur Verfügung stand.

Um dieses Problem zu Lösen wurden Trigraph-Zeichen genutzt, die aus 3 Zeichen bestanden, die überall zur Verfügung standen. Die Zeichen werden durch „??“ eingeleitet. Abhängig vom darauf folgenden Zeichen ersetzt der Präprozessor dann den Trigraph durch das eigentlich benötigte Zeichen.

In C stehen hierbei folgende Zeichen zur Verfügung:

```
Trigraph   Zeichen
------------------
??=          #
??(          [
??)          ]
??/          \
??'          ^
??!          |
??<          {
??>          }
??-          ~
```

1.2.3 Einfügungen: #include

Der Präprozessor kann den Inhalt aus anderen Dateien in den Quellcode einfügen. Hierzu dient der Befehl #include. Die Synatax hierzu ist:

```
#include <dateiname>
```

beziehungsweise

`#include „dateiname“.`

Der Unterschied besteht darin, das bei Benutzung der spitzen Klammern die darin benannte Datei nur im include-Verzeichnis der C-Umgebung gesucht wird, bei Benutzung Anführungszeichen wird zusätzlich das aktuelle Verzeichnis im Dateisystem durchsucht.

1.2.4 Makros: #define

Mit dem #define-befehl werden im Präprozessor Makros erstellt. Diesen können auch Parameter übergeben werden.

Die Syntax hierfür ist folgende:

`#define` **`NAME`** `(`*`para1`*`,` *`para2`*`) Ersetzung`

Der Name des Makros (**NAME**) wird hierbei gemäß Konvention in Großbuchstaben geschrieben. Der Präprozessor ersetzt jedes Vorkommen von NAME durch die dahinterstehende Ersetzung. (Aus „hier steht eine NAME“ wird also „Hier steht eine Ersetzung“) Der Geltungsbereich dieses Makros beginnt hierbei erst nach der Anweisung. Die Ersetzung kann hierbei beliebig lang sein, und auch weitere Makros enthalten (siehe auch: 1.4.2 Verschachtelte Makros).

Übergebene Parameter (z.B. para1) können in der Ersetzung eingefügt werden indem der Name des Parameters aufgerufen wird. Anstatt die Anzahl der übergebenen Parameter festzulegen, kann man auch beliebig viele optionale Parameter festlegen, indem man „...“ als Parameternamen angibt. Auf diese kann dann mit dem Makro __VA_ARGS__ Zugegriffen werden.

Hiermit können zum Beispiel eigene Methoden für Fehlermeldungen erstellt werden:

Durch die Definition folgenden Makros:

```
#define errprintf(...) fprintf(stderr, __VA_ARGS__)
```

kann mit

```
errprintf("Hallo Welt %d %s\n", val, str);
```

eine Fehlermeldung ausgegeben werden. Soll die Fehlermeldung später in anderer Art ausgegeben werden, muss nur das Makro geändert werden.

1.2.4.1 Einige Vordefinierte Makros

C bringt bereits eine Reihe vordefinierter Makros mit. Standardmäßig sind hierbei die folgenden Makros verfügbar.

__LINE__ : Gibt die aktuelle Zeilennummer aus.
__FILE__ : Gibt den Namen der aktuellen Datei aus.
__DATE__ : Gibt das Datum zum Übersetzungszeitpunkt aus.
__TIME__ : Gibt die Zeit zum Übersetzungszeitpunkt aus.
__STDC__ : ist auf 1 Gesetz, wenn die Übersetzungsumgebung dem ANSI-C-Standard entspricht.
__STDC_VERSION__ : gibt ab C99 die C-Version des Übersetzers an.

1.2.5 Makros: #undef

Der #undef-Befehl markiert das Ende des Geltungsbereiches eines Makros.

Die Syntax hierzu ist:

```
#undef NAME
```

Am dem Punkt an dem diese Anweisung steht wird der Präprozessor kein Vorkommen von NAME mehr ersetzen, außer es wird eine neue #define-Anweisung gegeben.

