Präprozessor & C-C++ Mixin

Nicht von mir aber hilfreich!

Inhaltsverzeichnis

Teil I. Präprozessor

Der Präprozessor ist ein mächtiges und gleichzeitig fehleranfälliges Werkzeug, um bestimmte Funktionen auf den Code anzuwenden, bevor er vom Compiler verarbeitet wird.

Direktiven[1]

Die Anweisungen an den Präprozessor werden als Direktiven bezeichnet. Diese Direktiven stehen in der Form `#Direktive Parameter` im Code. Sie beginnen mit # und müssen nicht mit einem Semikolon abgeschlossen werden. Eventuell vorkommende Sonderzeichen in den Parametern müssen nicht escaped werden.

#include

Include-Direktiven sind in den Beispielprogrammen bereits vorgekommen. Sie binden die angegebene Datei in die aktuelle Source-Datei ein. Es gibt zwei Arten der #include-Direktive, nämlich

```
#include <Datei.h>
//bzw
 #include "Datei.h"
```

Die erste Anweisung sucht die Datei im Standard-Includeverzeichnis des Compilers, die zweite Anweisung sucht die Datei zuerst im Verzeichnis, in der sich die aktuelle Sourcedatei befindet; sollte dort keine Datei mit diesem Namen vorhanden sein, sucht sie ebenfalls im Standard-Includeverzeichnis.

#define

Für die #define-Direktive gibt es verschiedene Anweisungen. Die erste Anwendung besteht im Definieren eines Symbols mit `#define SYMBOL` wobei SYMBOL jeder gültige Bezeichner in C sein kann. Mit den Direktiven #ifdef bzw. #ifndef kann geprüft werden, ob diese Symbole definiert wurden. Die zweite Anwendungsmöglichkeit ist das Definieren einer Konstante mit `#define KONSTANTE Wert` wobei KONSTANTE wieder jeder gültige Bezeichner sein darf und Wert ist der Wert oder Ausdruck durch den KONSTANTE ersetzt wird. Insbesondere wenn arithmetische Ausdrücke als Konstante definiert sind, ist die Verwendung einer Klammer sehr ratsam, z.B.: `#define ERDBESCHLEUNIGUNG (9.80665)`. Zwischen dem Namen der Konstante und einer evtl. öffnenden Klammer des Wertes muss mindestens ein Leerzeichen stehen. Die dritte Anwendung ist die Definition eines Makros mit `#define MAKRO(parameter ...)Ausdruck` wobei MAKRO der Name des Makros ist und Ausdruck den Ersetzungstext für das Makros darstellt. Die öffnende Klammer für die Parameter muss unmittelbar auf den Makronamen folgen. Wird das Makro benutzt, werden die konstanten Textteile des Ausdruckes unverändert übernommen, Vorkommen der Parameter werden durch die Parameter-Werte des jeweiligen Makro-Aufrufes ersetzt.

[1] https://de.wikibooks.org/wiki/C-Programmierung:_Pr%C3%A4prozessor

Sowohl der Gesamtausdruck als auch alle Vorkommen der Parameter sollten in Klammern stehen, da sich sonst je nach Umgebung des Makro-Aufrufes eine unerwartete Rangfolge der Operatoren ergeben kann.
Wird beispielsweise ein Makro MAX mit den Parametern a und b definiert `#define MAX(a,b)(( a >= b)? (a): (b))` kann man dieses später verwenden, z.B. mit `maximum = MAX(5,eingabe);`
In diesem Fall wird also 5 als aktueller Text für den Parameter a angegeben und eingabe als Text für den Parameter b.

Die Ersetzung ergibt dann `maximum = ((5 >= eingabe)? (5): (eingabe));`

#undef

Die Direktive #undef löscht ein mit define gesetztes Symbol. Syntax: `#undef SYMBOL`

#ifdef

Mit der #ifdef-Direktive kann geprüft werden, ob ein Symbol definiert wurde. Falls nicht, wird der Code nach der Direktive nicht an den Compiler weitergegeben. Eine #ifdef-Direktive muss durch eine #endif-Direktive abgeschlossen werden.

#ifndef

Die #ifndef-Direktive ist das Gegenstück zur #ifdef-Direktive. Sie prüft, ob ein Symbol nicht definiert ist. Sollte es doch sein, wird der Code nach der Direktive nicht an den Compiler weitergegeben. Eine #ifndef-Direktive muss ebenfalls durch eine #endif-Direktive abgeschlossen werden.

#endif

Die #endif-Direktive schließt die vorhergehende #ifdef-, #ifndef-, #if- bzw #elif-Direktive ab. Syntax:

```
#ifdef SYMBOL
// Code, der nicht an den Compiler weitergegeben wird
#endif

#define SYMBOL
#ifndef SYMBOL
// Wird ebenfalls nicht kompiliert
#endif
#ifdef SYMBOL
// Wird kompiliert
#endif
```

Solche Konstrukte werden häufig verwendet, um Debug-Anweisungen im fertigen Programm von der Übersetzung auszuschließen oder um mehrere, von außen gesteuerte, Übersetzungsvarianten zu ermöglichen.

#error

Die #error-Direktive wird verwendet, um den Kompilierungsvorgang mit einer (optionalen) Fehlermeldung abzubrechen. Syntax: `#error Fehlermeldung`. Die Fehlermeldung muss nicht in Anführungszeichen stehen.

#if

Mit #if kann ähnlich wie mit #ifdef eine bedingte Übersetzung eingeleitet werden, jedoch können hier konstante Ausdrücke ausgewertet werden.

```
#if (DEBUGLEVEL >= 1)
#  define print1 printf
#else
#  define print1(...) (0)
#endif

#if (DEBUGLEVEL >= 2)
#  define print2 printf
#else
#  define print2(...) (0)
#endif
```

Hier wird abhängig vom Wert der Präprozessorkonstante DEBUGLEVEL definiert, was beim Aufruf von print2() oder print1() passiert.

Der Präprozessorausdruck innerhalb der Bedingung folgt den gleichen Regeln wie Ausdrücke in C, jedoch muss das Ergebnis zum Übersetzungszeitpunkt bekannt sein.

defined

defined ist ein unärer Operator, der in den Ausdrücken der #if und #elif Direktiven eingesetzt werden kann.

```
#define FOO
#if defined FOO || defined BAR
#error "FOO oder BAR ist definiert"
#endif
```

Die genaue Syntax ist `defined SYMBOL` Ist das Symbol definiert, so liefert der Operator den Wert 1, anderenfalls den Wert 0.

#elif

Ähnlich wie in einem else-if Konstrukt kann mit Hilfe von #elif etwas in Abhängigkeit einer früheren Auswahl definiert werden. Der folgende Abschnitt verdeutlicht das.

```
#define BAR
#ifdef FOO
```

```
#error "FOO ist definiert"
#elif defined BAR
#error "BAR ist definiert"
#else
#error "hier ist nichts definiert"
#endif
```

Der Compiler würde hier BAR ist definiert ausgeben.

#else

```
#ifdef FOO
#error "FOO ist definiert"
#else
#error "FOO ist nicht definiert"
#endif
```

#else dient dazu, allen sonstigen nicht durch #ifdef oder #ifndef abgefangenen Fälle einen Bereich zu bieten.

#pragma

Bei den #pragma Anweisungen handelt es sich um compilerspezifische Erweiterungen der Sprache C. Diese Anweisungen steuern meist die Codegenerierung. Sie sind aber zu sehr von den Möglichkeiten des jeweiligen Compilers abhängig, als dass man hierzu eine allgemeine Aussage treffen kann.

Beispiele

Print Token

```
#define PRINT_TOKEN(token) printf(#token " is %d", token)
```

Swap (Multiline)

```
#define SWAP(a, b)  {                   \
                        a ^= b;         \
                        b ^= a;         \
                        a ^= b;         \
                    }
```

Swap

Mehrere Statements werden mit do {} while(0) trick als Macro verwendet

```
#define SWAP(a, b)  do { a ^= b; b ^= a; a ^= b; } while ( 0 )
```

Include Guard

```
#ifndef _FILE_NAME_H_
#define _FILE_NAME_H_

/* code */

#endif // #ifndef _FILE_NAME_H_
```

Array Size

```
#define ARRAYSIZE(arr) (sizeof(arr) / sizeof(arr[0]))
```

Increment

```
#define INCREMENT(x) x++
```