1.2.6 Bedingte Anweisungen: `#if`, `#ifdef`, `#ifndef`

Auch der C-Präprozessor beherrscht bedingte Anweisungen mit if-konstrukten. Diese Können mit #if, #ifdef oder #ifndef eingeleitet werden.

Hinter einem #if stehen hierbei die Vergleichsoperationen zur Verfügung, die auch aus C bekannt sind („==“, „<=“, „>=“, „<“, „>“) ebenso zur Verfügung wie die Verneinung mit „!“.

Hinzu kommt die Abfrage „defined“, welche überprüft ob das danach angegebene Makro an dieser Stelle definiert ist.

1.2.7 Bedingte Anweisungen: `#else`, `#elif`, `#endif`

Um den Bereich einer bedingten Anweisung zu beenden stehen #else, #elif und #endif zur Verfügung. #else entspricht hierbei dem bekannten „else“ aus C, #elif steht für „else if“, und #endif markiert das ende des aktuellen Blockes.

1.2.8 #line

Mit dem Befehl #line kann man die Zeilennummer und den Dateinamen beeinflussen, die bei Fehlermeldungen oder über die entsprechenden Makros ausgegeben werden. Die Syntax hierbei lautet:

```
#line zeilennummer „dateiname“
```

1.2.9 #error

Mit der #error-Anweisung kann dem Kompilierungsvorgang unterbrochen und dem Benutzer eine Meldung ausgegeben werden. Die Syntax hierzu lautet:

```
#error fehlermeldung
```

In der Fehlermeldung kann auch auf Makros Zugegriffen werden, z.B.:

```
#error Fehler trat auf in __FILE__ in Zeile __LINE__ .
```

1.2.10 #pragma

Pragma-Anweisungen sind Anweisungen für den Compiler, und unterscheiden sich damit von Compiler zu Compiler. Meist werden sie benutzt im Code, der für einen Compiler geschrieben wurde mit einem anderen Übersetzen zu können.

Der GNU GCC empfiehlt inzwischen #pragma-Anweisungen zu vermeiden, und verweist auf Funktionsattribute (http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html#Function-Attributes).

1.3 Übersetzungsphasen

Die Übersetzung von C-Quellcode erfolgt gemäß dem C-Standard in insgesamt 8 Phasen, von denen 4 durch den Präprozessor durchgeführt werden.

1. Ersetzung von Trigraph-Zeichen (siehe 1.2.2 Trigraph-Zeichen)

2. Zusammenführen von durch \\ markierten Zeilen (siehe 1.2.1 Zusammenführen von Zeilen)
3. Aufteilung in Tokens
   Hierbei werden Kommentare durch Leerzeichen ersetzt, und der gesamte Quelltext wird in kleinere Einheiten zerlegt, die für die folgenden Compilerphasen besser zu verarbeiten sind
4. Ersetzen von Makros und einschleusen von Quelltext
   Zuletzt werden #include, #if und #define-Anweisungen ausgewertet.

1.4 Anwendungsbeispiele

1.4.1 Dateien nur ein mal einfügen

Da der Präprozessor keine Überprüfungen durchführt kann es besonders in großen Projekten passieren, dass versucht wird, die gleiche Datei mehrmals per #include einzufügen. Die einzufügende Datei kann jedoch selbst verhindern, dass sie mehrfach inkludiert wird. Hierzu wird meist folgendes System benutzt:

```
#ifndef dateiname
#define dateiname
….Inhalt...
#endif
```

Wird versucht die Datei ein zweites mal einzubinden, wird erkannt das sie bereits einmal eingefügt wurde, und der Inhalt wird kein zweites mal eingefügt.

1.4.2 Verschachtelte Makros

Makros können auch innerhalb von Makros aufgerufen werden. Somit können zum Beispiel per Makro definierte Konstanten in anderen Makros aufgerufen werden:

```
#define PI 3,14159265
#define DURCHMESSER(r) PI*r
```

Die Makros können hierbei beliebig tief und oft verschachtelt werden.

1.4.3 Debug-Meldungen

Mit Hilfe von Makros und bedingten Anweisungen können Fehlermeldungen definiert werden, die nur in der Debug-Version einer Software vorhanden sind.

Beispiel:

```
#define DEBUG
...quelltext...
#ifdef DEBUG
errprintf(„Fehler trat auf“);
#endif
```

2 Aufruf und Bedienung des GNU C Präprozessors

Diese Anleitung bezieht sich auf den GNU-Präprozessor; selbstverständlich gibt es auch andere Compiler, entsprechende Präprozessoren, und der Aufruf erfolgt in den meisten Entwicklungsumgebungen nicht mehr per Kommandozeile.

Der Gnu Präprozessor kann einzeln aufgerufen werden, oder mit einem entsprechenden Parameter beim Aufruf des Compilers.

In der erste Möglichkeit ist der direkte Aufruf durch den Befehl `cpp` in der Kommandozeile.

Der C-Compiler selbst wird mit `gcc` aufgerufen; wird dem Aufruf die Option `-E` angehängt, wird lediglich der Präprozessor ausgeführt.

Zu beachten ist das in einigen Fällen weitere Parameter übergeben werden müssen. So übersetzt der Präprozessor Trigraphzeichen nur, wenn ihm der Parameter `-trigraph` beim Aufruf mitgegeben wird.

In beiden fällen gilt, dass zunächst der Name der zu kompilierenden Datei, danach Optional der Name der Zieldatei eingegen wird.

2.1 Beispiel zum Ausprobieren

Hier nun einige Beispiele für die Präprozessor-Direktiven zum selber testen. Diese sollen nur als einfache Demonstration und als Anregung für eigene praktische Versuche gelten.

2.1.1 Beispiel für __VA_ARGS__

Ein Macro für eine eigene Aufgabe-Funktion.

```
#include <stdio.h>
#define MYPRINT(string, ...) printf("Dies ist mein printing:");\
printf(string, __VA_ARGS__)\
;printf("\n")


  int main(void)
  {
     MYPRINT("Hier sind 3 Strings: %s %s %s", "Eins", "Zwei",
"Drei");
     MYPRINT("Hier sind 2 Strings: %s %s", "Eins", "Zwei");
  }
```

2.1.2 #define

Ein Beispiel, das nur für den Präprozessor geeignet ist und keinen wirklichen C-Quelltext ausgibt. Hinweis: hier ist die Konvention, das Makros aus Großbuchstaben bestehen nicht beachtet; der Aufruf von W3 zeigt, wozu das führen kann.

```
#define W Dies ist ein wichtiger Text
#define W2 W, wirklich.
```

```
#define Wort Problem
#define W3 So ein Wort ist schoen.

 int main(void)
  {
     W2
     W3
  }
```

2.1.3 Bedingte Ausgaben und Nutzung vordefinierter Makros

Hinweis: man setzte Debug auf 1, 2 oder 3 um zu sehen, welchen effekt es auf die Ausgabe hat.

```
#define DEBUG
#define WO printf("Ich bin in der Funktion %s in Zeile %i in Datei
%s\n"\
, __func__, __LINE__, __FILE__);

void F1(void){
#ifdef DEBUG
#if DEBUG >= 1
     WO;
#endif
#endif
}

void F2(void){
#ifdef DEBUG
#if DEBUG >= 2
     WO;
#endif
#endif
}

void F3(void){
#ifdef DEBUG
     WO;
#endif
```

```
}

 int main(void)
  {
     F1();
     F2();
     F3();
  }
```

2.1.4 Quellen

http://www.imb-jena.de/~gmueller/kurse/c_c++/c_trigra.html

http://www.imb-jena.de/~gmueller/kurse/c_c++/c_premac.html

http://en.wikipedia.org/wiki/C99

http://en.wikipedia.org/wiki/C_preprocessor

http://www.cs.hs-rm.de/~reith/lehre/krypto0708/assets/programmierrichtlinien.pdf

http://de.wikibooks.org/wiki/C-Programmierung:_Präprozessor

http://openbook.galileocomputing.de/c_von_a_bis_z/index.htm

http://publications.gbdirect.co.uk/c_book/chapter7/